Reengineering Object Oriented Software Systems for a better Maintainability

Soumia Zellagui

THESIS

To obtain the dregree of PHD - Doctor

Awarded by **University of Montpellier**

Prepared at **I2S*** Graduate School,
LIRMM Research Unit, MAREL Team.

Speciality: **Software Engineering**

Defended by **Zellagui Soumia**
zellagui@lirmm.fr

Reengineering Object Oriented Software Systems for a better Maintainability

Defended on 05 July 2019 in front of a jury composed of:

Mireille Blay-Fornarino, PU, University of Nice	Reviewer
Antoine Beugnard, P, Mines-Telecom Atlantique Institute	Reviewer
Bernard Coulette, PU, University of Toulouse-Jean Jaurès	President & Examiner
Chouki Tibermacine, MdC HDR, University of Montpellier	Director
Christophe Dony, PU, University of Montpellier	Co-director
Hinde Lilia Bouziane, MdC, University of Montpellier	Co-director

* **I2S:** Information, Structures and Systèmes.

Dedication

“ This thesis is dedicated to the memory of my father ”

Acknowledgment

My advisor, Chouki Tibermacine, has had the most profound influence on me as a researcher. When I started my thesis I was a blank page. I learned a lot from him and I have found working with him very rewarding. He was always on my side with his ideas, advices and encouragements and I will never forget his kindness and comprehensiveness. I felt blessed and lucky to have him as one of my supervisors. I thank also my two other supervisors, Hinde Bouziane and Christophe Dony who was always kind to me and comprehensive.

During last six months, I established a great working relationship with Ghizlane El-Boussaidi who gave me the opportunity to work with her in École de Technologie Supérieure de Montréal. She has always been ready to engage in long and rich research discussions and to provide her unique insight. I hope this relationship continues in future.

I would like to express my deepest gratitude to Mireille Blay-Fornarino, Bernard Coulette and Antoine Beugnard for making me the honor of accepting to evaluate my thesis.

I would like to thank everybody in my research team MAREL. They made me feel like home during these three years.

It should be noticed that this thesis would have not been possible without the funding from the Algerian Ministry of Higher Education and Scientific Research, which allowed me to stay completely focused on my research.

I owe my deepest gratitude to my Mother who has always been my friend and guide. I thank her for believing in me and encouraging me to follow my dreams. I thank my sisters for their incredible support, I could have never achieved so much without them. I also thank all my friends for the pleasant moments we had together.

Résumé

Les systèmes logiciels existants représentent souvent des investissements importants pour les entreprises qui les développent avec l'intention de les utiliser pendant une longue période de temps. La qualité de ces systèmes peut se degrader au fil du temps en raison des modifications complexes qui leur sont incorporées. Pour faire face à une telle dégradation lorsqu'elle dépasse un seuil critique, plusieurs stratégies peuvent être utilisées. Ces stratégies peuvent consister à: 1) remplacer le système par un autre développé à partir de zéro, 2) poursuivre la maintenance (massive) du système malgré son coût, ou 3) conduire une réingénierie du système. Le remplacement et la maintenance massive ne sont pas des solutions adaptées lorsque le coût et le temps doivent être pris en compte, car elles nécessitent un effort considérable et du personnel pour assurer la mise en oeuvre du système dans un délai raisonnable. Dans cette thèse, nous nous intéressons à la solution de réingénierie. En général, la réingénierie d'un système logiciel inclut toutes les activités après la livraison à l'utilisateur pour améliorer sa qualité. Cette dernière est souvent caractérisée par un ensemble d'attributs de qualité. Parmi ces attributs, nous nous intéressons à la maintenabilité. Cette dernière est caractérisée par un ensemble de caractéristiques telles que la modifiabilité, la compréhensibilité et la modularité. Afin d'améliorer la modifiabilité, nous proposons, dans la première contribution, de migrer les systèmes logiciels orientés objets vers des systèmes orientés composants. Contrairement aux approches existantes qui considèrent un descripteur de composant comme un cluster de classes, chaque classe du système existant sera migrée vers un descripteur de composant. Afin d'améliorer la compréhensibilité, qui a un impact sur la maintenabilité, nous proposons, dans la seconde contribution, une approche pour la reconstruction de modèles d'architecture d'exécution (graphes d'objets) des systèmes orientés objets et la gestion de la complexité des modèles résultants. Les modèles (graphes) générés avec notre approche ont les caractéristiques suivantes: les nœuds sont étiquetés avec des durées de vie et des probabilités d'existence permettant 1) une visualisation des modèles avec un niveau de détail, et 2) de cacher/montrer la structure interne des noeuds. Afin d'améliorer la modularité, et donc la maintenabilité, des systèmes logiciels orientés objets, nous proposons, dans la troisième contribution, une approche d'identification des modules et des services dans le code source de ces systèmes. Dans cette approche, nous soutenons l'idée considérant la structure composite comme la structure principale du système. Celle-ci doit être conservée lors du processus de modularisation, le composant et ses composites doivent être dans le même module.

Les travaux de modularisation existants qui ont cette même vision, supposent que les relations de composition entre les éléments du code source sont déjà disponibles, ce qui n'est pas toujours vrai. Dans notre approche, l'identification des modules commence par une étape de reconstruction de modèles d'architecture d'exécution du système étudié. Ces modèles sont exploités pour identifier des relations de composition entre les éléments du code source du système étudié. Une fois ces relations ont été identifiées, un algorithme génétique conservatif aux relations de composition est appliqué sur le système pour identifier des modules. En dernier, les services fournis par les modules sont identifiés à l'aide des modèles de l'architecture d'exécution du système logiciel analysé. Quelques expérimentations et études de cas ont été réalisées pour montrer la faisabilité et le gain en modifiabilité, compréhensibilité et modularité sur de vrais logiciels analysés avec nos propositions.

Abstract

Legacy software systems often represent significant investments for the companies that develop them with the intention of using them for a long period of time. The quality of these systems can be degraded over time due to the complex changes incorporated to them. In order to deal with these systems when their quality degradation exceeds a critical threshold, a number of strategies can be used. These strategies can be summarized in: 1) discarding the system and developing another one from scratch, 2) carrying on the (massive) maintenance of the system despite its cost, or 3) reengineering the system. Replacement and massive maintenance are not suitable solutions when the cost and time are to be taken into account, since they require a considerable effort and staff to ensure the system conclusion in a moderate time. In this thesis, we are interested in the reengineering solution. In general, software reengineering includes all activities following the delivery to the user to improve the software system quality. This latter is often characterized with a set of quality attributes. Among those, we are particularly interested in maintainability. In turn, maintainability is characterized with a set of characteristics such as modifiability, understandability and modularity. In order to improve modifiability, we propose to migrate object-oriented legacy software systems into equivalent component based ones. Contrary to exiting approaches that consider a component descriptor as a cluster of classes, each class in the legacy system will be migrated into a component descriptor. In order to improve understandability, which has a direct impact on maintainability, we propose an approach for recovering runtime architecture models of object-oriented legacy systems and managing the complexity of the resulted models. The models recovered by our approach have the following distinguishing features: Nodes are labeled with lifespans and empirical probabilities of existence that enable 1) a visualization with a level of detail. 2) the collapsing/expanding of objects to hide/show their internal structure. In order to improve modularity, and thus maintainability, of object-oriented software systems, we propose an approach for identifying modules and services in the source code. In this approach, we believe that the composite structure is the main structure of the system that must be retained during the modularization process, the component and its composites must be in the same module. Existing modularization works that has this same vision assumes that the composition relationships between the elements of the source code are already available, which is not always obvious. In our approach, module identification begins with a step of runtime architecture models recovery. These models are

exploited for the identification of composition relationships between the elements of the source code. Once these relationships have been identified, a composition conservative genetic algorithm is applied on the system to identify modules. Lastly, the services provided by the modules are identified using the runtime architecture models of the software system. Some experimentations and case studies have been performed to show the feasibility and the gain in modifiability, understandability and modularity of the software systems studied with our proposals.

Contents

Figures list

Tables list

Chapter

1

Introduction

Contents

1.1 Context

A software system is defined by Sommerville [Sommerville 2011] as: *"a number of separate programs, configuration files, which are used to set up these programs, system documentation, which describes the structure of the system, and user documentation, which explains how to use the system"*. Therefore, a software system is not only a computer program but also all associated configuration and documentation data needed to use this system correctly. For many years, software systems increasingly influenced almost all areas of society and have become fundamental in performing a wide variety of tasks.
Companies develop software systems with the intention of using them for a long period of time in order to get a return on the costs spent on their development. The average lifetime of software systems is more than 10 years with a minimum of two years and a maximum of thirty as stated by Tamai et al [Tamai 1992]. Old systems are called *"legacy systems"*. Legacy systems characteristics are summarized by Crotty et al [Crotty 2017] in: business critical, long time lived, developed using outdated technologies, with poor documentation if available, degraded structure, and needs a lot of time for maintenance tasks even small ones.
The increasing dependence on computers and software systems at all levels of the society requires a continuous integration of new functionalities, that are sometimes complex, to legacy systems. Due to the complex incorporated functionalities in these systems over time, including new functionalities in the future will be more and more difficult for developers who spend a large amount of their time reading the code and the accompanying system documentations and models, in order to understand the system's structure and organization. Therefore, progressive changes of software systems lead to their *quality degradation* since they become more complex [Lehman 1997].
Software quality plays a crucial role for the competitiveness and survival of companies because it has a significant impact on the costs arising during development, maintenance and use of software. Several quality models have been proposed in the literature [Boehm 1976], [McCall 1977], [Grady 1992], [Dromey 1995], [ISO.25010 2008] in order to define specific requirements for quality. In these models, quality is categorized into a set of characteristics which in turn are broken down into sub-characteristics. These characteristics are known as *"quality attributes"*. General software quality attributes include maintainability, modifiability, understandability, modularity, reusability, interoperability, performance, reliability, security, etc.
A number of solutions have been proposed to deal with legacy software systems when their quality degradation exceeds a critical threshold. These solutions can be summarized in: 1) replacing the system by another one developed from scratch, 2) carrying on the (massive) maintenance of the system despite its cost, or 3) resorting to the system reengineering [Bennett 1999]. Several decisional models have been

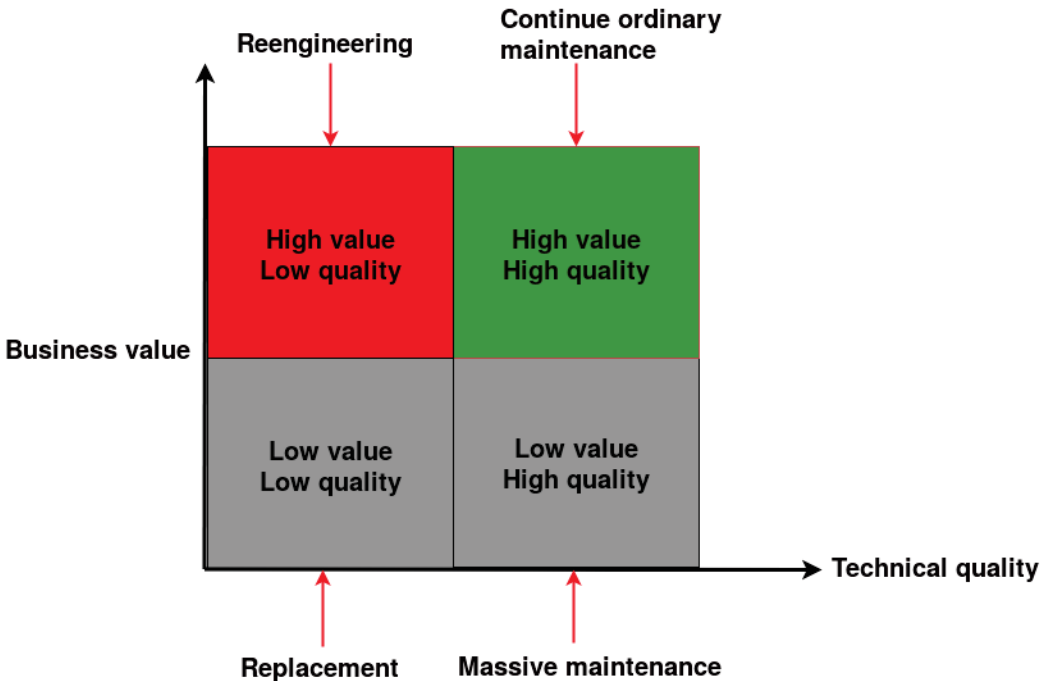

Figure 1.1: Legacy system decisional matrix [Lucia 2001].

proposed [Lucia 2001,Ransom 1998,Alkazemi 2013] in order to reach a decision about the best solution, from the aforementioned ones, to consider. They are based on both business and technical values. Business values concern user satisfaction. Technical values are measures calculated on the system (e.g, size, complexity, etc). The output of these models is plotted on a decisional matrix that indicates a recommended solution as depicted in Figure 1.1.
The first two solutions, replacement and massive maintenance, are not suitable solutions when the cost and time are to be taken into account, since they require a considerable effort and staff to ensure the system conclusion in a moderate time. Therefore, in this thesis, we are interested in the reengineering solution (depicted in red in Figure 1.1).
Software reengineering is the process of generating evolvable systems [Seacord 2003] and therefore extending the life time of a legacy software system. Several definitions have been proposed to describe the software reengineering process. The most commonly used ones are presented in Table 1.1.
These definitions share the fact that software reengineering consists of taking a legacy system whose quality is degraded and extracting knowledge on its internal structure or business process and reconstituting it in a new form using new technologies. In general, software reengineering includes all activities following the delivery to the customer to improve the software system quality attributes. We use in this thesis the definition and the conceptual model of Chikofsky and Cross [Chikofsky 1990] since numerous publications use the author's reengineering taxonomy.

Table 1.1: Software reengineering state of the art definitions.

[Chikofsky 1990]	"...the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."
[Arnold 1993]	" any activity that: (1) improves one's understanding of software, or (2) prepares or improves the software itself, usually for increased maintainability, reusability, or evolvability. In this definition, the term software includes, in addition to source code, documentation, graphical pictures and analyses."
[Tilley 1995]	" Reengineering is the systematic transformation of an existing system into a new form to realize quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule or risk to the consumer. "
[McClure 1992]	" Reengineering is the process of examining an existing system (program) and/or modifying it with the aid of automated tools to: improve its future maintainability, upgrade its technology, extend its life expectancy, capture its components in a repository where CASE tools can support it, and increase maintenance productivity."

In this definition, software reengineering process encompasses a combination of three subprocesses: examination or understanding subprocess in which knowledge about the system is acquired, alteration subprocess which consists of changing the system, and the reconstitution subprocess in which the modifications are realized. These subprocesses correspond respectively to *reverse engineering*, *restructuring* and *forward engineering*. These three subprocesses are depicted in Figure 1.2 and defined below:

1. *Reverse engineering (examination):* is the process of analyzing low level abstractions, e.g., source code, of the subject system in order to extract high level abstractions, e.g., design. Several synonyms exist for the reverse engineering process in the literature such as: recovery [Solms 2015, Lutellier 2015], reconstruction [Schmidt 2018, Ahn 2018], identification [Shahmohammadi 2010, Athanasopoulos 2017], mining [Shatnawi 2015] and extraction [Mazlami 2017].

2. *Restructuring (alteration):* also known by the name *"refactoring"*. Refactoring is defined by Martin Fowler [Fowler 1999] as *"the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure"*. Refactoring always takes place within one level of abstraction, e.g., code-to-code, in order to fix this abstraction's *bad smells*. Bad smells are symptoms of poor quality [Fowler 1999] that may hinder code maintainability [Kruchten 2012]. Whenever they appear, they indicate that the code or/and the design should be reexamined. An example of bad smell is *Long Method* which is a method unduly long in terms of lines of code and with lots of parameters and local variables. This method should be decomposed for ease of maintainability.

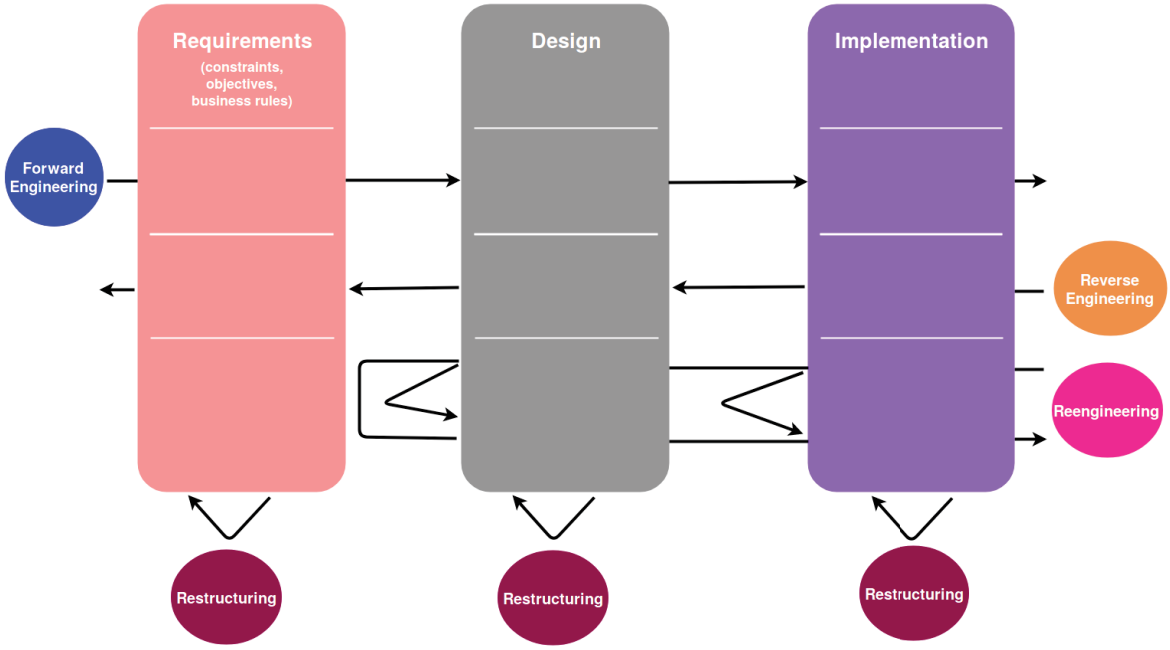

Figure 1.2: Chikofsky and Cross [Chikofsky 1990] conceptual model of software reengineering.

3. *Forward engineering (reconstitution):* is the classical software development process. It consists of moving from a high level of abstraction (e.g., requirements) to a lower level (e.g., source code) [Van Vliet 1993].

1.2 Problem Statement

Several studies have presented evidence that the major expense in the life cycle of a software system is maintenance [Schneidewind 1987, Benaroch 2013, Galorath 2006], that at least 50% of the total life cycle is devoted to maintenance, depending on the references. Two real life examples on maintenance cost are given in [Anjos 2017]:

- **Example 01:** *In 1997, the Associated Press today reports that Robin Guenier, head of the UK's TaskForce 2000, estimates that Y2K reprogramming efforts will cost Britain $50 billion dollars, three times the guesstimates of business consultants and computer service companies. Guenier suggested that 300,000 people may be required to tackle the problem. Coincidentally, that number is roughly equivalent to the number of full-time computer professionals in the UK.*

- **Example 02:** *In 2000, the city of Toronto lost out on nearly $700,000 in pet fees because nearly half of Toronto's dog and cat owners were never billed due an outdated computerized billing system. The staff who knew how to run the computerized billing system was laid off. [...] Only one city employee ever understood the system well enough to debug it when problems arose, and that employee was also laid off in 2000 due to downsizing, leaving no one to get things going again when the system ran into trouble and collapsed.*

Maintainability is the term used to describe the quality attribute concerned with software maintenance. It represents *" the ease with which a software system or component can be modified to correct faults, improve performance or adapt it to a changed environment"* [ISO.25010 2008]. Since a system with a high degree of maintainability leads to low maintenance costs, maintainability is the most targeted quality attribute by reengineering processes [Abdellatif 2018]. Maintainability is characterized in several quality models by a set of characteristics. From these characteristics, we are particularly interested in modifiability, understandability and modularity.
Modifiability refers to *" the degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality "* [ISO.25010 2008]. Since maintenance costs decrease when a system is developed in a way that future changes will be relatively easy to implement, we believe that modifiability has a direct impact on maintainability.
Understandability has a direct impact on maintainability. That is, if a software is sufficiently understood, it can be properly maintained [Cornelissen 2009a]. As stated by Biggerstaff et al [Biggerstaff 1993], *"a person understands a program when he or she is able to explain the program, its structure, its behavior, its effects on its operation context, and the relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program"*. Therefore, understanding is related to the ease of reading and correctly interpreting the informations contained in the software system.
Another quality attribute that has a direct impact on maintainability, and even understandability, is modularity. It is defined as *"the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components"* [ISO.25010 2008]. High cohesion and low coupling are drivers for good modularity. Cohesion refers to the intra-component dependencies and coupling refers to the inter-component dependencies. It is a known fact that modularity facilitates maintainability [Davis 1990] since modules are independent from each other. This independence allows maintaining different parts of the system in parallel by distinct maintainers and restricting change propagation.

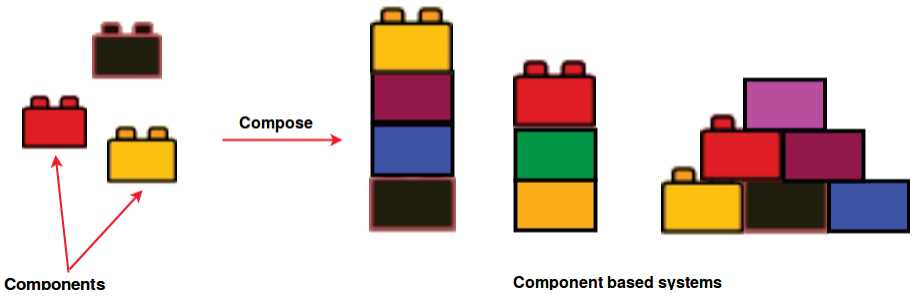

Figure 1.3: Component Based development principle.

The Component Based (CB) development paradigm [Bertolino 2005] has been recognized as one of the paradigms that promote modifiability and thus maintainability. The idea behind this paradigm is a metaphor of the LEGO game. The principle is illustrated in Figure 1.3: on the left of the figure, there are software components which correspond, according to Szyperski's definition [Szyperski 1999], to *" a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties"*. As stated in this definition, components can be composed in order to create new systems. By analogy, with some LEGO pieces, it is possible to create pieces as complex as a car, a house, etc.
Several approaches have been proposed for migrating legacy object oriented software systems into equivalent component based ones in order to benefit from the advantages of the CB paradigm. These approaches consider the component descriptor as a cluster of classes. Some of these approaches generate clusters/components with shared classes/interfaces. When these components are reused/composed to create new systems, duplicated code can occur in these new systems. Duplicated code will make the maintainability task difficult especially when bugs are detected in this code. Moreover, in the approaches where no classes/interfaces are duplicated in several components, if a user wants to develop a new system using an independent class or subset of classes in a cluster, it is required to use the entire cluster. This implies that the new developed system contains unnecessary code. The maintenance of this latter is a waste of time, this led us to ask the following research question:

Research Question 1. Does considering each class in the object oriented legacy system as a component descriptor contribute in improving modifiability?

This is the first research problem/question that is studied in this work (**RQ1**).

Furthermore, several reverse engineering approaches have also been proposed in order to improve understandability, and thus maintainability. This improvement is achieved by recovering a high level view, an architecture model, of the system's structure and behavior. These architecture models are used to acquire a global understanding of the system instead of wasting time looking at source code artifacts.

Most of these approaches target the recovery of class-based models of the system under study. In our context, we are interested in recovering the runtime architecture, which is composed of the system's concrete running entities (objects) and dependencies between them. The importance of having the runtime architecture in order to improve understandability was stressed in several experimental studies such as the one of Lee et al [Lee 2008]. In this study, one participant stated: *"draw how objects connect to each other at runtime when I want to understand code that is unknown; an object diagram is more interesting than a class diagram, as it expresses more how it functions"*. While the structure and relationships between objects are implicit in static views, they are explicit in the runtime architecture. This facilitates handling comprehension tasks that require knowledge about object interactions [Ammar 2012, Lee 2008]. Only few approaches target the recovery of runtime architectures. They produce valuable models for understanding the structure of the system during its execution. However, they fail most of the time to provide models of a reasonable size which can be visualized by humans/developers, and this is particularly true for large (legacy) software systems. This led us to consider the following research question:

Research Question 2. How to recover the runtime architecture of legacy object oriented systems and how to manage the complexity of the recovered architecture?

This is the second research question that is investigated in this work (**RQ2**).

Moreover, the improvement of legacy system modularity has been widely studied in the literature. Most of the time, remodularization approaches are based on clustering techniques in order to identify highly cohesive and lowly coupled module candidates. In general, cohesion and coupling are measured based on structural relationships between source code artifacts. To the best of our knowledge, most of existing remodularization approaches consider that the different types of structural relationships (e.g., field typing, method parameter typing, method invocation) between source code artifacts are equivalent, which is not always true and not accurate enough. We argue that structural relationships types should be differentiated. Moreover, we believe that source code artifacts with composition relationships should logically be clustered together in the same module. To the best of our knowledge, remodularization approaches that have this same point of view, classes/interfaces with a composition relationship should be clustered in the same module, suppose that composition relationships are already available in the form of class-based models, which is not always obvious. This led us to ask the following question:

Research Question 3. Does differentiating structural relationships and grouping artifacts with composition relationships in the same module produce modular solutions?

This was the third and last problem tackled in this work (**RQ3**).

1.3 Thesis Contributions

The ultimate goal of this thesis is the improvement of maintainability of legacy software systems. To that end, we propose three approaches that are briefly discussed below.

1.3.1 Migrating Object-Oriented Software Systems into Component-based equivalent ones

To answer **RQ1**, we proposed a source code restructuring approach to improve object oriented software systems modifiability. In particular, this approach enhances decoupling by considering that some dependencies between classes should be set through abstract types (interfaces) like in component based systems. In addition, some anticipated instantiations of these classes buried in the source code are extracted and replaced by declarative statements (like connectors in CB applications) which are processed by a dependency injection mechanism. For doing so, a set of modifiability defects has been defined. These defects are first detected in the source code. Then, some refactoring operations are applied for their elimination. At the end of the process, each class in the object oriented system conforms to a component descriptor. This approach is discussed in detail in Chapter 3.

1.3.2 Recovering the Runtime Architecture of Object-Oriented Software Systems and Managing its Complexity

To answer **RQ2**, we developed an approach to build runtime architecture models of object oriented systems. The approach combines static and dynamic analysis to build an object graph that includes information regarding probabilities of allocation site execution and lifespans of objects. This information is used to manage the complexity of the recovered object graphs by making it possible for the designer to focus for example on the most likely and durable objects. In addition, composition/ownership relations between objects are exploited to embed composite structures into the object graph nodes. This enables to support a hierarchical visualization of the recovered architecture. Understandability improvement brought by the approach is discussed in Chapter 4.

1.3.3 Identifying Modules and Services from the Source Code of Object-Oriented Software Systems

To answer **RQ3**, we proposed an approach for improving the modular structure of object-oriented software systems by identifying modules and services in the source code. In contrast to existing works in the literature, the process starts by a step of runtime models recovery. These models represent the concrete interacting objects that compose the running system and their inter-dependencies. These mod- els are exploited in order to identify composition relationships objects. Once these composition relationships have been identified, a composition conservative genetic algorithm is applied on the software system in order to identify modules. At last, services that allow modules to communicate are identified, based on the runtime models, in order to further improve decoupling. Modularity improvement brought by the approach is discussed in Chapter 5.

1.4 Thesis Organization

The rest of this thesis is organized into five chapters:

- **Chapter 02:** discusses the state of the art related to the problem of improving quality attributes of object oriented legacy systems.
- **Chapter 03:** presents the first contribution on modifiability improvement which deals with **RQ1**.
- **Chapter 04:** presents the second contribution on understandability improvement which deals with **RQ2**.
- **Chapter 05:** presents the third contribution on modularity improvement which deals with **RQ3**.
- **Chapter 06:** discusses conclusions and future directions.

1.5 Publications

The following accepted or submitted papers are partial outputs of this thesis. Table 1.2 organizes them according to the thesis contributions.

1.5.1 International Journals

1. Soumia Zellagui, Chouki Tibermacine, Hinde Bouziane and Christophe Dony. *"Identification of Modules and Services in the Runtime Architectures of Object-oriented Software Systems"*. Submitted to Information & Software Technology journal.

2. Soumia Zellagui, Chouki Tibermacine, Hinde Bouziane and Christophe Dony. "A Method for the Automatic Recovery and Complexity Management of Runtime Architectures of Medium and Large Sized OO Software Systems". Under review (revision) for the Automated Software Engineering journal.

1.5.2 International Conferences

3. Soumia Zellagui, Chouki Tibermacine, Ghizlane El-Boussaidi, Hinde Bouziane, Abdelhak-Djamel Seriai and Christophe Dony. *"Recovering Runtime Architecture Models and Managing their Complexity using Dynamic Information and Composite Structures"*. In proceedings of 33rd ACM/SIGAPP Symposium On Applied Computing (SAC 2018), PAU, France, April 9 - 13, 2018 **(Acceptance rate: 25%)**

4. Soumia Zellagui, Chouki Tibermacine, Hinde Bouziane, Abdelhak-Djamel Seriai and Christophe Dony. *"Refactoring Object-Oriented Applications towards a better Decoupling and Instantiation Unanticipation"*. In proceedings of 29th International Conference on Software Engineering and Knowledge Engineering (SEKE 2017), Wyndham Pittsburgh University Center, Pittsburgh, USA, July 5 - July 7, 2017 **(Acceptance rate: 35%)**

1.5.3 French-Speaking Conferences

5. Soumia Zellagui, Chouki Tibermacine, Hinde Bouziane, Abdelhak-Djamel Seriai and Christophe Dony. "Recovering Runtime Architecture of Object Oriented Software". Doctiss (2018), Montpellier, France, 14 June, 2018

6. Soumia Zellagui, Chouki Tibermacine, Hinde Bouziane, Abdelhak-Djamel Seriai and Christophe Dony. "Refining the Reconstructed Runtime Architecture of Object Oriented Software". Journée RIMEL (2016), Nantes, France, 08 Dec, 2016

7. Soumia Zellagui and Joffray Braga. "Refactoring des applications à objets pour un meilleur découplage et une non-anticipation des instanciations". In 5ième Conférence en IngénieriE de Logiciel (CIEL 2016), Besançon, France, 7 Jun 2016.

Table 1.2: List of publications and their distribution over thesis contributions.

	Contribution 1	**Contribution 2**	**Contribution 3**
International Journals		(2)	(1)
International Conferences	(4)	(3)	
French-Speaking Conferences	(7)	(5), (6)	

Chapter

2

State of the art

Contents

2.1 Introduction

This chapter discusses the state of the art related to maintainability improvement. The goal of this chapter is to present the main ideas and concepts of the research field, to show some similarities and differences between solutions and to reveal some shortcomings in existing works from the literature.

There are numerous works on improving maintainability of legacy systems. The described works are grouped into three categories. The first category groups works that attempt to improve maintainability by migrating the legacy system into a new programming paradigm (presented in Section 2.2). The second category groups works that improve maintainability by remaining in the object-oriented one (presented in Section 2.3. The third category groups works that improve maintainability, particularly the understandability and modularity, by recovering the implementation-level and design-level architecture of the legacy system (presented in Section 2.4). A summary discussion of the presented works is made at the end of the chapter in Section 2.5.

2.2 Improving Maintainability by Migrating to a New Paradigm

By new paradigms we mean here new ways of developing software. We focus on those that promote the maintainability quality attribute, and which are: the component-based, service-based, microservice-based and aspect-based ones.

2.2.1 Migrating to Component Based Paradigm

Component-Based (CB) paradigm has been recognized as an approach that emphasizes software maintainability. Thereby, several works have been proposed to migrate object oriented software systems into component-based ones. The migration process consists of two steps: the first step is the component based architecture recovery where components and their dependencies are identified. The second step is code transformation in which the object oriented code is transformed into an equivalent component based one. We have selected below a set of the most important (mostly cited) migration works.

- Jain et al [Jain 2001] proposed a semi-automatic approach for component identification from object oriented legacy systems. The input of the approach is two types of models of the legacy system: the *UML* class diagram and use case diagram. These models are used in order to calculate the strength of structural relationships between classes of the legacy system[1]. A hierarchical agglomerative clustering algorithm is then applied in order to identify initial components. This algorithm groups initially classes having the highest structural relationship strengths in the same component and later clusters them until an end point is reached. This point corresponds to a threshold limit of the relationship strengths. The identified components can then be refined using both manual and/or automatic heuristics. An example of heuristics is

[1]The strength is calculated based on weights given by the user to the structural relationships

moving a particular class from its component if the user feels that it is more appropriate to place it in another component. The authors reported a case study of the approach. This case study showed that maintainability is slightly improved. However, details on how its values were measured are lacking.

- In [Lee 2003], component identification, which is done manually, includes two steps. The first step consists of grouping together classes/interfaces with composition and inheritance relationships to form base components. The second step of the process consists of grouping similar base components or assigning classes, which where not assigned before to a base component, to a new component. This grouping is based on quality metrics such as cohesion which are considered as similarity measures for clustering. For each identified component, a pair of required and provided interfaces is created. The required interface is the set of all methods in other components that are called in a component. The provided interface is the set of public methods of the component called by other components. The application of the approach on an example system showed an improvement in the modularity.

- The manual identification of components in [Kim 2004] is based on the degree of dependency between use cases of a use case diagram which is supposed to be available. The dependency between two use cases is calculated taking into account several criteria such as: use cases invoked by the same actor or manipulating the same set of data. Once the dependencies are computed for every pair of use case, related use cases are allocated in the same cluster. Then, the dynamic behavior of each use case is represented by a sequence diagram, supposed to be available, which specifies the set of participating objects/classes. The participating classes in each sequence diagram are assigned to the cluster of the corresponding use case. The class diagram is then used to refine the resulted clusters by exploiting several types of relationships between classes. Unfortunately, the authors do not test the applicability of the approach on real world systems. Therefore, it is not clear if the approach contributes really in improving maintainability.

- Washizaki et al [Washizaki 2005] presented an automatic approach to extract JavaBeans components from Java systems. The input of the approach is the source code of the system and an extraction criterion that represents a functionality to be reused. A component groups the class that implements the functionality to be reused and all classes that are reachable from this class. Reachability is determined using a class-based graph, in which nodes represent classes/interfaces and edges represents structural relationships such as reference and inheritance. A class B is said reachable from class A if there is a path from A to B in the class-based graph. The approach was applied on nine systems. For these systems, the number of the extracted components

and their reusability are reported. Another semi-automatic similar approach is that of [Constantinou 2015] where the extraction task starts with the selection of an origin class. This approach focuses on the reduction of candidate component sizes.

- Allier et al [Allier 2011] proposed an approach to automate the process of migration from Object-Oriented systems to Component Based ones. The used data to identify the components are execution traces. An execution trace is a tree in which each node is the execution of a method and each edge is a method call. Once the components have been identified, the component interfaces are made operational by the use of two design patterns: Adapter and Façade. The approach was illustrated on a real system implemented in Java which is migrated into an OSGi equivalent system. However, no measurement of the improvement in maintainability was done.

- Alshara et al [Alshara 2015, Alshara 2016] proposed an approach to automatically transform Java systems into OSGi systems. This approach takes as input the source code of the system and the description of the corresponding component based architecture (obtained using ROMANTIC (Re-engineering of Object-oriented systeMs by Architecture extractioN and migraTIon to Component based ones) [Kebir 2012]). In this architecture, a component is a set of object oriented classes/interfaces. This architecture is obtained by a clustering based on a fitness function which measures the quality of a component. This latter is defined according to the component characteristics, namely autonomy, specificity and composability. These characteristics are refined into sub-characteristics which are in turn refined into component properties (e.g. required interfaces). Then, these properties are mapped to the properties of the group of classes from which the component is identified (e.g. group of classes coupling). At the end, these properties are refined into object oriented metrics (e.g. coupling metric) calculated based on structural relationships between source code artifacts. Once the component based architecture has been recovered, dependencies between classes belonging to different components are transformed into interactions via interfaces. For this, the authors were interested in two types of relations between classes. The first type is a "use" relation identifying the case when a class in a component creates an instance of another class in another component. This relation must be transformed in order to respect the component interaction principle which states that each component must hide its internal structure and behavior and provide its services without exposing the classes that implement them [Szyperski 2002]. The second type is the "inherit" relation when the parent class and child class are not in the same component. According to the point of view of the authors, this relation needs to be transformed into delegation since several component models do not support inheritance relationship between components. The ap-

plicability of the approach was tested on nine systems. The results reported the number of components in each system and the number of relations that have been transformed. No discussion or measurement of the improvement in maintainability was done.

- Shatnawi et al [Shatnawi 2016] proposed an approach that aims at recovering software components from Object-Oriented APIs. In this approach, groups of API classes that are able to form components are identified. This identification is based on the probability of classes to be reused together by clients, and the structural and behavioral relationships between classes/interfaces. The approach was applied on four APIs that are used by 100 clients. The authors stated that the improvement in the understandability is related to the percent of the number of identified components on the number of classes composing the API. That is, if this percent is small, this means that the effort spent to understand API entities is reduced since the API size is reduced.

- Starting from the component based architecture and the software execution data, which record method calls, Liu et al [Liu 2018] propose an approach to identify a set of interfaces for each component. Moreover, in order to understand how each interface actually works, a behavioral model for each identified interface is discovered. A component is defined as a set of classes. The component interfaces are identified based on the component execution data which represents all method calls in the software execution data referring to instances of the component classes. From these methods, a subset which represent methods called by methods of another component is selected. The methods in this subset are grouped according to their caller, methods called by the same method are grouped in the same interface, to form the component interfaces. In order to eliminate duplication, similar interfaces are merged based on a similarity measure proposed by the authors. Interfaces whose similarity value is superior to a threshold are merged. Once the interfaces have been identified, their contracts which defines in which order the methods should be invoked are discovered and represented by a behavioral model. The authors evaluated the approach on three systems. Since the authors are interested only in component interfaces identification, the evaluation focused on the quality of the identified interfaces.

2.2.2 Migrating to Service Based Paradigm

Service-oriented architecture (SOA) has received much popularity in the previous decade and has been adopted by many companies. The main elements that compose a service oriented architecture are highly cohesive and loosely coupled services. These services are published in a service repository by a service provider and used by service consumers as depicted in Figure 2.1.

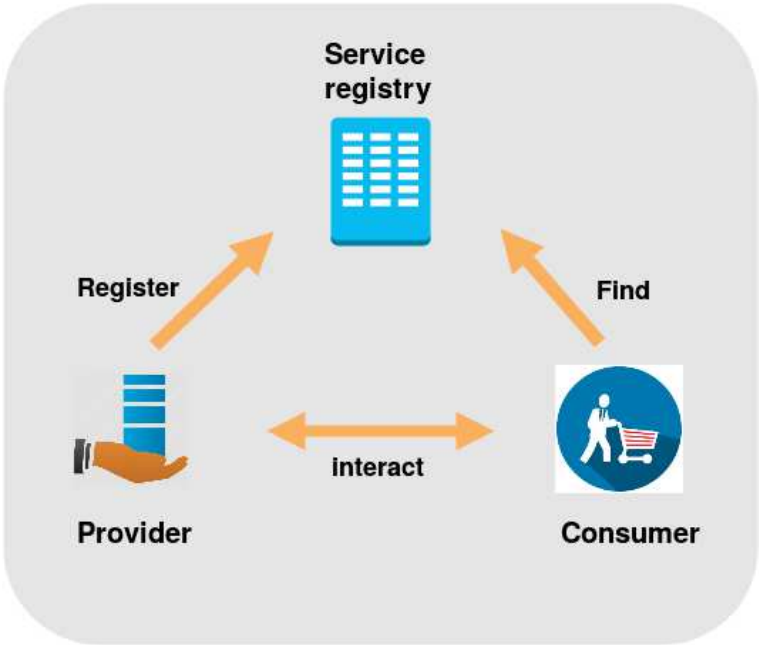

Figure 2.1: Service-oriented architecture (SOA) main elements

The modernization of legacy software towards SOA is promising since it enables sharing services between several systems which leads to a reduction of the development and maintenance costs [Griffiths 2010].
Several approaches have been reported in the literature to migrate legacy systems to SOA based ones. Most of these approaches (e.g., [Alahmari 2010, Khadka 2011, Khadka 2013]) propose only general recommendations and best practices that should be taken into consideration. They are given based on practices that have been repeatedly applied and proven to be successful. Next, we present works that provide structured approaches. Moreover, only the service identification step is discussed. The deployment of an identified service is not discussed since this step is rarely handled in related works.

- [Li 2006] developed an automatic approach made of four steps: *Architecture Recovery*, *Service Identification*, *Component Generation* and System *Transformation*.
  - *Architecture recovery* aims at reconstructing a view of the implementation-level architecture. The output of this step is two architectural models: Class/Interface Relationship Graph (CIRG) which is a direct graph that represents classes/interfaces and their different relationships (inheritance, composition, etc) and Class/Interface Dependency Graph (CIDG) which is an undirected CIRG. Unfortunately, it is not clear how the *CIRG* is exploited in the following steps of the approach.

  - *Service identification* step consists of identifying top-level and low-level services. Top-level services are not used by another service but can contain low-level services. Low-level services are underneath a top level service. Top-level services are identified by decomposing the *CIDG* into a set of connected components with a unique root such that each component is an independent subgraph of the *CIDG*. These rooted components are called modularized *CIDG (MCIDG)*. The root of each *MCIDG* is considered as a top-level service candidate and the other nodes as the low-level service candidates underneath the top-level service candidate.
  - *Component generation* consists of identifying components that realize top-level and low-level services. This is done by identifying the component elements defined by the authors, such as: the component facade and constituting set of classes.
  - *System transformation* step consists simply of applying refactoring operations to transform the object oriented system into a concrete service-based version of it.

  The approach was evaluated on a case study. The goal of this evaluation is to measure the reusability of the resulted top-level services. For that, the authors used the reusability model proposed by Washizaki et al [Washizaki 2003]. In this model, reusability is decomposed into three quality attributes: understandability, adaptability and portability. These quality attributes are hierarchically subdivided into metrics from the literature. The authors concluded that services extracted by their approach have a reasonable level of understandability, adaptability and portability.

- [Canfora 2008] proposed a wrapper-based semi-automatic approach for migrating form-based legacy systems, a class of interactive systems, to service oriented architectures. The process is decomposed into three steps: selection of the candidate services (use cases), wrapping of the selected use cases and the deployment and validation of the wrapped use cases.

  - Candidate services selection consists of determining which legacy system use cases can be exposed as services in a SOA based on state of the art approaches that are interested in resolving this decision problem.
  - Selected use cases wrapping step consists of i) identifying simple and/or composite services that correspond to use cases. ii) a reverse engineering step to generate a *finite state automaton* (FSA), screen template and the interface of the wrapped service. iii) wrapper design in order to generate the FSA description document. The FSA specifies user-system interactions. In this automaton, states correspond to the different displayed screens and transitions correspond to actions performed by the user on screens. In addition to the finite state automaton, the structure of the

UI forms is needed. This structure is specified by a model called *screen template*. The screen template is characterized by text fields and their positions in the screen. The FSA and the screen template represent the main requirements for the wrapper.

  - The deployment and validation step consists of importing and publishing the identified services.

- [Fuhr 2011] proposed a semi-automatic approach to identify services using clustering techniques. The inputs of the approach are business processes modeled as activity diagrams and the source code of the system in question. The first step consists of instrumenting the source code of the system given as input. After that, for each activity selected by the user, this latter simulates the selected activity on the instrumented legacy system to generate code execution log. Service identification is based on a clustering technique where the similarity measure is based on how often legacy classes are used together in each activity of the business processes. The authors compared the authoritative service-based architecture, provided by the original developer, and the service-based architecture resulted when applying their approach on a subject system. The comparison was done by calculating precision and recall between clusters of the two architectures. The results of this comparison showed that the approach correctly clustered a large proportion of the studied system.

- [Adjoyan 2014] proposed an automatic service identification approach for legacy to SOA migration. The service identification is based on three quality characteristics of services namely: functionality, composability and self-containment. These characteristics are refined into metrics calculated based on the structural relationships between classes/interfaces. The approach takes as input the object-oriented source code of the legacy system and produces a set of services, clusters of classes. For this purpose, the authors propose a hierarchical agglomerative clustering algorithm. This algorithm groups together the classes with the maximized value of a fitness function defined in function of the aforementioned quality characteristics. This approach was tested on two systems. The values of functionality, composability and self-containment for the identified services were reported. Moreover, the identified services were mapped to well known architecture models of the studied systems. The results showed that a high percent of the extracted services were successfully mapped in the architecture models.

- [Kerdoudi 2016] proposed a semi-automatic approach for migrating Web applications toward Web service-oriented systems. The input of this approach is the source code and an XML file that describes the navigation between interfaces of the Web application. As output, the approach produces a set of Web services. The overall process is decomposed into several steps. First, an identification of a set of operations from each element of the Web application

is done. Then, the input and output messages related to each identified operation in the Web services are identified. After that, the developer eliminates each operation that should not be published and the remaining operations are grouped in the same Web service. Finally, the dependencies between the different selected operations in the Web services are identified.

2.2.3 Migrating to Microservice Based Paradigm

Microservice-based Architectures (MSA) are originating from SOA. However, MSA can be distinct from SOA by some key characteristics, such as service granularity where microservices are relatively small with respect to services in SOA as depicted in Figure 2.2.

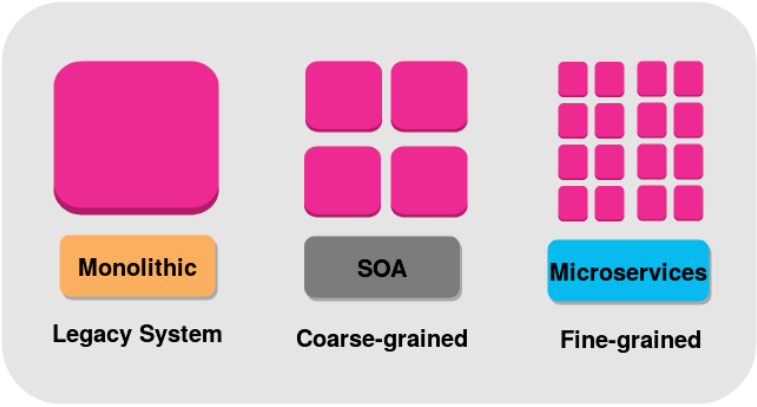

Figure 2.2: SOA and MSA granularity

However, The term "small" was not precisely defined in the works that dealt with the topic of microservices [Newman 2015] and most definitions of microservices do not provide any insight into the level of granularity required for the functionality to be branded as a microservice. There exist several factors that can define how small is small such as the number of members of the team managing the microservice and the number of functions that a microservice is designed to perform. In the following, some works that target the migration to the microservice based paradigm are presented:

- [Escobar 2016] proposed an automated approach to partition Java EE systems into microservices and visualize the resulted microservice-based architecture. The first step of the approach consists of representing the input system in a graph format where nodes represent classes/interfaces and edges represent structural relationships between these classes/interfaces. An algorithm is then applied on the graph to obtain clusters and links between them. This algorithm groups in the same cluster all classes/interfaces that participate in an invocation sequence (e.g., if there exists two nodes A and B, all classes/interfaces that exists between invocations that starts in A and finishes in B are

grouped, with A and B, in the same cluster). There may be classes/interfaces that belong to two or more clusters. The obtained clusters are linked via edges labeled with the number of elements that result from the intersection between the set of classes/interfaces of the two clusters. Once the clusters and the links between them have been identified, each cluster and the set of clusters with which this cluster is related by a link, whose label value is greater than a user-defined threshold, are grouped in a microservice. The approach was applied on a case study which basically reported measures on the number of identified microservices and the number of classes/interfaces in each identified microservice. Moreover, the authors discussed that the understanding is improved through the proposed approach because it provides visualizations of the identified microservices.

- [Gysel 2016] proposed an automatic approach for service decomposition based on a catalog which assembles 16 coupling measures. The input of the approach consists of nine user representations of the system to be decomposed into a set of microservices. The approach extracts from these representations the coupling measures and nano-entities. Each nano-entity serves as the building block of microservices, it can be a field, a method or a class. The extracted nano-entities and coupling criteria are transformed into an undirected and weighted graph where nodes represent nano-entities and the weights of edges indicate how coupled two nano-entities are. A clustering algorithm is then applied on the graph to find candidate microservices. The approach was assessed on two systems. For these systems, the authors decided to classify the identified microservices into four categories: excellent, good, acceptable or bad. This classification was based on the authors experience in microservice design. The identified microservices of the two systems were judged good and acceptable.

- Levcovitz et al [Levcovitz 2016] proposed an approach that identifies manually microservices from monolithic systems represented by a tuple *(Façade, business functions, database tables)*. Façade represents the entry points of the system that use business functions. Business functions are methods that depend on database tables. The approach supposes that the input system is structured into subsystems. The first step of the process maps each subsystem to the database on which it depends. Then, a dependency graph is created. Nodes in this graph are the monolithic system elements (Façade, business functions and database tables). Edges can be of three types: (i) calls from facades to business functions; (ii) calls between business functions; and (iii) accesses from business functions to database tables. Using the created graph, a set of pairs of the form (Façade, database table) are identified where a path from Façade to database table exists in the dependency graph. After that, for each subsystem, a set of the pairs identified in the previous step is selected where the database table parts in these pairs are the same mapped to the subsystem.

For each selected pair, a microservice candidate for the subsystem is identified. The approach was applied on a large system. For this system, discussions were made on the number of the identified microservices. No discussions were made about the benefit brought by the approach in terms of improvement in maintainability.

- [Mazlami 2017] proposed an automatic approach for extracting microservices from monolithic software systems. The input of the approach is the source code of the monolithic object oriented software system from which a graph representation is recovered. The nodes of the graph are classes and each edge has a weight defined by a weight function which determines how strong is the coupling between classes according to the used coupling strategy. Three coupling strategies are proposed: logical coupling, semantic coupling and contributor coupling. The final step of the process consists of partitioning the graph representation into connected components to obtain candidates for microservices. The approach was applied on 21 systems. For each system, the quality of the extracted microservices is measured. A microservice based solution is said of high quality if the team size across all microservices is reduced compared to the team size of the original monolithic system.

- [Selmadji 2018] proposed an automatic approach for microservice identification from object oriented applications. The identification is based on a quality function defined by an analysis of microservice characteristics namely granularity, cohesion and autonomy. This function is based on metrics that measure these characteristics. These metrics are calculated based on the structural relationships between source code artifacts. The input of the approach is the source code of the legacy system and produces a set of microservices as output. A microservice is defined by the authors as a set of classes, where each class is allocated to exactly one microservice (no shared classes between microservices). Classes are allocated to clusters using a hierarchical agglomerative clustering algorithm. This algorithm groups together the classes with the maximized value of the aforementioned fitness function. The approach was tested on three systems. The resulted microservices were compared to microservices extracted manually by the authors. The comparison showed that there is a great matching between the extracted services by the proposed approach and by those extracted manually.

2.2.4 Migrating to Aspect Oriented Paradigm

The aspect oriented programming (AOP) is a paradigm that aims to separate the crosscutting functionalities, which are of non-functional nature (authentication and encryption, for example) or technical (access to a database, for example), from the core, business, functionalities in order to improve maintainability, understandabil-

ity and modularity. These crosscutting functionalities are known by the name of *"Aspects"*. Therefore, an aspect oriented program is decomposed into two parts: i) classes that represent the core functionalities and ii) aspects that represent the crosscutting functionalities.
Aspects are not called by classes that represent the core functionalities. They are applied on the program through an *aspect weaver*. *Aspect weaving* is the operation that takes classes and aspects as input and produces a system that integrates the features of both classes and aspects. *AOP* core principle is depicted in Figure 2.3. In this figure, the main concepts of *AOP* namely *pointcut, jointpoint, advice and weaving* are presented. *Joinpoint* is a point in the program in which one or more aspects can be applied. It can be, for example, a method being called or a variable being modified. *Pointcut* defines at what joinpoints, the associated *Advice* should be applied. An *advice* corresponds to the code executed before, after or around a joinpoint.

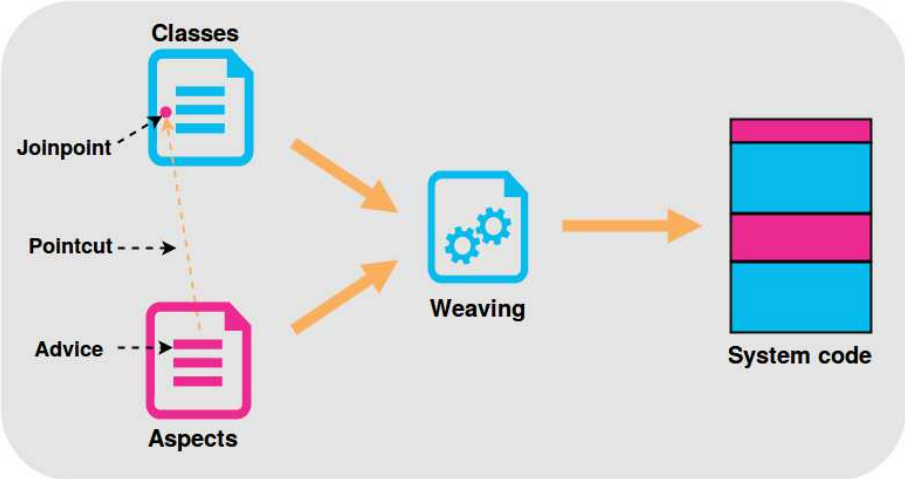

Figure 2.3: Weaving Principle in Aspect Oriented Programming (AOP)

In order to benefit from the advantages of aspect oriented programming, several approaches have been proposed in the literature to migrate existing object oriented systems to aspect oriented ones. The migration passes through two phases: aspect mining that consists of the identification of code representing the existing crosscutting concerns, and aspect extraction (including object-to-aspect refactoring). The first step has been well studied in the literature [Breu 2004, Marin 2004, Tonella 2004, Tourwé 2004, Bernardi 2016]. Some migration approaches are presented bellow:

- Binkly et al [Binkley 2006] proposed an automatic refactoring approach for migrating existing Java applications into AspectJ ones. They assume that aspects are already identified. The parts of code supposed to be refactored are method calls. For this, six refactorings were introduced according to the position of the call (before/after a method call, beginning/end of a method,

etc.). The proposed refactorings were applied on four case studies. For these case studies, the authors discussed the impact of the refactoring on performance, in terms of execution time, and size of the system. Overall, the results showed that no performance degradation was noticed for the refactored code. Moreover, the authors described that while the base code size is only slightly reduced, the base code structure is substantially simplified.

- In the same context, Ceccato et al [Ceccato 2007, Ceccato 2008] proposed an automatic approach for aspect identification. Once the candidate aspects are located, six refactoring operations can be applied to support migration (namely: Extract Beginning/End of Method/Handler, Extract Before/After Call, Extract Conditional, Pre Return, Extract Wrapper and Extract Exception Handling). In the case when none of these refactorings apply to an annotated code, additional transformations to code can be applied to make one or more of the six refactorings applicable. In order to assess if the approach is beneficial to understandability, maintainability and modularity, the authors asked software developers to perform some maintenance tasks either on the base system or on its refactored version. Details of the experiment are not given in the papers describing the approach (neither in [Ceccato 2007] nor [Ceccato 2008]). However, the authors summarize their finding by stating that the proposed approach improved modularity, understandability and maintainability.

2.3 Improving Maintainability by Remaining in the Object Oriented Paradigm

In addition to the aforementioned works that improve maintainability by modernizing the legacy system to a new paradigm, a large body of works has been proposed to improve maintainability by remaining in the same paradigm used in developing the legacy system. An interest in eliminating bad structures has been growing in the community of software engineering in order to improve maintainability. The elimination of bad structures is done by applying refactoring operations.
In this context, Fowler [Fowler 1999] defined 22 refactorings for Java programs and initially introduced the concept of bad smells in code as an indicator when (and where) to apply refactorings. Many development environments and plug-ins such as Eclipse[2], IntelliJ IDEA[3], JDeodorant[4] and RefactorIT[5] provide automated support of several Fowler's refactorings.

[2]https://www.eclipse.org/
[3]http://www.jetbrains.com/idea/
[4]http://www.jdeodorant.com
[5]http://sourceforge.net/projects/refactorit/

Tourwé et al [Tourwé 2003] proposed a semi-automatic approach that identifies automatically two types of bad smells, obsolete parameter and inappropriate interface, using logic meta programming. Once instances of these bad smells have been identified, the user can choose refactoring operations from the five proposed *(remove parameter, add class, add method, rename variable and pull up variable)* and apply it manually to eliminate the bad smell.
For decoupling classes using interfaces, Steimann et al [Steimann 2006] proposed a fully automated refactoring approach for the introduction of new interfaces. This refactoring calculates from variable declarations, the minimal types (interfaces), containing all the method declarations needed from the chosen reference and all other references it gets possibly assigned to.
Shah et al [Shah 2013] proposed an algorithm that uses various refactoring techniques to automatically remove unwanted dependencies in Java programs. This algorithm is designed to eliminate maintainability defects represented by four types of anti-patterns: circular dependencies between packages, subtypes knowledge (when a subtype is used either directly or indirectly by its super type), abstraction without decoupling and degenerated inheritance. They classified dependencies between classes in four categories and for each category they specified a refactoring operation.
Ouni et al [Ouni 2013] proposed an approach for automatically detecting and correcting several maintainability defects in source code. Authors focused on three types of maintainability defects, namely *Blob, Spaghetti Code* and *Functional Decomposition*. Quality metrics are used in order to generate rules for the detection of each type of maintainability defect. An example of a rule is: a class having more than 10 attributes and 20 methods is considered as a blob. In this rule, the number of attributes and the number of methods of a class correspond to two quality metrics that are used to detect a blob defect. In the correction step, Fowler's catalog of refactoring is used to recommend the suitable refactoring for each maintainability defect.

2.4 Improving Understandability and Modularity by Architecture Recovery

As described earlier, migration processes into new paradigms, whether to components, services or micorservices, passes through a step of architecture recovery. However, many other works do not propose architecture recovery approaches for the migration purpose but for the understanding purpose or for restructuring the system by following an obtained modular architecture.
The most significant works of this category are presented below. We present first approaches that target the recovery of the implementation-level architecture and after that the approaches whose goal is recovering the design-level architecture. Design-level architectures describe the different high-level parts of a legacy system. Implementation-level architectures describe the program artifacts.

2.4.1 Implementation-level architecture recovery

Several approaches and tools (e.g., Understand[6], Structure101[7], Lattix [8]) have been developed to recover class-based models of object oriented software systems. In the following, works that recover object-based models are discussed. These works rely either on static, dynamic or hybrid analysis. Static analysis refers to source code examination without executing the legacy system. Dynamic analysis consists of observing the system during its execution. While static information presents a complete picture of what could happen at runtime, it does not show what actually happens. On the other hand, dynamic information is precise but it is challenged by the coverage problem. The combination of static and dynamic analyses is known as hybrid analysis.

- Both Spiegel et al [Spiegel 2002] and Abi-Antoun et al [Abi-Antoun 2009] proposed static analysis techniques, named Pangaea and SCHOLIA respectively, in order to recover object graphs of Java systems. Nodes of the graphs recovered in the two works represent the objects that exist at runtime whereas edges between two nodes can be of three kinds: creation, reference and usage according to Spiegel et al [Spiegel 2002], and represent references acquired by a field according to Abi-Antoun et al [Abi-Antoun 2009]. In order to mitigate the complexity of the recovered model, in SCHOLIA, architectural extractors (developers) use ownership domain annotations to annotate the Java code manually, then they use static analysis to extract a hierarchical Ownership Object Graph (OOG). This aspect of mitigating the complexity of the recovered graphs is not taken into account in Pangaea

- Each of the works of de Brito et al [de Brito 2013], Flangan et al [Flanagan 2006] and Briand et al [Briand 2006] recovers object graphs dynamically. Nodes of these graphs represent objects. The edges between nodes have different meanings in each approach. According to de Brito et al [de Brito 2013], an edge between two objects (o1 and o2) indicates that o1 has obtained a reference to o2 at some point in its lifespan. This reference could be acquired by an object's field, a local variable or by a method's formal parameter. In Flangan et al [Flanagan 2006], an edge between two nodes o1 and o2 means that a field of o1 points to o2. Since the approach of Briand et al [Briand 2006] recovers a special type of object graphs which is a scenario diagram (a simplified version of a sequence diagram that depicts a specific scenario), edges between nodes are method invocations. To promote the scalability of OGs, while de Brito

[6]https://scitools.com/
[7]http://structure101.com/
[8]http://lattix.com/lattix-architect

et al [de Brito 2013] use the summarization by domain, a group of nodes explicitly defined by developers, and Flangan et al [Flanagan 2006] apply some abstractions such as: defining ownership and containement relations between objects, this aspect is not discussed in the work of Briand et al [Briand 2006].

- In order to recover object graphs, Wang et al [Wang 2008] and Labiche et al [Labiche 2013] proposed reverse engineering techniques based on a hybrid analysis. In the work of Wang et al [Wang 2008], static analysis is used to build the object graph (nodes represent objects and each edge represents a specific relation between two objects: creation, invocation, read or write). This graph is then enhanced with dynamic profiling information such as allocation frequency on nodes and interaction count on edges. Thereafter, this information is used to reduce the object graph to a tractable size. Labiche et al [Labiche 2013] presented a technique that also combines both static and dynamic analyses in order to recover scenario diagrams. In this technique, instead of instrumenting the control-flow structures, as in their previous work [Briand 2006], the static analysis is used to reverse engineer control flow graphs. Like in their previous work, Labiche et al [Labiche 2013] do not discuss the aspect of mitigating the complexity of recovered graphs.

The visualization of legacy systems source code artifacts has been widely addressed in the literature. In particular, attention was drew to the visualization of classes/interfaces characteristics such as: the number of lines of code, number of methods, number of fields, etc. Some of these visualization works are presented bellow:

- Langelier et al [Langelier 2005] proposed a semi-automatic approach for large-scale software visualization in order to understand software properties. In this visualization, a class is represented by a 3D box. Some class characteristics are captured throughout well known metrics. Box color, twist, and size are matched to those characteristics. The authors defined two class layouts: *Treemap* and *Sunburst* in order to allow a separation of the classes into areas. In addition, some filters were implemented in order to focus on useful elements and reduce the visual importance of useless elements. As a filter example, for a given class, the expert can view only classes that are related to it by a particular type of link (association, aggregation, etc.).

- Wettel et al [Wettel 2011] developed the CodeCity tool that uses the city metaphor to visualize software systems as cities. In these cities, buildings represent classes and districts represent packages. The number of methods, attributes and lines of code are mapped to height, base size and colors of the buildings respectively. This visualization allowed the assessment of the design quality by detecting some anti-patterns as the *God Class* one. Balogh et al [Balogh 2013, Balogh 2015] developed the CodeMetropolis tool to visualize software systems elements. Their visualization is also based on the city metaphor.

- Cornelissen et al [Cornelissen 2009b] developed EXTRAVIS tool for executing trace visualization. EXTRAVIS provides two views: the *massive sequence view* which represents a UML sequence diagram and the *circular bundled view* which displays the system artifacts on a circle and shows their relations in a bundled way.

- Fittkau et al [Fittkau 2017] developed the ExplorViz 3-D trace visualization tool. ExplorViz provides two types of visualization: the Landscape-level visualization and the Application-level visualization. In the Application-level visualization, packages are displayed as green boxes, classes as purple boxes and links between classes as orange lines. In the Landscape-level visualization, a landscape, displayed as a gray box, is a group of systems which represents a logical union of multiple applications, purple boxes, and servers(node groups), green boxes. The communication between applications is visualized by orange lines.

2.4.2 Design-level architectures recovery

Mitchell et al [Mitchell 2006] proposed the Bunch clustering tool which uses hill-climbing and genetic algorithms to group classes/interfaces into clusters. The input of the tool is a class-based graph in which nodes represent classes/interfaces and edges represent dependencies between these classes/interfaces. These classes/interfaces are clustered based on a modularization quality function. This function measures the quality of the input graph clustering solutions quantitatively as the trade-off between inter-connectivity (i.e., dependencies between the classes/interfaces of different clusters) and intra-connectivity (i.e., dependencies between the classes/interfaces of the same cluster).
Tzerpos and Holt [Tzerpos 2000] proposed the ACDC (Algorithm for Comprehension-Driven Clustering) clustering algorithm. This algorithm is pattern-driven. In a first stage, the algorithm clusters a large proportion of the system artifacts based on patterns that are commonly observed in decompositions of large software systems. These patterns allow grouping in the same cluster: 1) procedures/functions, as well as variable declarations that reside in the same source file 2) source files that exist in the same directory, 3) source files that are leaves in a system's graph, 4) source files that are accessed by the majority of clusters, 5) source files that depend on a large number of other resources and 6) source files that belong to a subgraph obtained through dominance analysis. The second stage concerns source files that are still not assigned to a cluster, since they did not fit any of the used patterns. For that, the technique of orphan adoption [Tzerpos 1997] is used. This technique attempts to place each orphan source file in the cluster that seems more appropriate.

ARC (Architectural Recovery using Concerns) is a clustering algorithm developed in [Garcia 2011]. This algorithm uses the identifiers and comments in source code to detect the concerns the system addresses. These concerns, combined with structural informations, are used in order to group the system artifacts in clusters based on similarity measures. In this algorithm, similarity measures between concerns are computed using a statistical language model called Latent Dirichlet Allocation (LDA) [Blei 2003].

2.5 Discussion

Based on works presented in Sections 2.2 and 2.3, maintainability can take many forms in terms of first class programming entities such as: components, services, microservices, aspects and well structured classes. For several approaches ([Jain 2001, Lee 2003, Kim 2004, Washizaki 2005, Constantinou 2015, Li 2006, Fuhr 2011, Escobar 2016, Mazlami 2017]) that aim to improve maintainability by migrating to the component, service, or microservice-based paradigm by considering that the unit of maintainability is a cluster of classes/interfaces, and that classes/interfaces can be shared between several units/clusters. Therefore, if we build a new system with multiple components obtained with these approaches, then we may have duplicated code. This will complicate maintainability tasks especially in the case when bugs exist in the duplicated code [Chatterji 2013] (DRY principle in not respected).
Moreover, in the works where no classes/interfaces are duplicated in several clusters [Alshara 2015, Adjoyan 2014, Selmadji 2018, Allier 2011], if a user wants to develop a new system using an independent class or subset of classes in a cluster, it is required to use the entire cluster. This implies that the new developed system contains unnecessary code. The maintenance of this latter is a waste of development time and resources [Eder 2012].
In order to tackle the problems of duplicated and unused code, we argue that refactoring classes individually, resembling thus to component descriptors, makes them more maintainable.
The maintainability covered by the works that target the migration to aspect oriented programming [Binkley 2006, Ceccato 2007, Ceccato 2008] does not involve the same program elements. Indeed, in these works, the refactoring aims to separate the crosscutting functionalities, which are of a non-functional or technical nature (authentication or access to a database, for example), from the core functionalities. We argue that isolating parts of the code that represents architecture description (instantiations, connections, provided/required interfaces declarations...) form parts of code that represents the core functionalities makes the OO systems more maintainable. The two categories of work are perfectly complementary.

Moreover, approaches that propose refactoring operations by remaining in the same paradigm [Fowler 1999, Tourwé 2003, Steimann 2006, Shah 2013, Ouni 2013] have the same goal, which is improving maintainability. In the case of refactoring classes individually, we argue that the same goal is targeted likewise but with a one other requirement which is having at the end of the process a class conform to a component descriptor.

As discussed in Section 2.4.1, several approaches have been proposed for the recovery of traditional design representations such as class-based graphs. Since software architectures are increasingly dynamic, with components being assembled at runtime, recovering the runtime architecture is becoming more significant. The runtime architecture of a given system represents a model of the system's concrete running entities (objects) and dependencies between them. We believe that a runtime architecture, an object-based model, communicates different information from a class based model. This is because an object-based model shows only the set of objects that really exist at runtime and their relationships. Moreover, a class-based model shows in a generic way all associations to which a class can participate. However, it is when this class will be instantiated, it will be clear which instances really participate in these associations. We believe this has a great impact on comprehension.

The importance of having the runtime architecture to handle comprehension tasks was stressed in several experimental studies such as the ones from [Lee 2008] and [Ammar 2012]. [Lee 2008] conducted an interview-based experimental study to investigate which kind of information is desired by maintainers/developers during comprehension tasks. 19 maintainers were involved in this study. Each maintainer was provided with source code and a simple tool that models the relationships of the code elements. The intent of this tool is to get information about what maintainers think that a diagram should contain. The participants described many kinds of information that should be in a diagram including objects interactions. Many participants stressed the importance of having a diagram that models objects and their interactions. One of these participants described "draw how objects connect to each other at runtime when I want to understand code that is unknown; an object diagram is more interesting than a class diagram, as it expresses more how it functions". Moreover, [Ammar 2012] evaluated whether a runtime architecture is more helpful in handling comprehension tasks than class diagram. The results of their study confirmed their research hypothesis which was: for comprehension tasks that require knowledge about the runtime structure, maintainers that use a runtime architecture requires less effort and spend less time than developers who use only class diagrams.

Most of the automatic and semi-automatic approaches for runtime architecture recovery build flat models (e.g. [Spiegel 2002]). These flat models can have a small size in case of small-sized software systems. However, in the case of medium and large sized systems, these models become unreadable, with thousands to millions of modeling elements.

Other approaches that take into account this aspect of complexity management (such as [Wang 2008, Abi-Antoun 2009, de Brito 2013, Flanagan 2006] group sets of objects in several summarizing objects and/or add labels that corresponds to properties, on runtime architecture nodes and/or edges. Thereafter, these labels are used in order to reduce the runtime architecture model size by definitely discarding, from the final graph, nodes and/or edges that do not fit to some criteria. This can be beneficial depending on the goal of the approach, however, in some situations of comprehension, it would be more interesting to have an interactive way that allows to hide and display nodes and edges as needed, having thus a visualization steered by the user.
It is argued that the lifespan of running objects in a system is an important information that is often required to reason about the performance of a software system [Peiris 2016]. Moreover, the probability of existence of objects at runtime could be used to eliminate *dead code*, which involves objects which never really exist at runtime, during comprehension tasks since the comprehension of *dead code* is a waste of development time and resources.
Despite the importance of these types of information, to the best of our knowledge, an approach that exploit them in order to manage the complexity of the recovered runtime architecture has never been approached in the literature.

Approaches that target the recovery of the design-level architecture for the remodularization [Mitchell 2006, Tzerpos 2000, Garcia 2011] and/or migration[9] purposes [Jain 2001, Lee 2003, Washizaki 2005, Constantinou 2015, Allier 2011, Alshara 2015, Adjoyan 2014, Escobar 2016, Gysel 2016, Selmadji 2018] consider, in general, structural relationships between source code artifacts for measuring coupling and cohesion of the different decompositions of the legacy system. These approaches do not make a difference between the different types of structural relationships (e.g., field typing, method invocation, etc) that can exist between classes/interfaces. These structural relationships are considered equivalent which is not always true and not accurate enough. We believe that the different types of relationships should be differentiated. Moreover, we believe that a composite class/interface and its components should logically be clustered in the same module and their separation across different modules results in a high coupling between these modules. To the best of our knowledge, only the approaches of Lee et al [Lee 2003], Kim et al [Kim 2004] and Li et al [Li 2006] consider composition relationships represented in a *UML* class diagram. In Kim et al's [Kim 2004] approach, the class diagram is supposed to be available beforehand which is not always obvious. The approach of Lee et al [Lee 2003] is composition and inheritance conservative. That is, classes/interfaces with composition and inheritance relationships are always grouped in the same module. However, these relationships have been identified manually and no details were given on how they

[9]The ultimate goal of these approaches is the migration into a new paradigm, however, they produce intermediate design-level architectures which are beneficial for understanding and remodularization.

was identified in source code[10]. For the approach of Li et al [Li 2006], no details were given on how the composition relationships were identified automatically in source code. Moreover, these relationships are not exploited in the following steps of their recovery process. Thus, a composition conservative remodularization approach that differentiates between types of structural relationships is needed.

The following part of the thesis aims at presenting our contributions answering all the shortcomings mentioned above.

[10]We speak in particular of the identification of the composition relationships. Inheritance relationships identification is straightforward.

Chapter 3

Migrating Object-Oriented Software Systems into Component-based equivalent ones

Contents

3.1 Introduction and Problem Statement

Maintainability in Object-Oriented (OO) systems has been a major concern since the early years of OO programming languages. Component-Based (CB) paradigm has been recognized as an approach that emphasizes software maintainability. However, many existing (especially, business) software systems are built using the Object-Oriented (OO) development paradigm. Many of these systems have complex and

numerous dependencies which make them hard to modify and maintain. Therefore, it would be interesting to migrate OO systems into CB ones. This migration enables to benefit from CB development paradigm characteristics. Existing works that propose migration solutions consider that the component is a group of classes, called a cluster. To the best of our knowledge, no solution proposes refactoring classes individually to make them component descriptors although doing so helps reducing maintainability effort.
This chapter focuses on the first contribution of this thesis. We propose a migration solution that considers each class in an OO application as a component descriptor in a target CB application. The proposed process is decomposed into two steps. The aim of the first step is to detect modifiability/maintainability bad smells, i.e. decoupling and unanticipated instantiation violation (presented in Section 3.2.1). The second step allows the elimination of these defects by automatically applying a composition of code refactoring operations (Section 3.2.2). Section 3.3 discusses the results of an experimentation of this solution conducted on a set of open source Java projects. Section 3.3.4 discusses the threats to validity of the approach and Section 3.4 concludes the chapter and presents future works.

3.2 Foundations of the Proposed Approach

3.2.1 Decoupling and Non Anticipated Instantiations Violation

While code decoupling and non anticipated instantiations principles are fundamental [Fabresse 2008], they are not necessarily always respected in existing OO systems. Therefore, it is necessary to identify the symptoms of their violation in an existing OO code to enable their detection and elimination.

3.2.1.1 Decoupling Violation

In CB programming, code decoupling means that components are assumed to communicate only through their interfaces/ports. Therefore, a component has not a direct access to a component with which it interacts. To have this decoupling in OO applications, assuming that each class will correspond to a component descriptor, each class must, for example, expose all its public methods in abstract types (provided interfaces). Then, other classes that use these methods should declare their dependence on these abstract types, which become their required interfaces.

However, most existing OO systems have multiple dependencies between their different classes (direct concrete types) for a cooperative business processing. In particular, it is possible for a field or a parameter to be typed with a concrete class of the application. These situations lead to code decoupling violation. To deal with decoupling violation, we consider the two symptoms of the modifiability/maintainability defect: *"Absence or Incompleteness of Provided Interfaces(AIPI)" and "Absence of Required Interfaces (ARI)"*.
AIPI symptoms are identified when:

1. a class defines a public non-static method not declared in the interfaces implemented by this class (or no interface is implemented by this class),
2. a class declares public fields or fields with no explicit visibility modifier, and
3. a class declares global constants.

Since it is recognized that program variables should be typed with abstract types rather than concrete classes [Steimann 2006, Gamma 1995, Fowler 1999], *ARI* symptoms are identified when a class declares fields with a concrete class type.

3.2.1.2 Non Anticipated Instantiations Violation

In CB systems, non anticipated instantiations principle means that a component requiring a service can be connected to any other component providing such a service. That is, the implementation of a component should not include a connection to another particular component. This connection should be established only by a third party, who is the developer of the system/component that uses the two components to be connected. To comply with this connection fashion in OO systems, constructor calls should not be used. Instead, declarative annotations should be defined; these are processed by a (dependency injection) mechanism that manages instances at runtime.
To deal with non anticipated instantiations violation, we consider the symptoms of the modifiability/maintainability defect of type *EAI "Existence of Anticipated Instantiations"*. *EAI* symptoms are identified when a reference to a created object is stored in a field/local variable, is a returned value of a method, or is an argument of a method invocation. In the present work, we consider that these instantiations are not surrounded by a control flow statement.

3.2.2 Refactoring Operations

In this section, the strategy used to work out how to eliminate the modifiability defects from source code is discussed. Table 3.1 gives a summary view of the symptoms detected in the source code and the technique used for the removal of each symptom.

[1]Protected visibility is out of the scope of our approach

Table 3.1: Refactoring Operations

Symptom	Operation
Public fields or fields with no explicit visibility modifier[1] (package visibility)	Change visibilities
Global constants	Move declarations
Public non-static methods not exposed in interfaces	Expose methods
Fields typed with concrete classes	Create required interfaces
Anticipated instantiations	Use dependency injection

3.2.2.1 The "Change Visibility" Refactoring

This operation considers the *AIPI* symptom when a class field is public or has no explicit visibility modifier, i.e., has the package default visibility for Java, for example. In this case, the field visibility is simply changed to private, and a pair of setter/getter methods is inserted to access this field (only a getter method in the case of a public final field). The resulted methods will be exposed via interfaces as explained through the next refactoring operation.

3.2.2.2 The "Expose Methods" Refactoring

This type of refactoring deals with the *AIPI* symptom when a class A defines a public non-static method m and its declaration does not exist in any interface from those implemented by A. The idea here is to add this declaration to an interface I_1. This (changed) interface should not be implemented by any other class. Otherwise, i.e., when all the interfaces implemented by A are also implemented by other classes, a new interface I_2 is created and m's signature is added to it.
Another case which is taken into account is when the class A implements two interfaces or more, and all these interfaces are not implemented by other classes. In this case, it is necessary to distribute the methods on interfaces in a manner that each signature added to an interface should be cohesive with other already existing methods in this interface. To do this, the *Chidamber and Kemerer* [Chidamber 1994] *LCOM (Lack of Cohesion of Methods)* metric is calculated in order to evaluate the cohesion of each signature added to an interface and the other existing methods in this interface.
LCOM is selected because it is widely used and it has been validated by several approaches, such as [Basili 1996]. The theoretical basis of *LCOM* uses the notion of degree of similarity of methods. This degree of similarity between n methods can be defined to be the intersection of the sets (F_1,...,F_n) of a class fields that are used by the methods (M_1 ,...,M_n). *LCOM* is the number of empty intersections P minus the number of non empty intersections Q if $P > Q$ or 0 otherwise. In our case, the *LCOM* metric is calculated between all methods of a given interface, among

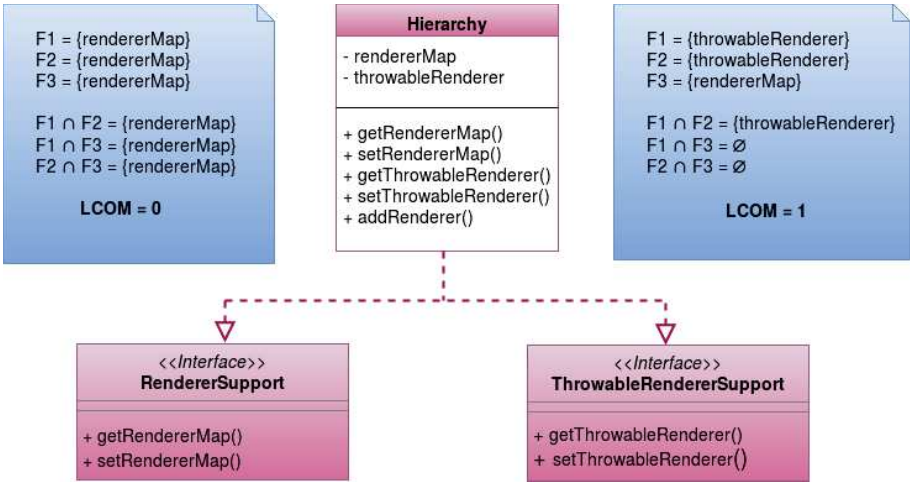

Figure 3.1: Using LCOM metric to apportion methods on interfaces

the interfaces that are not implemented by any other class, plus the method that must be exposed in an interface. In this way, methods that use the same fields will be exposed in the same interfaces. To better understand this idea, an illustrative example, extracted from the Log4j[2] open source project, is given in Figure 3.1.
In this example, we have a class *Hierarchy* that declares two fields (*throwableRenderer and rendererMap*) and five public methods (*getThrowableRenderer(), setThrowableRenderer(), getRendererMap(), setRendererMap() and addRenderer()*). This class implements two interfaces *RendererSupport* and *ThrowableRendererSupport*. The methods *getThrowableRenderer()* and *setThrowableRenderer()* are exposed in the interface *ThrowableRendererSupport* and the methods *getRendererMap()* and *setRendererMap()* are exposed in the interface *RendrerSupport*. The method *addRenderer()* is not exposed in the interfaces implemented by the class *Hierarchy*. The two interfaces are not implemented by other classes so the declaration of *addRenderer()* can be added to one of them. To decide to which interface it should be added, the LCOM metric is calculated to measure the cohesion between *RendererSupport* methods with *addRenderer()* and between *ThrowableRendererSupport* methods with *addRenderer()*. In Figure 3.1, F sets in the blue boxes correspond to the fields used by each method. For example, if we take the blue box on the right, F_1 set contains the field *throwableRenderer* which is the field used by the method *getThrowableRenderer()*, F_2 set contains the field *throwableRenderer* which is the field used by the method *setThrowableRenderer()* and F_3 set contains the field *rendererMap* which is the field used by the method *addRenderer()*.
According to LCOM values, we can deduce that the *addRenedrer()* method is more cohesive with the methods exposed in *RendererSupport* ($0 < 1$), therefore, the declaration of the addRenderer() method will be added to the interface *RendererSupport*.

[2]https://logging.apache.org/log4j/1.2/download.html

Someone can find that the use of *Default Methods* in Java 8 can be useful to eliminate this type of modifiability/maintainability defect. But the idea here consists in exposing only the declarations of methods not their implementations.

3.2.2.3 The "Move Declaration" Refactoring

The use of global constants is very common and even essential. The proper way to define a constant in Java is to define a public static final field. The fact that it is public and static allows access from anywhere and the modifier final prohibits its modification, which is generally sought for a constant. To deal with a global constant declaration (*AIPI* symptom type), we thought the movement of the field declaration from its class to one of the interfaces implemented by this class.

3.2.2.4 The "Create Required Interface" Refactoring

This refactoring is used when a field is typed with a concrete class A. It consists of the following steps: search all invocations to external methods whose receiver is saved in the considered field, collect the signatures of these methods, create a new interface (considered as the required interface), add the signatures of the invoked methods on the field[3] to this interface and replace the type of the field by the newly created interface. The last step consists of adding inheritance links between the required interface and the provided interfaces implemented by A (the provided interface extends the required interface). The class' required interfaces will be as many as the number of concrete classes used as types for its fields. However a single required interface is created for two fields with the same type.
A case that must be taken into consideration is when the field typed with a concrete class is assigned to other references directly or indirectly (e.g, local variable). In this case, the program can be type incorrect when additional methods, methods that were not invoked on the field, are invoked on these references. For this reason, further changes to the program may therefore be necessary. Changes include the possible change of the declared type of these references.
To better understand this refactoring, suppose that there are two declarations A a and B b and the assignment $b = a$. A and B can refer to the same type or A *extends/implements* B. Suppose also that RI is the interface that contains the methods needed from the reference a. Methods needed by b and that do not exist in RI are added to RI and the declaration B b is changed to become RI b.

By applying this type of refactoring and the former ones, the required and provided interfaces are henceforth defined explicitly in the source code.

[3] In this way, the interface segregation principle is respected.

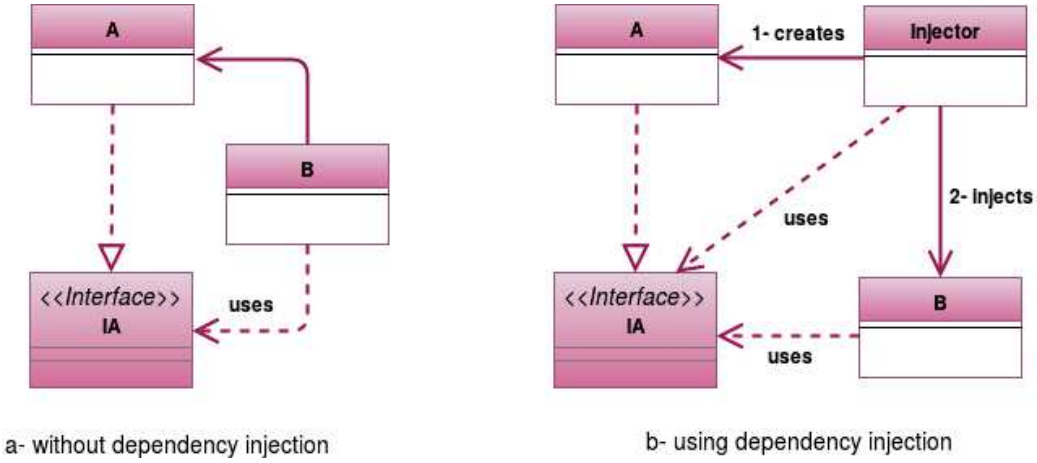

Figure 3.2: Dependency Injection Mechanism

3.2.2.5 The "Use Dependency Injection" Refactoring

Dependency injection (DI) is a powerful technique for decoupling classes. In this technique, the client class does not depend on a specific implementation class. The implementation class is injected at runtime by a container. To better understand the technique, suppose we have a class B that uses an object of the class *A* (Figure 3.2). This class *A* is an implementation of an interface *IA*. Dependency injection removes the dependence of class *B* on class *A* by adding an injector (container) and making it responsible of the dependency look up. This injector is often externally configured by an XML file. The advantage of using *DI* technique is: allowing class *B* to work with any implementation of the *IA* interface without any changes in source code, only the XML file will be changed in the case of XML-based configuration, so there is no recompilation of the source code.
Many dependency injection containers can be used to inject dependencies at run-time such as Spring DI[4], PicoContainer[5], Google Guice[6] and Dagger (1 and 2)[7]. They provide almost exactly the same functionality. Each needs slightly different configuration.
In the following, the general structures that indicate the use of dependency injection are given. All these structures will be illustrated by examples extracted from the Jasml[8] and FreeCS[9] open source projects.

1. ***Instances stored in fields***

[4]https://spring.io/
[5]http://picocontainer.com/
[6]https://github.com/google/guice
[7]http://google.github.io/dagger/
[8]http://jasml.sourceforge.net/
[9]http://freecs.sourceforge.net/

An instantiation, constructor call, statement in a method/constructor where this statement's left-hand-side corresponds to a field. The called constructor does not take method/constructor parameters as arguments or it takes attainable ones (arguments whose values can be calculated by a static analysis). In this case, the refactoring is typically done through the following steps:

(a) save the arguments of the constructor call if any,

(b) replace the instantiation statement by an annotation used by the used DI framework. For example, the *@Autowired* annotation is used on fields in Spring DI (the field must be non final).

An example of this case is illustrated in listings 3.1. The annotation *@Autowired* enables the automatic dependency injection based on the type.

Listing 3.1: Instances stored in fields

```
// Before dependency injection use
public class SourceCodeBuilder{

   private SourceCodeBuilderConfig config;

   public SourceCodeBuilder(){
       config = new SourceCodeBuilderConfig();
       ...
   }
}

// After dependency injection use
public class SourceCodeBuilder{

   @Autowired
   private SourceCodeBuilderConfig config;

   public SourceCodeBuilder(){
       ...
   }
}

// Architecture description in Spring
<bean id="sourceCodeBuilder"
class="SourceCodeBuilder">
</bean>
<bean id="sourceCodeBuilderConfig"
class="SourceCodeBuilderConfig">
</bean>
```

2. ***Instances stored in local variables***

The second case is when an instantiation statement is made inside a method-/constructor and the obtained reference is stored in a local variable. As in the previous case, we suppose that the constructor call does not take any argument or take attainable ones. This local variable will be removed from the method-/constructor body and turned into a private field of the class(this refactoring, transforming a local variable to a field, is failure-safe as it has been experimented in the literature [Gligoric 2013]). This field will be treated following

the previous case. Renaming this local variable, before moving it, could be another additional refactoring. In contrast to the previous case, since what is transformed is a local variable and not a field, we use here a lazy initialized DI so that the created field is injected when it is first requested (during the execution of the method/constructor where it was originally declared as a local variable), rather than at startup.

An example of this case is given in Listing 3.2. The attribute *lazy-init* of a bean is added to allow creating a bean instance when it is first requested, rather than at startup since *info* was initially a local variable and it has been turned into a field.

Listing 3.2: Instances stored in local variables

```
// Before dependency injection use
public class OpcodeLoader{
        ...
    public void processOpcode(Node node){
        OpcodeInfo info = new OpcodeInfo();
        ...
    }
}

// After dependency injection use
public class OpcodeLoader{
        ...
    @Autowired
    private OpcodeInfo info;

    public void processOpcode(Node node){
        ...
    }
}

// Architecture description in Spring
<bean id="opcodeLoader" class="OpcodeLoader">
</bean>
<bean id="opcodeInfo" class="OpcodeInfo"
lazy-init="true">
</bean>
```

To the best of our knowledge, *PicoContainer*, *Dagger* and *Guice* do not propose a technique to inject dependencies in local variables. Conversely, Spring propose the lookup method injection that can be used in such case. In few words, it consists of a dynamic subclassing and the ability of the container to override methods on container managed beans. The use of this type of injection is explained in Listing 3.3.

Listing 3.3: Instances stored in local variables with Spring's lookup-method

```
// Before dependency injection use
public class OpcodeLoader{
        ...
    public void processOpcode(Node node){
        OpcodeInfo info = new OpcodeInfo();
        ...
    }
}
```

```
// After dependency injection use
public class OpcodeLoader{
        ...
    public void processOpcode(Node node){
        OpcodeInfo info = createOpcodeInfo();
    }

    public OpcodeInfo createOpcodeInfo(){
        return null;
    }
}

// Architecture description in Spring
<bean id="opcodeInfo" class="OpcodeInfo"
lazy-init="true"></bean>
<bean id="opcodeLoader" class="OpcodeLoader">
<lookup-method name="createOpCodeInfo"
bean="opCodeInfo"/></bean>
```

In this case, the *Spring* container will create a subclass of *OpcodeLoader* class and override the *createOpcodeInfo()* method to return an instance of *OpcodeInfo* class, based on the architecture description.

Someone can ask if *Spring* already offers a way to do this, what is the interest to use another way? We propose our new technique for two reasons. The first is avoiding to be dependent on a particular DI framework. The second reason is about the key limitation of method lookup if we have a static method. To better understand the problem, suppose that *processOpcode()* is a static method. In this case, *createOpcodeInfo()* must be static so it can not be overridden.

In fact, even if we use our technique when the method is static, the local variable info must be turned into a private static field and static fields can not be autowired, since beans can have a singleton scope. To solve this problem, we propose to add a non static setter for the field and use setter injection. This case is explained in Listing 3.4.

Listing 3.4: Instances stored in local variables inside static methods

```
// Before dependency injection use
public class OpcodeLoader{
        ...
    public static void processOpcode(Node node){
        OpcodeInfo info = new OpcodeInfo();
        ...
    }
}

// After dependency injection use
public class OpcodeLoader{
        ...
    private static OpcodeInfo info;

    @Autowired
    public void setInfo(OpcodeInfo info){
        this.info = info;
    }
```

```
20      public static void processOpcode(Node node){
            ...
22      }
}
24
// Architecture description in Spring
26  <bean id="opcodeLoader" class="OpcodeLoader">
    </bean>
28  <bean id="opcodeInfo" class="OpcodeInfo"
    lazy-init="true">
30  </bean>
```

3. ***Instances stored in fields/local variables; these instances take constructor/method parameters as arguments***

   The last case is where instances' references are stored in fields/local-variables while using non-literal values as arguments in their instantiation. To deal with this case, first, a new "default" constructor is created in the instantiated class, and the initial constructor call, in the instantiation, is replaced by this new constructor call. Then, a new method that contains exactly what the initial constructor contains is added to the instantiated class. Finally, the instantiation statement is treated following one of the two previous cases, and an invocation statement of the new method is added to the instantiating class. Anonymous object instantiations, i.e., instantiations which play the role of arguments in method invocations or returned values, for instance, are considered the same as instantiations made as right-hand-side expressions of assignments to local variables. They are processed following the same procedure than the two previous cases.

3.3 Experimental Results and Evaluation

We have implemented a prototype of the described method using Spoon [Pawlak 2015b] which is a library for source code analysis and transformation. We conducted some experiments to evaluate the truthfulness of the stated hypothesis of migrating OO systems into CB ones in order to improve their modifiability and thus maintainability. These experiments were conducted to answer the following research questions:

- **RQ1**: What is the efficiency (precision) of the detection phase?
- **RQ2**: To what extent does the proposed approach improve software modifiability and thus maintainability?

3.3.1 Data Collection

For our study, four open source Java projects were used. In order to gather these projects, *Qualitas Corpus* [10] which is a large curated collection of open source Java projects was used. Table 3.2 provides a brief description of these projects. They are of different sizes, varying from 5732 to 23012 LoC, 50 to 214 concrete classes and 1 to 36 abstract types, and developed by different teams to avoid the influence of characteristics related to team habits on results.

Table 3.2: Data collection

System	Description	LOC	#Classes	#Interfaces + Abstract classes
Jasml	Java classes visualization tool	5732	50	1 + 0
CoCoME	Commercial application	5779	99	21 + 0
FreeCS	Chat server	23012	139	17 + 6
Log4j	A Logging Framework	20129	214	20 + 16

3.3.2 Used Measures

The first research question deals with measuring the efficiency of the detection algorithms. To answer this question, we measured precision, well-known metric in the information retrieval domain. Precision assesses the ratio of true modifiability/-maintainability defects identified among the detected ones (true positives + false positives). To do this, we analyzed the four projects manually to consider identified defects as true positives:

$$precision = \frac{True\ \ Positives}{True\ \ Positives + False\ \ Positives} \tag{3.1}$$

To answer the second research question, we use the *Maintainability Index (MI)* metric that measures the maintainability of a software system [Welker 2001], and which was considered in many recent works, such as [Börstler 2016, Koteska 2018]. It allows to determine how easy it will be to modify and maintain a system. High MI values indicate that the system is easier to maintain for future changes. There are different versions in calculating MI. These are presented below:

$$MI1 = 171 - 5.2ln(V) - 0.23 * C - 16.2ln(LOC) + (50 * sin(sqrt(2.46 * NOLComments)) \tag{3.2}$$

[10] http://qualitascorpus.com/

V is the Halstead's volume which is calculated based on the number of operands and operators in methods, more details on this metric can be found in [Al Qutaish 2005]; C is the cyclomatic complexity value; LOC is the number of lines of code and *NOLComments* is the number of lines of comments. In the case of systems which do not have considerable comments, the above formula can be simplified to omit the involvement of *NOLComments* as bellow:

$$MI2 = 171 - 5.2ln(V) - 0.23 * C - 16.2ln(LOC) \tag{3.3}$$

For our study, the tool used to calculate the MI value is *JHawk*[11] which calculates a wide number of Java code metrics. It provides the two previous versions of the Maintainability Index.

3.3.3 Results

We asked four master and one PhD student, who were not involved in this work before, to analyze the systems source code manually. We gave the *Jasml* and *CoCoME* systems to two master students, we asked the two other master students to divide the *FreeCS* system and each of them analyzes half of the packages, and the PhD student was assigned to analyze the *Log4j* system.
The five students used as reference a detailed description of the modifiability/maintainability defects we wrote. All the students visualized the source code of each class separately using Eclipse and produced Excel files[12] containing the number of occurrences of each modifiability/maintainability defect for each class and the total number of defects in the entire project. Some defects have been missed by mistake due to the nature of the task.
We report the results of the detection phase in table 3.3. It provides for the four systems the number of existing smells, the result of manual analysis (M) in the first line of each row, the smells detected by our implementation (A for automatic) in the second line, and the precision in the third line. For the four systems, all the defects detected manually are also detected automatically. Table 3.3 shows that a large percentage of the results obtained with our approach are validated manually (from 87% in average for FreeCS to 94% for Jasml)
We have calculated *MI* values to compare systems before and after applying our approach on the four projects. Since comments will not be added with the created elements (interfaces, methods and fields), we used the *MI2* formula in our study. This formula is calculated at the class level since we have used a trial copy of the *JHawk* tool. In this copy, the developers have mentioned that a number of Java files can be selected for parsing but only the results of a few classes will be displayed. Table 3.4 shows the MI scores before and after applying the method on the four projects. *MI* represents the average of the classes' MI value.

[11] http://www.virtualmachinery.com/index.htm
[12] https://www.dropbox.com/sh/whjczwv6qvbglq2/AADh7jU6p23SVjdNAWR4IQ-da?dl=0

Table 3.3: Detected smells (In each row, M = results of Manual analysis; A = results of Automatic analysis).

System		Public & package fields	Public non-static methods not exposed	Fields of concrete class type	Instantiations that can be injected	Avg
Jasml-0.10	M	420	48	26	41	
	A	447	49	27	46	
	Prec	93.96%	97.96%	96.29%	89.13%	94%
CoCoME	M	32	221	25	82	
	A	34	223	29	85	
	Prec	94.11%	99.1%	86.20%	96.47%	93%
FreeCS-1.3	M	380	571	83	79	
	A	402	837	88	86	
	Prec	94.52%	68.22%	94.31%	91.86%	87%
Log4j-1.2.17	M	365	750	150	105	
	A	370	753	167	133	
	Prec	98.65%	99.6%	89.82%	78.95%	91%

Table 3.4: MI values before and after applying the method

System	MI before	MI after	Improvement factor
Jasml-0.10	125.49	147.95	1.17
CoCoME	125.12	161.64	1.29
FreeCS-1.3	110.21	120.89	1.09
Log4j-1.2.17	114.52	122.68	1.07

From the results of Table 3.4, it is clear that the maintainability index is better, with an improvement factor that ranges from 1.09 to 1.29, when applying our approach on the listed systems. The improvement in maintainability, according to this metric, is not an insignificant score, regarding the size of these systems.
The improvement in *MI* scores is mainly due to the use of new interfaces. These interfaces have a high *MI* score comparing to concrete classes. The main reason is that an interface has a low value of *LOC*, comparing to classes. When the average of MI on the classes and interfaces is calculated, the MI value tends to have a better score. Another reason of the improvement in MI scores, is the low Halstead Volume (V) value of added setters and getters to classes. When the average of V on the methods of a class is calculated, the V value tends to have a lower value.

Better MI scores of interfaces are justified. This is because interfaces define methods that accomplish specific functionalities without stating how such functionalities will actually work. The real implementation is provided by the class implementing the interface. Therefore, during maintenance tasks, new maintenance requirements can be defined in separate classes that are used by the system in the same way as other already existing classes. The add of new classes makes the maintenance task easier. Then, in order to check if during the evolution of a single system, the proposed refactorings keep stable this improvement in modifiability/maintainability, we evaluated MI for six versions of *Log4j API*, which where developed over a period of 17 years.
The MI values for the six versions before and after applying the proposed refactorings are depicted in Table 3.5. From this table, we can see that there is an increase in MI values of Log4j, before applying our approach, in all the analyzed versions. This is justified by the fact that from a version to another, new functionalities are added to the system or bugs are corrected, but developers of this system pay attention to its maintainability. As an example of modifications that have been performed in version 1.0.4 and contributed to improve the maintainability of version 1.1.3: *FileAppender* class from *org.apache.log4j* package has been splitted into three classes (*ConsoleAppender*, *WriterAppender* and *FileAppender*) and the MI value for this class passed from 120.47 to 122.75 (the average MI for the three new classes).

Table 3.5: MI values of Log4j versions.

Version	# Classes	MI before	MI after	Imp. factor
1.0.4	146	111.31	121.59	1.09
1.1.3	162	112.25	122.47	1.09
1.2.1	179	114.81	120.66	1.05
2.0	87	116.08	119.64	1.03
2.4	112	117.66	121.56	1.03
2.8	172	118	121.16	1.02

As we can observe, in all the versions, the maintainability is improved. However, the improvement is greater in the first versions. This is explained by the fact that starting from the (major) version 2.0, the structure of *Log4j* has completely changed, and its maintainability was substantially improved. In the following versions, the system keeps a good *MI* score, even if this is slightly improved by our refactorings. This shows that our refactorings give better results on old versions of legacy systems, compared to new, potentially refactored, ones.

3.3.4 Threats to Validity

3.3.4.1 Internal Validity

On threat concerns the fact of injecting only instances that are not surrounded by a control flow statement. In order to investigate the applicability percentage of the "use dependency injection" refactoring, we checked the number of instances that can not be injected. For the four systems, instances that can not be injected range from 19% in the case of CoCoME to 71% in the case of FreeCS. These are high percentages that require to be taken into account in the future.
The obtained results in the detection phase depend on the specification of modifiability/maintainability defects and on the profile of students. We tried to be the most accurate possible in the description of defects and we have chosen students who have some experience in Java programming. Another aspect can bias the results is related to the number of persons involved in the experiments: one student was assigned to one system or to a part of a system. Several persons should be assigned per system to have more accurate results. In our study, we gave these students large periods of time (2 weeks in average) to carefully check the defects.

3.3.4.2 External Validity

We tried to collect systems of different sizes and developed by different teams to diversify the data. It is sure that with a larger set of systems we may obtain more precise results. However, since the results were all positive with the four studied systems, which vary in size, our intuition, on the interest of transforming OO code into CB one using the proposed refactoring operations, is strengthened.

3.4 Conclusion

We presented in this chapter an approach for improving the modifiability/maintainability of object-oriented source code, by focusing on what component-based development brought to programming, i.e. decoupling and non anticipated instantiations. Our approach was experimented on a set of Java projects to evaluate its efficiency in the detection of modifiability/maintainability defects, and the improvement it brings to maintainability. The results of this experimentation helped to answer of the first research question 1 in Chapter 1 and showed that there is a potential in using the proposed process in migrating existing legacy OO systems.
Perspectives of this work include the experimentation of this approach on a larger set of projects with larger sizes. From a tool-support point of view, our prototype solution could be improved and integrated to the Eclipse IDE as a monolithic refactoring solution with Eclipse already existing refactorings in order to experiment its usability by maintainers in their real-life maintenance tasks.

Chapter 4

Recovering the Runtime Architecture of Object-Oriented Software Systems and Managing its Complexity

Contents

4.1 Introduction and Problem statement

We presented in the previous chapter an approach which focuses on the improvement of the maintainability quality attribute of legacy systems. Experiments showed that this approach performed well on the studied systems. However, more experiments on a larger set of systems (with larger sizes) is needed and envisaged in the future. Another important quality attribute that has a direct impact on maintainability is understandability. In order to improve this latter, a high level view, an architecture model, of the system's structure and behavior is needed. The evolution of software systems over time often leads to an erosion of its architecture. In this case, an architecture recovery process should be carried out in order to build the as-implemented architecture of the system to be evolved.
This chapter focuses on the second contribution developed in this thesis. We propose an approach for recovering runtime architectures and managing the complexity of the recovered architectures. To this end, static and dynamic analyses are combined. Static analysis is used to build object graphs (OG); i.e., graphs where the nodes represent objects and the edges represent objects' field assignments. For large software systems, such graphs often contain hundreds or thousands of nodes and edges, hence directly viewing such a graph is of no help. Thus, these graphs are refined using information obtained through the analysis of execution traces. The information added to these graphs includes the lifespan of each object and its "empirical" probability of existence at runtime. This added information is used to reduce the complexity of the resulting refined graph according to the developer preferences. In fact, developers can use this information to set thresholds and reduce the size of the graph focusing, for instance, on the most likely or durable objects. Furthermore, composite (internal) structures of objects are identified in the graph. This organizes the refined OG into a hierarchy of composite structures/nodes that can be collapsed or expanded to hide or show their internal structure. We experimented the approach through examples conducted on Java open-source projects. These examples showed that our approach recovers the runtime view of the analyzed system and effectively manages the complexity of this view in order to handle some understanding tasks.
In this chapter, Section 4.2 presents a general overview of the approach which is defined as a multi-step process. Sections 4.2.2 to 4.2.6 detail each step of the process. We present the experimentation of this process and discuss our observations in Section 4.3. We conclude in Section 4.4.

4.2 Foundations of the proposed Approach

4.2.1 The Process in a Nutshell.

The proposed process is depicted in the activity diagram in Figure 4.1. This figure presents the six main steps (activities) of the OG recovery process. The first step is a static analysis of the source code (a1) from which an initial OG, similar to the one introduced in [Tonella 2005], is recovered. The second step (a2) consists of automatically instrumenting source code, inserting statements at specific locations, to create logs about object creation and destruction. This instrumented code is then executed in the third step of the process (a3) using a set of test cases or passed to users for a period of time. The output of this step is a set of execution traces. The steps a1 and a2+a3 can be executed in parallel. The generated traces are analyzed to extract information to refine the preliminary OG (a4). The fifth step (a5) of the process consists in managing the complexity of the refined OG using two techniques: i) exploiting the lifespans and probabilities of existence, and ii) identifying the so-called composite structures, which make the graph hierarchical and thus reduce its complexity. Indeed, the identification of composition relations between objects enables us to build the composite structures of objects in the form of hierarchical nodes in the graph, which embed their inner objects (inner nodes) and their inter-relationships (inner edges). In order to understand the runtime architecture of a given software system, developers can customize their visualization (a6) by focusing on the most durable objects and/or the most likely to exist at runtime. Developers can also focus on particular objects by unfolding hierarchical nodes to analyze their composite structure, or to visualize a high level view of the architecture (the graph hiding the internal composite structures). Sections 4.2.2 to 4.2.6 discuss in detail the different steps of the process for recovering this refined hierarchical graph.

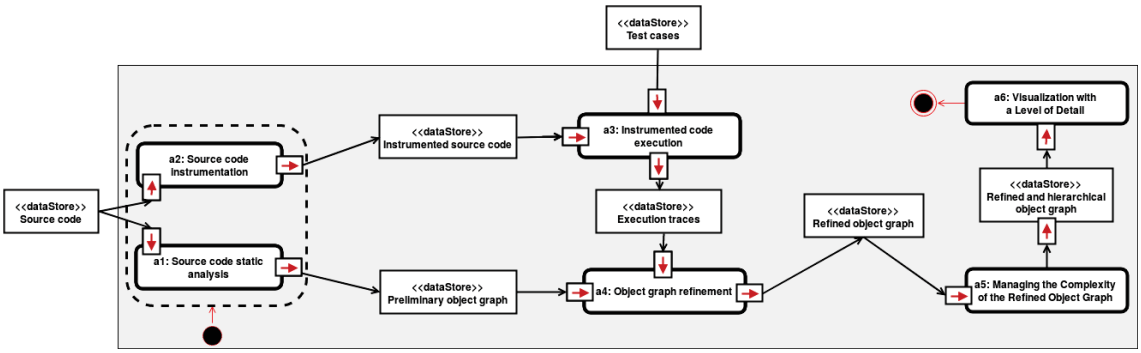

Figure 4.1: Process for the creation of a refined hierarchical object graph

4.2.2 Source code static analysis

This first step of the process aims at building a preliminary OG of the system under study. This is achieved by a static analysis of the source code which consists of reasoning about the behavior of a program without actually running it.
An *OG* $(Nodes_{OG}, Edges_{OG})$ is a directed labeled graph that represents the structure of a given software system in terms of objects. In this graph, $Nodes_{OG}$ denote objects. A directed *edge*$(o_1, o_2) \in Edges_{OG}$ indicates that o_1 has obtained a reference to o_2 at some point during its execution, and this reference was assigned to one of o_1's fields. Edges are labeled with field names. The focus on field assignments is motivated by the fact that fields store and affect the state of the object while local variables and method parameters are simply short-lived variables for executing a method. Moreover, fields reflect the design of the application, since they characterize the structure of objects.
Since objects are not necessarily directly assigned to fields, the recovery of this preliminary OG relies on another graph named the Object Flow Graph (OFG). This OFG allows tracking objects created during system execution from their creation until the storage of their references in fields or their usage in method invocations. An OFG is a directed graph in which nodes can be of two types: i) Objects and ii) Program variables (fields, local variables, methods' parameters or methods' arguments). Edges of the graph represent assignments between these variables.
To build the OFG, we are interested in three kinds of statements:

- Allocation sites (x = new constr([a_1, a_2, ..., a_n]);),
- Assignment sites (x = y;) and
- Invocation sites ([x =][1] y.meth([a_1, a_2, ..., a_n]);)

For variables x, y, a_1, a_2 and a_n we are only interested by those which are typed by user-defined classes/interfaces or collections of user-defined classes/interfaces. We ignore types from libraries in order to limit the developer focus on the system code only.
Objects in the OFG are collected from allocation sites and the flow/track of each object is inferred by analyzing the statements in which the reference of this object is used.

4.2.2.1 Object Flow Graph Recovery

Algorithms 1 and 2 present how the object flow graph is recovered based on an object sensitive analysis. Object sensitivity means that the program variables are distinguished by the objects they belong to instead of their classes.
In Algorithm 1, *callChains* refers to the call chain of the system starting from the entry point main method.

[1] Optionally because we can find in source code x = y.meth(...) or simply y.meth(...)

Algorithm 1 OFG Edges Construction

Input: SS : *source code of the studied system*
Output: *OFG edges* E_{ofg}
  $callChains \leftarrow LinkedList$
  $E_{ofg} \leftarrow \emptyset$
  $main \leftarrow getEntryPointMethod(SS)$
  $E_{ofg} \leftarrow E_{ofg} \cup analyseBlock(main.getBody(), callChains)$
  while $\neg callChains.isEmpty()$ **do**
    $element = callChains.removeFirst()$
    $E_{ofg} \leftarrow E_{ofg} \cup analyseBlock(element.getDeclaration().getBody()$
    $, callChains)$
  end while
  return E_{ofg}

E_{ofg} refers to the object flow graph edges. The *getBody* function returns the block of statements enclosed in curly brackets of the current receiver. The variable *element* can be either a constructor call or a method invocation. *getDeclaration* function establishes a link between the element and its definition.
The *analyseBlock* function in Algorithm 2 returns a subset of the object flow graph edges local to the analyzed method/constructor. *getStatements* function returns the set of statements[2] contained in the analyzed block. *getUserDefinedTypedVariables* returns the set of variables whose types are user-defined. For example, let *a = new A(b, 8, " ")* where *b* is of type *B*. *A* and *B* are user defined classes. In this case, *getUserDefinedTypedVariables* returns *a*, *b* and the formal parameter in the A constructor that corresponds to b.
For the three kinds of statements we are interested in, the *addEdge* function adds the following edges to the set of the OFG edges[3]:

- (i) For x = new *constr*([a_1, a_2, ..., a_n])

  1. An edge between the created object *constr'.this* and the object scoped variable x'.
  2. Edges between the object scoped arguments a'$_1$, a'$_2$, ..., a'$_n$ and the corresponding *constr* object scoped formal parameters f'$_1$, f'$_2$, ..., f'$_n$.

  Example: Suppose that we have the allocation sites *c = new C(); a = new A(c);* inside a class *B* such that *a* is of type *A* and *c* is of type *C*. *A*, *B* and *C* are user defined classes. When the class B is instantiated, Edges {*B1.c = C1.C.this, B1.a = A1.A.this, A1.A.c*[4] *= B1.c*} are added to the set of edges.

[2]Allocation, assignment and invocation sites
[3]a_1, a_2, ..., a_n correspond to class scoped variables and a'$_1$, a'$_2$, ..., a'$_n$ correspond to object scoped variables.
[4]This refers to the formal parameter *c* in the constructor *A* of the object *A1*

Algorithm 2 analyseBlock function

Input: *The block of an element and callChains*
Output: *Local egdes in the block*

```
1:  localCallChains ← LinkedList
2:  localEdges ← ∅
3:  scopes ← ∅
4:  statements ← getStatements(block)
5:  for s : statements do
6:     UDTV ← getUserDefinedTypedVariables(s)
7:     for v : UDTV do
8:        scopes ← scopes ∪ scope(v)
9:     end for
10:    localEdges ← localEdges ∪ addEdges(s, scopes)
11:    if s.containsConstructorCalls() ∨ s.containsInvocations() then
12:       constructorCalls ← getConstructorCalls(s)
13:       methodCalls ← getMethodCalls(s)
14:       for c : constructorCalls do
15:          localCallChains.add(c)
16:       end for
17:       for mc : methodCalls do
18:          localCallChains.add(mc)
19:       end for
20:    end if
21: end for
22: while ¬localCallChains.isEmpty() do
23:    e = localCallChains.removeLast()
24:    callChains.addFirst(e)
25: end while
26: return localEdges
```

- (ii) For x = y,

  1. An edge between the object scoped variable x' and the object scoped variable y';

  Example: Suppose that we have the statements *c = new C(); C sc = c;* inside a class *B* such that *c* is of type *C*. *B and C* are user defined classes. When the class B is instantiated, Edges {*B1.c = C1.C.this, B1.sc = B1.c*} are added to the set of edges.

- (iii) For [x =] y.meth([a_1, a_2, ..., a_n])

  1. Edges between the object scoped arguments a'$_1$, a'$_2$, ..., a'$_n$ and the corresponding *meth* object scoped formal parameters f'$_1$, f'$_2$, ..., f'$_n$.
  2. An edge between the target object *this* of the object scoped invoked method *meth'.this* and the object scoped variable y'.
  3. An edge between the object scoped *return* value of the invoked method *meth'.return* and the object scoped variable *x'*.

  Example: Suppose that we have the statements *d = new D(); a = new A(); c = a.createC(d);* inside a class *B* such that *a* is of type *A*, *c* is of type *C* and *d* is of type D. *A, B, C and D* are user defined classes. When the class B is instantiated, Edges {*B1.d = D1.D.this, B1.a = A1.A.this, A1.createC.this = B1.a, B1.c = A1.createC.return, A1.createC.d = B1.d*} are added to the set of edges.

Object scoped variables a'$_1$, a'$_2$, ..., a'$_n$, f'$_1$, f'$_2$, ..., f'$_n$, y' and x' are represented by a form of fully qualified names of the variables a_1, a_2, ..., a_n, f_1, f_2, ..., f_n, y and x. However, the class name is replaced by the object identifier since our analysis is object sensitive (for example, *B1.a* instead of *B.a*). This object identifier is called "scope".
Scopes of the different variables can be obtained, by the *scope* function, as follows:

- In (i), scope(f'$_1$), scope(f'$_2$), ..., scope(f'$_n$) and scope(*constr'*) is the object identifier of the allocation site (i).

- In (i), (ii) and (iii) two cases must be taken into account:

  1. If x, y, a_1, a_2, ..., a_n are local variables, current method parameters or current object fields, scope(x'), scope(y'), scope(a'$_1$), scope(a'$_2$), ..., scope(a'$_n$) is the object identifier scoping the current method.
  2. If x, y, a_1, a_2, ..., a_n are accesses to fields of an object other than the current one, of the kind *v.field*, scope(x'), scope(y'), scope(a'$_1$), scope(a'$_2$), ..., scope(a'$_n$) is the output of *v*. The output of a variable means the object stored in this variable. For example, if somewhere in the code we have *v = new constr()*, output(v) is the object identifier of this allocation site.

- In (iii), two cases must be taken into account:
  1. If *y.meth* is an invocation performed on the current object, scope(meth), scope(f'$_1$), scope(f'$_2$), ..., scope(f'$_n$) is the object identifier scoping the current method.
  2. If *y.meth* is an invocation performed on an object other than the current one, scope(meth), scope(f'$_1$), scope(f'$_2$), ..., scope(f'$_n$) is output(y).

Listing 4.1: MovieCatalog class example

```
public class MovieCatalog extends WmvcApp {

  private MainView mainView;
  private MovieListView listView;
  private MovieItemView itemView;

  public MovieCatalog(String name) {
    MovieModel movieM =  new MovieModel();
    setModel(movieM);
    mainView = new MainView();
    listView = new MovieListView();
    itemView = new MovieItemView();
  }

  public static void main(String[] args) {
    MovieCatalog movieCat=new MovieCatalog("");
    movieCat.showApp();
  }
  // The following elements are extended form WmvcApp
  // we put them here to facilitate understanding
  private static MovieModel theModel;
  public static void setModel(MovieModel m){
        theModel = m;
  }
}
```

To have a more clear insight on how OFG graphs are recovered, consider the example in listing 4.1, of the *MovieCatalog* class [Wampler 2002]. This a class that belongs to a small-sized application, 19 classes, which is based on the Model-View-Controller (MVC) pattern. This application allows a user to lookup, create, edit and delete movies.
Starting from the *main* method, the call chain is:
new MovieCatalog("") > new MovieModel() > setModel(movieM) > new MainView() > new MovieListView() > new MovieItemView() > movieCat.showApp()

Scopes and outputs of the different variables are presented in Table 4.1 and the set of the OFG edges are depicted in Listing 4.2 (objects are highlighted in crimson). Edges are presented as assignments between nodes. The assignment's right and left hand sides represent the edge's source and target nodes respectively.

Table 4.1: Scopes and outputs

Variables	**movieCat**	**movieM**	**m**	**theModel**
Scope	MovieCatalog[5]	MovieCatalog1	MovieCatalog1	MovieCatalog1
Output	MovieCatalog1	MovieModel1	MovieModel1	MovieModel1

Variables	**mainView**	**listView**	**itemView**
Scope	MovieCatalog1	MovieCatalog1	MovieCatalog1
Output	MainView1	MovieListView1	MovieItemView1

Listing 4.2: OFG edges

```
1 MovieCatalog.main.movieCat = MovieCatalog1.MovieCatalog.this
  MovieCatalog1.MovieCatalog.movieM = MovieModel1.MovieModel.this
3 MovieCatalog1.setModel.m = MovieCatalog1.MovieCatalog.movieM
  MovieCatalog1.theModel = MovieCatalog1.setModel.m
5 MovieCatalog1.mainView = MainView1.MainView.this
  MovieCatalog1.listView = MovieListView1.MovieListView.this
7 MovieCatalog1.itemView = MovieItemView1.MovieItemView.this
  MovieCatalog1.showApp.this = MovieCatalog.main.movieCat
```

The corresponding OFG is presented in Figure 4.2. It traces the flow of five objects (the *solid-line red circles*) from their creation by allocation sites until their assignment to class fields (*dashed-line green circles*), or their usage in method invocations (*dotted-line blank circles*). Fields are represented by *ObjectIdentifier.fieldName*. *ObjectIdentifier* represents the class name followed by an integer which is incremented each time an instantiation of this class is found. Method local variables and parameters are represented by *ObjectIdentifier.MethodName.VarName/ParamName*. *ObjectIdentifier* is replaced by the class name in case of static members.
For example, the object *MovieCatalog1* is stored in the local variable *movieCat*. After that, *movieCat* is used to invoke the *showApp* method, so *movieCat* is assigned to the target object *this* of *showApp*, which is represented by the edge between *MovieCatalog.main.movieCat* and *MovieCatalog1.showApp.this* in the OFG in Figure 4.2, and so on for the other objects.

4.2.2.2 Preliminary Object Graph Recovery

Once the OFG is obtained, the OG can be recovered by analyzing the output sets of the OFG nodes that correspond to fields, *dashed-line green circles*.

[5]It is the root node of the graph

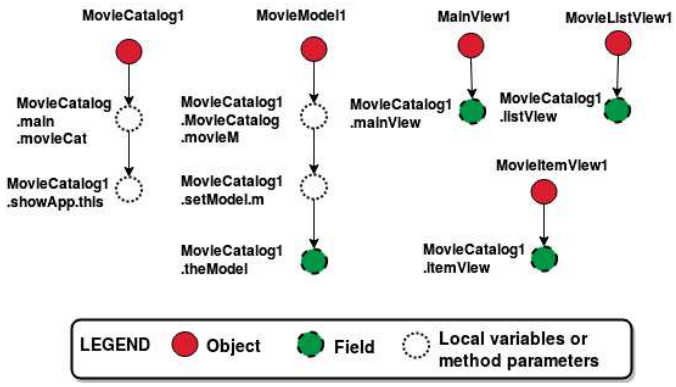

Figure 4.2: The OFG of the *MovieCatalog* class

The output sets are presented in Listing 4.3 and the preliminary object graph is depicted in Figure 4.3.

Listing 4.3: Output sets

```
Output[MovieCatalog1.theModel] = {MovieModel1}
Output[MovieCatalog1.mainView] = {MainView1}
Output[MovieCatalog1.listView] = {MovieListView1}
Output[MovieCatalog1.itemView] = {MovieItemView1}
```

For example, the output of the *MovieCatalog1.theModel* field is *MovieModel1*. This is depicted in Figure 4.3 by the link labeled *theModel* between *MovieCatalog1* and *MovieModel1*.

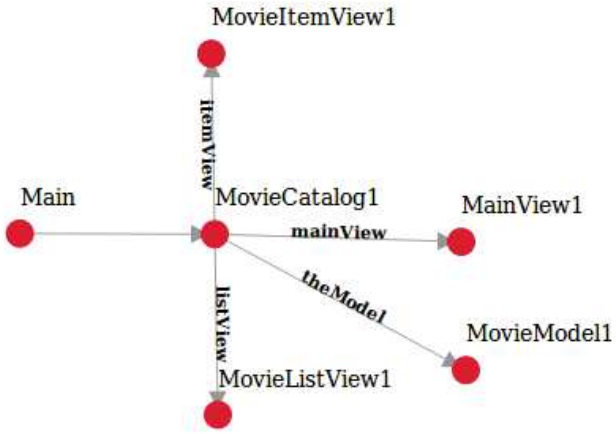

Figure 4.3: The OG of the *MovieCatalog* class

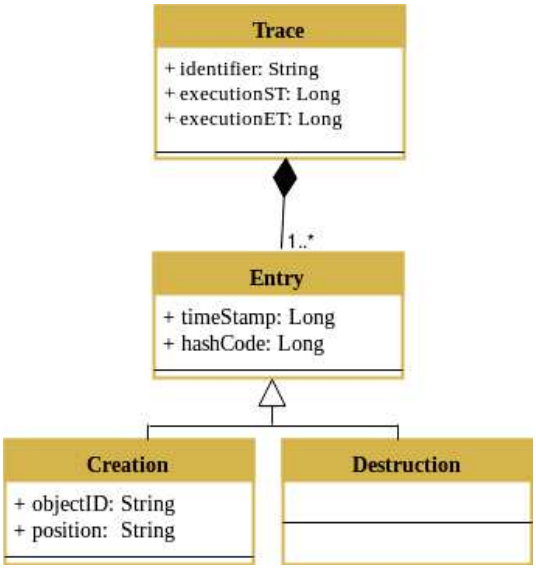

Figure 4.4: Trace metamodel

4.2.3 Source Code Instrumentation & Instrumented Code Execution

4.2.3.1 Source Code Instrumentation

In order to produce execution traces, an instrumentation strategy has been worked out. The instrumentation consists in automatically adding statements in specific places of the source code. When executing the instrumented code, the added statements produce in a *Log* file (execution trace) a text representing the runtime information.
The trace metamodel is depicted in Figure 4.4. From this metamodel, runtime information reported in each trace can be summarized in the following:

- System start *(executionST)* and end *(executionET)* timestamps,
- Object creation: creation timestamp, object identifier, the position (class name + line number) of the allocation site responsible for creating the object and the object hashcode, and
- Object destruction: destruction timestamp and the hashcode of the destroyed object.

In order to get this runtime information, a field *(objectID)* and two methods (*objectIDgenerator()*, to generate object identifiers, and *hashCodeGenerator()*, to generate object hashcodes) are added to the code of each class of the system in question. A class *Logger* is also added to the set of classes of the system under study. This class contains methods that enable writing the log file.

Since some languages, like Java, do not have explicit destructors, the instrumentation of object destruction may not be straightforward. In this case, one option consists of tracking all the references of an object. If the last reference of this object is reassigned to another object or to null, this reassignment statement is considered the destruction site of the object. The destruction timestamp can be then determined by instrumenting this statement. Another simpler option, in Java, consists of overriding the *finalize()* method if it does not exist and changing its body to log destruction timestamps and the hashcode of the destructed object. In the current work, the second option is used.
The code of the automatically added elements, the *Logger* class, the instrumented code of the *MovieCatalog* class and the code of an execution trace generated when running the instrumented *MovieCatalog* class are presented in listings 4.4, 4.5, 4.6 and 4.7 respectively.

Listing 4.4: Added elements in the instrumentation phase

```
public static int ObjectID = 0;

public static int objectIDgenerator() {
 return ObjectID++; }

public int hashCodeGenerator() {
 return System.identityHashCode(this); }

protected void finalize() throws Throwable {
 super.finalize();
 loggerClass.log("Destruction:"+
   System.identityHashCode(this));}
```

Listing 4.5: The Logger class

```
public class Logger {

    public static PrintWriter writer;

    public static void log(String msg) {
      if(writer == null){
         try{
          writer = new PrintWriter("TraceName.log",
           "UTF-8");
          } catch(java.lang.Exception e) {}
         }
         writer.println("TimeStamp= "+ Long.toString(
         System.nanoTime()) + ", " + msg);
         writer.flush();
```

```
    }
16
    public static <T> T logNewInstance(T instance,
18  String  objectID, String position, Long hashCode) {
        Logger.log("new: "+objectID+", "+ position+",
20       "+hashCode);
        return instance;
22  }

24 }
```

Listing 4.6: The instrumented code of the *MovieCatalog*

```
public class MovieCatalog extends WmvcApp {
2   private MainView mainView;

4   private MovieListView listView;

6   private MovieItemView itemView;

8   public MovieCatalog(String name) {
        super(name, true, true);
10
        MovieModel movieM = Logger.logNewInstance(
12      new MovieModel(),
        "ObjectID = MovieModel"
14      +MovieModel.objectIDgenerator(),
        "Position = MovieCatalog: 9",
16      "Hashcode = "+movieM.hashCodeGenerator()
        );
18      setModel(movieM);

20      mainView = Logger.logNewInstance(
        new MainView(),
22      "ObjectID = MainView"
        +MainView.objectIDgenerator(),
24      "Position = MovieCatalog: 11",
        "Hashcode = "+mainView.hashCodeGenerator()
26      );

28      listView = Logger.logNewInstance(
        new MovieCatalogue.MovieListView(),
30      "ObjectID = MovieListView"
        +MovieListView.objectIDgenerator(),
```

```
32          "Position = MovieCatalog: 12",
            "Hashcode = "+listView.hashCodeGenerator()
34          );

36          itemView = Logger.logNewInstance(
            new MovieItemView(),
38          "ObjectID = MovieItemView"
            +MovieItemView.objectIDgenerator(),
40          "Position = MovieCatalog: 13",
            "Hashcode = "+itemView.hashCodeGenerator()
42          );
        }
44
        public static void main(String[] args) {
46          MovieCatalog movieCat = Logger.logNewInstance(
            new MovieCatalog(""),
48          "ObjectID = MovieCatalog"
            +MovieCatalog.objectIDgenerator(),
50          "Position = MovieCatalog: 20",
            "Hashcode = "+movieCat.hashCodeGenerator()
52          );
            movieCat.showApp();
54      }

56      public static int ObjectID = 0;

58      public int hashCodeGenerator() {
            return System.identityHashCode(this);
60      }

62      public static int objectIDgenerator() {
            return ObjectID++;
64      }

66      @Override
        protected void finalize() throws Throwable {
68          super.finalize();
            Logger.log("Destruction: "
70          +System.identityHashCode(this));
        }
72  }
```

Listing 4.7: A trace example

```
Start time: 1493642333528
TimeStamp= 1493642333681, new:ObjectID = MovieModel1,
Position = MovieCatalog: 6, HashCode = 186370029
TimeStamp= 1493642333861, new:ObjectID = MainView1,
Position = MovieCatalog: 8, HashCode = 1915503092
TimeStamp= 1493642333862, new:ObjectID = MovieListView1,
Position = MovieCatalog: 9, HashCode = 1567581361
TimeStamp= 1493642333866, new:ObjectID = MovieItemView1,
Position = MovieCatalog: 10, HashCode = 1688376486
TimeStamp= 1493642333889, new:ObjectID = MovieCatalog1,
Position = MovieCatalog: 13, HashCode = 1793329556
End time: 1493642837688
```

4.2.3.2 Instrumented Code Execution

The next step consists of executing the instrumented system using test cases or pass it to users to be used for a period of time in order to generate execution traces. In order to guarantee coverage of the generated traces, the measure of "function coverage" proposed by Hu et al [Hu 2014] is used. The "function coverage" of a given trace i is calculated using the following formula:

$$FC_i = \frac{M_{dcg}}{M_{scg}} * 100$$

where, M_{dcg} is the set of methods in a dynamic call graph (execution trace i) and M_{scg} is the similar set of the corresponding methods in a static call graph.
The dynamic call graph is built from execution traces, where method invocations/-calls are reported. Methods of the static call graph are collected by a static analysis of method calls starting from the main method of the system under study. High percentage of "function coverage" means that a high percentage of code has been traversed by the use scenario. Therefore, traces that have a high percentage of "function coverage" are used in the next step of the process.

4.2.4 Object graph refinement

The refinement consists of adding two kinds of labels on nodes of the preliminary OG: "empirical" probabilities and lifespans.
The probability of each object represents the ratio of the number of occurrences of this object in execution traces to the total number of execution traces.
Lifespans are measured using the creation and destruction timestamps, which are read from execution traces. The lifespan of a node n is a range, having as a minimal value the scaled creation timestamp (sts_c) and as a maximal value the scaled destruction timestamp (sts_d). This scaling enables to see the timestamps as per-

centages to the lifespan of the application. sts_c and sts_d are calculated using the following simple formulas:

$$sts_c(n) = \frac{1}{m} * \sum_{i=0}^{m} \frac{(cts_i(n) - st_{sys_i})}{length_{sys_i}} * 100$$

$$sts_d(n) = \frac{1}{m} * \sum_{i=0}^{m} \frac{(dts_i(n) - st_{sys_i})}{length_{sys_i}} * 100$$

where $cts_i(n)$ *and* $dts_i(n)$ are creation and destruction timestamps of the node n in the execution trace i, st_{sys_i} is the system's start time during trace i, m is the number of execution traces, and $length_{sys_i}$ is the difference between the system's start time and end time in trace i. As mentioned above, this scaling enables to see the timestamps as percentages to the lifespan of the application.
The objects created through the same allocation site (in the case of loops) have many occurrences in the same execution trace, with different object identifiers, but they are represented by only one node in the OG. These objects are identified by object identifiers and position in the source code. In this case, to label the node that represents a set of objects in an execution trace by the lifespan, we use the same formulas as previous ones, but we replace $\text{cts}_i(n)$ and $\text{dts}_i(n)$ by $AVG(cts_i(n))$ *and* $AVG(dts_i(n))$ which represent the ratio of the sum of creation/destruction timestamps of the objects that have the same position as n in the initial OG, to the total number of creations/destructions in the trace. An additional label that represents the frequency of the creation is also added to the node.
As an example, we give the recovered OG of *MovieCatalog* application. In order to generate traces, 10 test cases have been executed. The average execution time of the application is 642 seconds. The refined OG is shown in Figure 4.5.
Labels added on nodes are of the form {probability, lifeSpan}, calculated using the two previous formulas. All the nodes in Figure 4.5 have a probability of existence equal to 1 except *Movie2* which has a probability equal to 0.9. Indeed, this node appears when the user chooses to open an existing catalog of movies and add/delete/edit movies. The only trace from the generated ones that does not contain this node corresponds to the scenario when the user runs the application for the first time, so a movie catalog must be created. Concerning lifespans, almost all nodes are created at the start of the application and destroyed at the end of it, except *Movie1* and *Movie2* that were created and destroyed before the end of the application's execution.

4.2.5 Managing the Complexity of the Refined Object Graph

To manage the complexity of the refined OG, we combine two techniques. The first technique exploits the information available in our refined graph, namely the object lifespan and probability of existence. The second technique aims at identifying the composite structures of objects in the previously recovered graph.

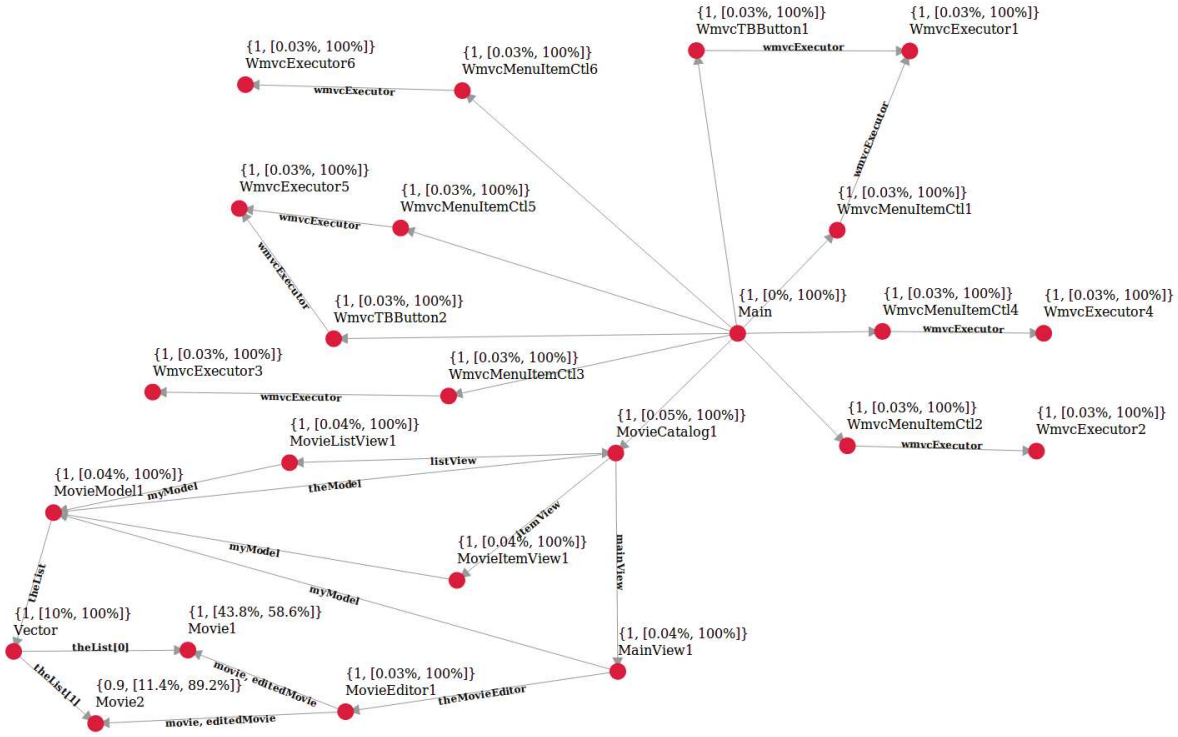

Figure 4.5: Refined OG of the MovieCatalog application

Having an OG that includes the lifespan of each object and its probability of existence at runtime, we provide assistance to manage the complexity of the OG by visualizing only relevant parts of the graph according to the developer's needs. In fact, developers can set thresholds for the values of the information added to the graph in order to focus, for instance, on objects that are the most durable or the most likely to exist at runtime. In general, we expect the objects that constitute the GUI to be the most durable; i.e., they are created when the software system is launched. Conversely, depending on the complexity of the application domain of the system, some domain-specific (business) objects may be more or less durable depending on the importance of the object in the domain. Due to the size and scope of the *MovieCatalog* example, most objects are durable; i.e., the number of use cases is limited and there is only one unique view in the GUI. In practice, the more business processes supported by the system, the greater is the number of business objects, and the probability of existence of a business object depends on the frequency of executing the business processes that act on it. Thus, filtering out business objects that have the lesser probability enables the developer to focus on the main business objects.

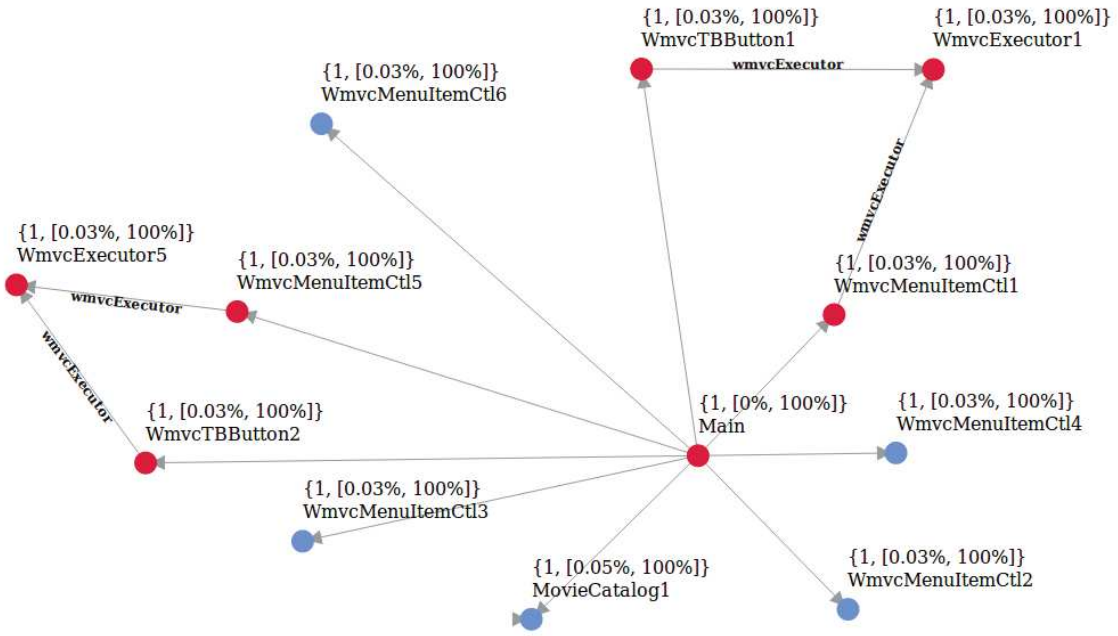

Figure 4.6: Refined and Hierarchical OG of the MovieCatalog application

The second technique identifies composite structures in the refined OG based on the notion of dominator. An object cannot be exposed outside of the boundary of its dominator. In other words, all access paths to the dominated object should pass through its dominator. In general, the dominator set for a graph nodes is calculated using the following equation:

$$dom(n) = \{n\} \cup (\cap_{(m \in predec(n)\}} dom(m))$$

where *predec(n)* = the set of all predecessors of the node n
Many algorithms have been proposed for finding dominators in graphs [Allen 1970, Allen 1972, Cooper 2001, Aho 1972]. The algorithm used in our work is that of Lengauer-Tarjan (more details on this algorithm can be found in [Lengauer 1979]) because it is the most widely used fast dominance algorithm [Cooper 2001].
When we apply this algorithm on the *MovieCatalog* system, we obtain the refined and hierarchical object graph, which is depicted in Figure 4.6, with *MovieCatalog1, WmvcMenuItemCtl2, WmvcMenuItemCtl3, WmvcMenuItemCtl4 and WmvcMenuItemCtl6* nodes collapsed. Although the recovered graph is of a manageable size, the application of the Lengauer-Tarjan algorithm showed that the number of the final visualized objects can be reduced by 50%, from 24 to 12. In Figure 4.6, blue nodes represent composite nodes whose internal structure is hidden and red nodes represent simple nodes.

4.2.6 Visualization with a level of detail

As mentioned earlier, our visualization is a user-oriented one, where the user is able to steer the displayed view when she/he is interested in identifying focal objects depending on a particular goal. In order to do this, an interface is displayed to the user. Through this interface, she/he can: i) set thresholds for lifetime interval, using a slider, and the probability value, and ii) choose to identify subsystems (composite structures) of the system under study or not. Once the settings were defined, the user can launch filtering. Filtering enables the selection of a subset of the nodes to be displayed in the view. Filtering removes uninteresting nodes from the view not fulfilling the criteria of the filters set by the user (lifetime interval and probability value). This technique of visualization reduces the cognitive effort spent to solve a particular task, required from the user to focus on a subset of the nodes when all nodes are visible [Huang 2009]. Thereby this contributes ultimately to improve understandability.
Some visualizations with a level of detail will be illustrated in the case study in the following section.

4.3 Experimental Results and Evaluation

In order to evaluate our approach, we conducted an experiment on a set of open source Java projects. In the remainder of this section, we present first the research questions that we want to answer, then the setup of the experiment, and at last the results and the threats to validity.

4.3.1 Research questions

The experiment was conducted to answer the following general research question: *To what extent does the output of the proposed approach contribute to understanding?* In order to proceed with the experiment, we need to refine this general research question. We thereby decomposed it into three sub-questions:

- **RQ1:** To what extent does the refined and the hierarchical OG contribute in reducing complexity?
- **RQ2:** To what extent does the refined and the hierarchical OG, compared to a class diagram[6], contribute to reducing the time spent for completing typical program understanding tasks and in increasing correctness of answers to questions given during those tasks?
- **RQ3:** To what extent does the refined and the hierarchical OG contribute to identifying refactoring opportunities?

[6]which is one of the most widely used diagrams to understand code structure [Ammar 2012]

4.3.2 Experiment Setup

A prototype of the method was implemented using Spoon [Pawlak 2015a], which is a source code manipulation tool. The generated graphs are defined in JSON format, which enabled us to build a Web page for their visualization.
We applied our approach on two software systems, namely Jext[7] and JHotDraw[8]. Table 4.2 provides some information on these systems. These systems were used instead of the systems used in Chapter 3 because test cases and scenarios are easier to obtain.

Table 4.2: Data collection

System	Description	#LOC	#Types
Jext	A text editor	46306	211
JHotDraw	A framework for graphics drawing	19959	269

As it was explained in the process in Section 4.2, the first step consists of recovering an initial object graph by static analysis. Then, we refine this graph with lifespans and probabilities using execution traces. To do so, in the case of Jext, we defined 15 scenarios according to the main functionalities described in the documentation. For example, we created scenarios for opening different documents (Java source files, HTML files, Zip files, etc.), performing edition activities (copy, cut, paste, comment, uncomment, etc.), searching and replacing a String, e-mailing the opened files, etc. Values of the function coverage [Hu 2014] metric for the generated traces range from 79% to 82% which are quite high values. For *JHotDraw*, five scenarios were selected. The function coverage values of the five generated traces vary from 69.82% to 73.69%. We wanted also to test our approach on an additional system which is *LogoPuzzle* which comes with *MiniDraw* [Christensen 2011]. However, for this system, only one scenario exists. This is because *LogoPuzzle* is an academic board game which is a puzzle on a university logo so, the only trace that can be generated is the one where puzzle pieces are moved to form the logo and there is no complex logic behind that. Moreover, the value of function coverage of this trace is 47%. For these reasons, this system was eliminated from the experiment.

[7] https://sourceforge.net/projects/jext/
[8] http://jhotdraw.org/

4.3.3 Results and discussion

4.3.3.1 RQ1

To provide a quantitative evaluation for this question, we used the *"Hierarchical Reduction (HR)"* metric [Vanciu 2013] which represents the ratio of the number of objects of two graphs. This measure estimates the effectiveness of the complexity management techniques compared to a flat object graph. Effective complexity management techniques would reduce the number of objects in the refined and hierarchical graph by an important number.

Table 4.3: Hierarchical Reduction (HR) results

System	L	Exploiting composite structures		probability = 1 lifespan length> 5%	
		#Rev Obj	HR	#Rev Obj	HR
Jext	0	13	6	9	5.89
	1	65	2.14	44	2.24
	2	88	1.15	63	1.2
	3	26	1.05	23	1.02
	4	10	1	3	1
	5	2	1	2	1
JHotDraw	0	43	3.4	43	3.4
	1	103	1.6	101	1.58
	2	87	1.09	83	1.08
	3	21	1.03	20	1
	4	5	1	4	1
	5	4	1	4	1

Table 4.3 presents the number of revealed objects when expanding composite nodes in each level of the object graphs in two cases: when exploiting only composite structures and when fixing the probability to 1 and selecting objects which have a lifespan length greater than 5%. A partial flat object graph of Jext and the flat object graph of JHotDraw are represented respectively in Figures 4.7 and 4.8. In the case of Jext, the displayed graphs in the two cases, of Table 4.3, are depicted in Figures 4.9 and 4.10 respectively. For JHotDraw, only the first case, when exploiting the composite structure, is depicted in Figure 4.11 since the graph in level 0 is the same in the two cases of Table 4.3.
As indicated in Table 4.3, the depth of the two graphs is 6. For Jext, the number of revealed nodes in each level, in the first case, ranges from 2 in level 5 to 88 in level 2. In the second case, the number of revealed nodes ranges from 2 in level 5 to 63 in level 2. For JHotDraw, in the first case, the number of revealed nodes in each level ranges from 4 in level 5 to 103 in level 1. In the second case, the number of revealed nodes ranges from 4 in levels 4 and 5 to 101 in level 1.

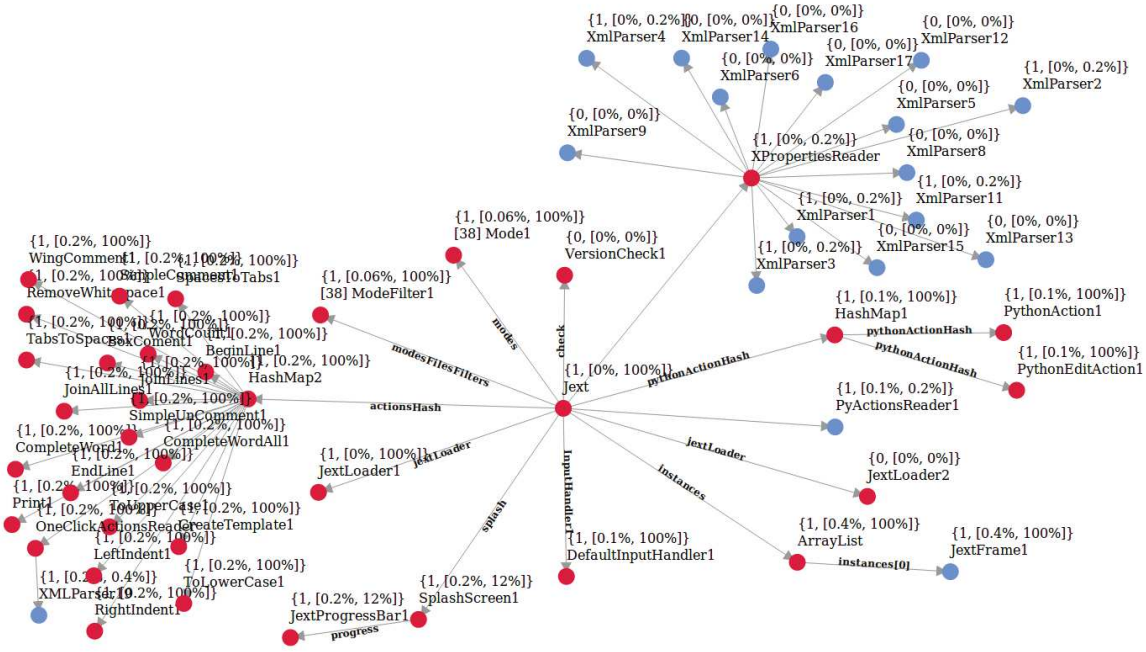

Figure 4.7: Jext partial flat object graph.

The number of objects in the initial object graph of *Jext* and *JHotDraw* is 204 and 263 respectively, which represents the sum of nodes of all the levels. Someone can wonder why the number of objects in the initial OG is less than the total number of types in the system, 211 and 269; this is explained by the facts that: i) some classes are instantiated but their instances are not stored in fields, and ii) the presence of *"Dead Code"* which represents in our case classes that are never instantiated or used. For Jext, the number of nodes of level 0 (i.e., 13) represents the final graph displayed to the user, if she/he does not fix thresholds for lifespans and probabilities. The value of HR between the flat graph and the final one displayed to the user is 15.69 which means that the number of objects in the refined and hierarchical object graph is almost 16 times smaller than the number of objects in the initial object graph. The third column, of the first case, in Table 4.3 represents the values of HR between a graph of a given level and the graph of the level above. These values range from 1 to 6. For example, HR between the top levels (5 to 4 and 4 to 3) is equal to 1, which means that there is no benefit through the hierarchical representation of objects. However, HR between level 1 and level 0 is equal to 6. It is also important to note that the exploitation of the probabilities and lifespans contributes in reducing the number of nodes displayed in each level. For example, if the user chooses to display only the objects that have a probability of one and a lifespan greater than 5%, this allows a reduction of nodes in level 0 by 4 nodes, nodes in level 1 by 21 nodes, nodes in level 2 by 25, nodes in level 3 by 3 nodes and nodes in level 4 by 7 nodes. For *JHotDraw*, the value of HR between the flat graph and the final one displayed to the

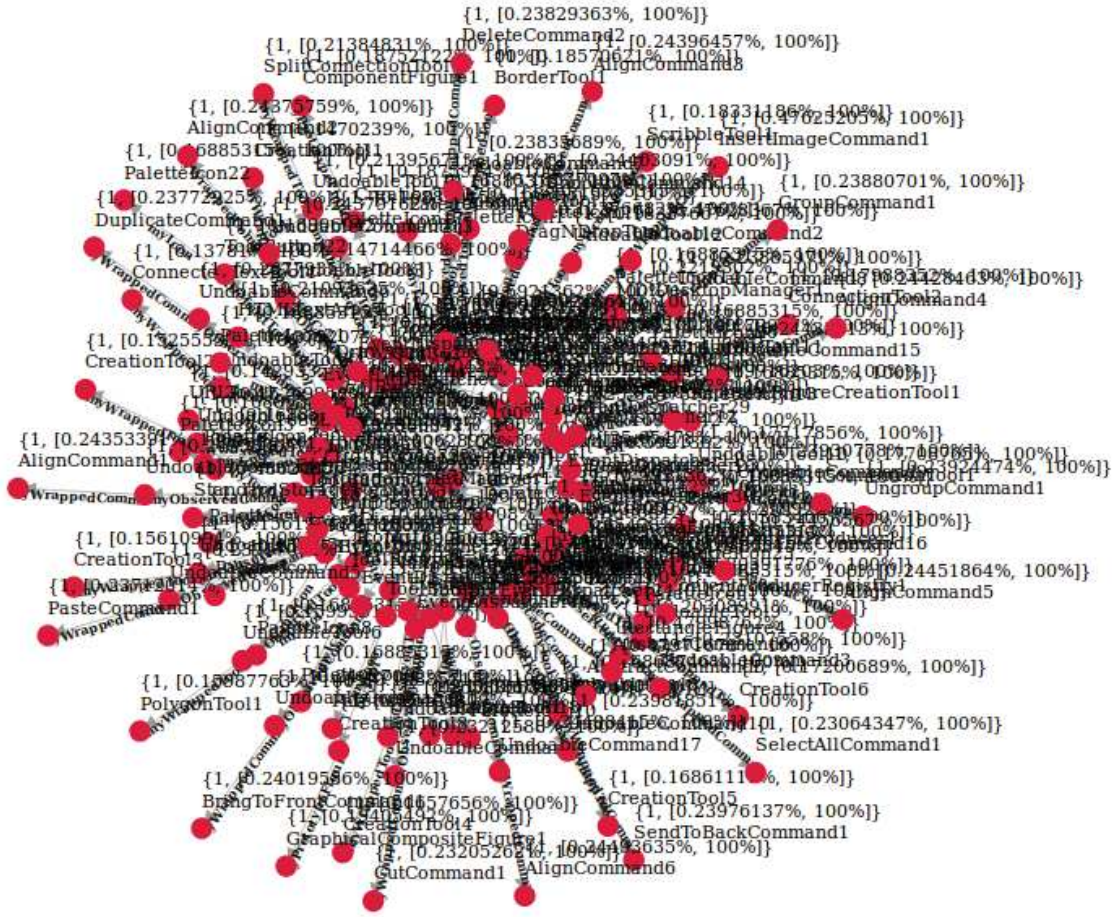

Figure 4.8: JHotDraw flat object graph.

user is 6.11 which means that the number of objects in the refined and hierarchical object graph is almost 6 times smaller than the number of objects in the initial object graph. HR values between other levels are calculated in the same way as in Jext.

It is argued that handling understanding tasks becomes particularly difficult and time-consuming when the number of nodes and edges increases [Huang 2009]. Therefore, a filtered version of the graph is visually less complex with only relevant information, from the user viewpoint, being displayed. Since our complexity management techniques allow controlling the size, and thus the complexity, of the visualized graphs by hiding and showing nodes as needed, we believe that our approach contributes in handling comprehension tasks more efficiently than using a flat object graph.

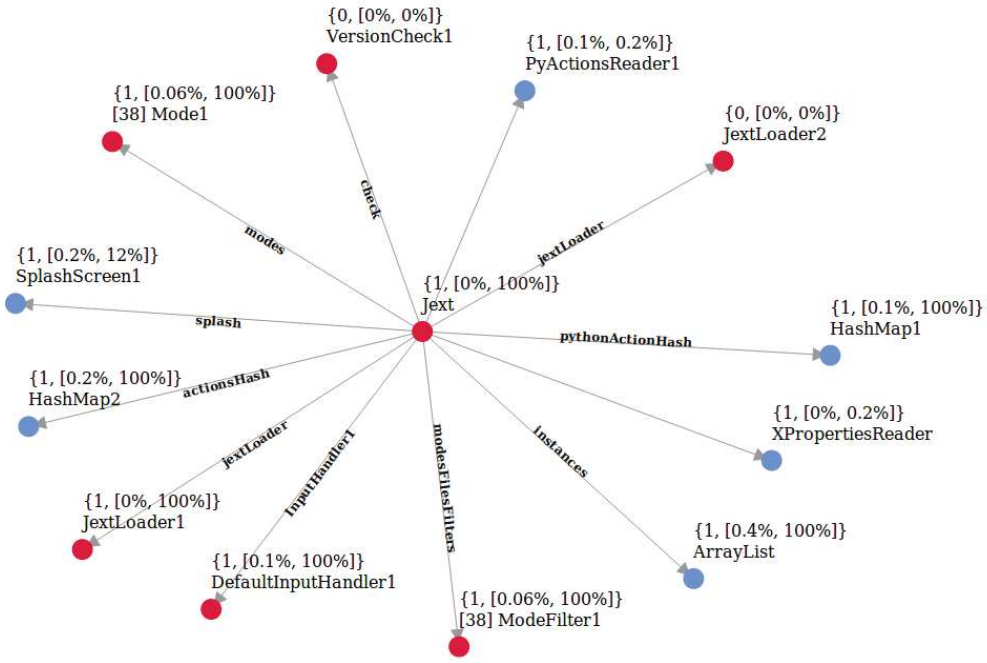

Figure 4.9: Jext refined and hierarchical object graph with only composite structure exploited.

4.3.3.2 RQ2

This research question deals with the measurement of the *spent time* and *correctness* which are typically used in the context of program understanding [Rajlich 1997]. To this end we follow the experimental design used by Cornelissen et al [Cornelissen 2009b], Fittkau et al [Fittkau 2015] and Alimadadi et al [Alimadadi 2018]. Unfortunately, it is difficult to motivate people to offer some hours from their precious time to participate in experiments. For this reason, the **RQ2** will be answered only in the case of *Jext*.

1. *Understanding Tasks:*

   The authors in [Cornelissen 2009b], [Fittkau 2015] and [Alimadadi 2018] stuck to the framework proposed by Pacione et al [Pacione 2004], which describes categories of program understanding tasks, in order to create representative tasks that highlight the aspects of the subject systems. The authors classified the tasks from literature studies according to nine principal software understanding activities that developers need to perform for understanding software, regardless of the language and the platform used. These activities are presented in Table 4.4.

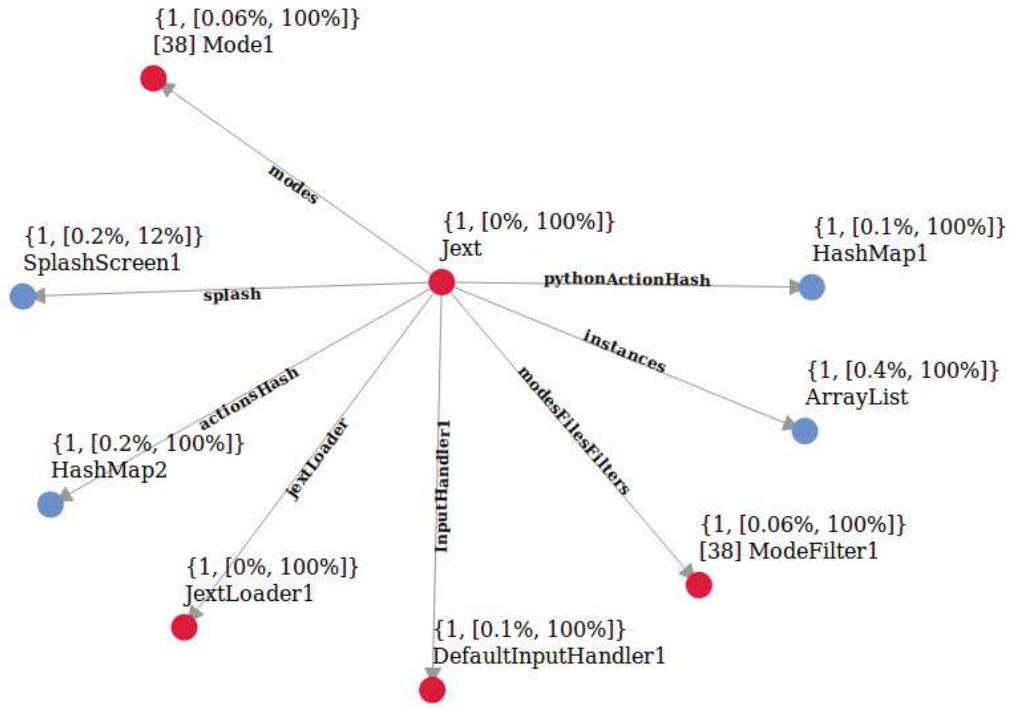

Figure 4.10: Jext refined and hierarchical object graph with composite structure, lifespans and probability exploited.

We adapted the tasks used by these authors to the context ("business domain") of Jext. Table 4.5 provides a description of the three understanding tasks (T1, T2 and T3) chosen in our study and shows their corresponding activities in [Pacione 2004]. Each of our tasks covers one or more activities, all activities are covered in our tasks except the activity A_8 which will be covered in the third research question.

2. *Participant Selection:*

   The persons involved in this experiment are seven Ph.D. students, who were not involved in this work before. These students were assigned to two groups (3 in each group) randomly. The role of the seventh participant is to evaluate tasks based on a response model. The first group *(SrcCode+CD)* used Jext source code and a class diagram recovered using the ObjectAid[9] tool to accomplish the understanding tasks. However, the second group *(SrcCode+OG)* used Jext source code and the object graph resulting from applying our approach on Jext. In order to ensure the equivalence between the two groups in terms of experience in OO programming, we asked the students to carry

[9] http://www.objectaid.com/

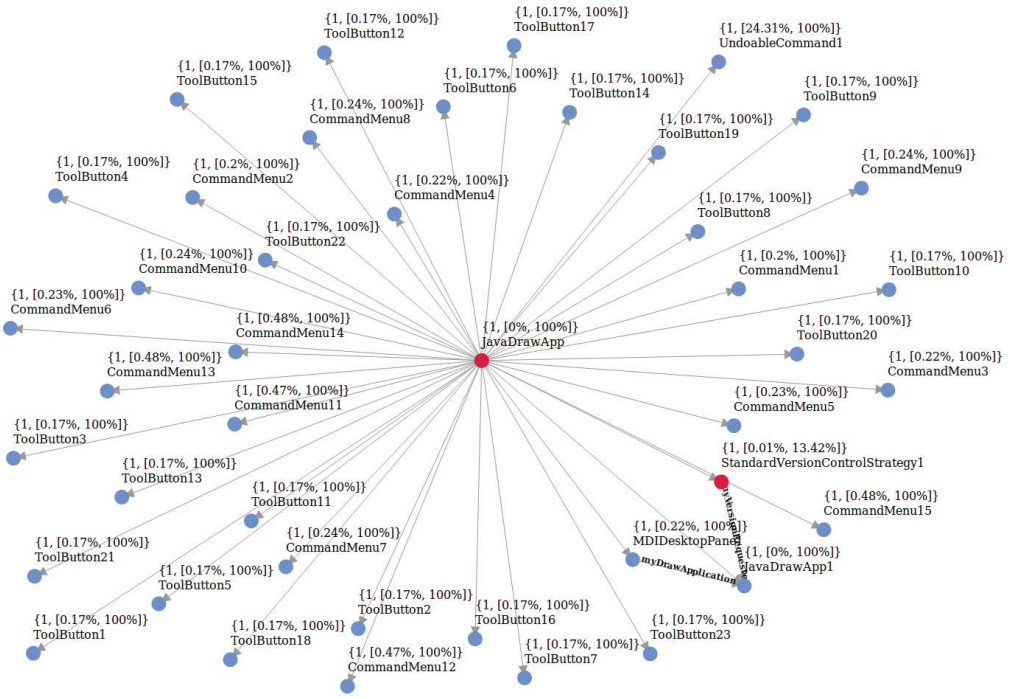

Figure 4.11: JHotDraw refined and hierarchical object graph with only composite structure exploited.

out a self-assessment on 5 points ranging from 1 (beginner) to 5 (advanced). The average experience in the *SrcCode+CD* group is 3 versus 3.33 in the *SrcCode+OG* group. Therefore, we conclude that the random assignment resulted in an almost equivalent experience between the two groups.

3. *Experimental Design:*

   The experiment was divided into two sessions: the *SrcCode+CD* group session and the *SrcCode+OG* group session. In the two sessions, the students were given a questionnaire containing the tasks described in Table 4.5. In addition, the subjects of *SrcCode+CD* group were given a pre-generated class diagram using the ObjectAid tool and were asked to use the tool, ObjectAid, for 15 minutes to be familiar with it. The subjects involved in the *SrcCode+OG* group session were given the OG and benefited from an explanation of 15 minutes of this graph. This familiarization time, 15 minutes for each group, is not considered in the total spent time in handling understanding tasks. In order to simulate real understanding and maintenance tasks, the students were not instructed to adhere to a limited timing.

4. *Correctness and Time Results*

Table 4.4: Principal software understanding activities

A_1 : Investigating the functionality of (a part of) the system
A_2 : Adding to or changing the system's functionality
A_3 : Investigating the internal structure of an artifact
A_4 : Investigating dependencies between artifacts
A_5 : Investigating the runtime interactions in the system
A_6 : Investigating how much an artifact is used
A_7 : Investigating patterns in the system
A_8 : Assessing the quality of the system's design
A_9 : Understanding the domain of the system

Table 4.5: Understanding tasks

ID	Activities	Description
T1	A_1, A_7 *and* A_9	What are the main stages in a typical Jext scenario? (formulate your answer from a high level perspective)
T2	A_3 *and* A_4	Name five text treatments/actions supported by Jext: which classes are responsible for these treatments? when were they created? who created them?
T3	A_1, A_2, A_5 *and* A_6	In general terms describe the life cycle of the org.jext.Mode class: when is it created? who created it? how many languages are supported in Jext?

Table 4.6 provides the results related to measurements of the time spent on tasks and of correctness of answers. From this table, we can note that the *SrcCode+OG* group required 36% less time. The *SrcCode+CD* group participants did not use the class diagram in answering the understanding tasks because of the noise caused by edges overlap in the class diagram. One participant described: *"it is difficult to follow the links between classes, if only the tool proposes a technique to focus on a subset of classes only, the task of following links will be easier"*. On the other side, the use of object graph contributes to reduce this time since some parts of some tasks can be directly identified from the graph. Therefore, the difference in time between the two groups is due to the fact that the *SrcCode+CD* group lost time in scrolling between source code files.

Concerning the results of correctness, we note that the answers given when using the object graph are more accurate averaging 5.2 out of 8.5 points compared to 4.8 points for the *SrcCode+CD* group which is rather good. This difference in correctness values between the two groups is due mainly to the third understanding task, more precisely to its last part: "how many languages

Group	Time Spent				Correctness			
	Min	Max	AVG	Diff	Min	Max	AVG	Diff
SrcCode+CD	66	148	98.7	-	4.25	6	4.8	-
SrcCode +OG	41	78	62.7	-36%	4.5	5.75	5.2	+ 8%

Table 4.6: Correctness (measured in points given to correct answers) and Time Spent (in minutes) results

are supported in *Jext*". The *SrcCode+CD* group participants looked at the answer in the *org.jext.Mode* whereas the answer lies in a properties file referenced in the *initModes* method of the *org.Jext* class. Whereas, the OG gives direct indications on the frequency of nodes.

4.3.3.3 RQ3

Antipatterns [Koenig 1998] are poor designs that can badly affect the system's quality, especially understandability. Because of their harmful effects, they should be detected in the code that must be refactored. To answer this question, we studied the impact of the availability of the refined and the hierarchical object graph on the detection of the "*Poltergeist*" antipattern [Brown 1998].
Poltergeists are classes with limited responsibilities and roles to play in the system; therefore, their effective life cycle is quite brief [Brown 1998]. This antipattern is also known by the name of *"Gypsy Wagons"* [Akroyd 1996] which are controller classes that exist only to recall methods of other classes.
This antipattern is selected because its identification is based on the lifetime property of the objects which is added, in our approach, as a label in the OG.
For Jext, the number of objects whose lifespan length (difference between the end timestamp and the start timestamp) is less than or equal to 20% in the recovered OG, equal to 37 objects. These objects are instances of 24 different classes. In the case of JHotDraw, the number of objects whose lifespan length is less than or equal to 20% is 16. These objects are instances of 10 different classes. These classes are candidates for the *Poltergeists* antipattern.
To validate this result, we consider an identification based on this antipattern symptoms reported in the literature. To the best of our knowledge, there exist only three works that focus on the detection and/or specification of the Poltergeist antipattern. Al-Rubaye [Al-Rubaye 2017] identified two symptoms of this antipattern namely cyclic association and short methods. A class is considered as a Poltergeist if it participates in a cyclic association and have low average line of code (LOC) value for its methods. The appropriate average length of methods can be identified based on the work of Lanza et al [Lanza 2007]. The authors of this work fixed the length to 7 for code written in Java.

Stoianov et al [Stoianov 2010] developed a tool (not publicly available) that detects a set of patterns and antipatterns among them the Poltergeist one. The authors defined rules in form of *Prolog* queries to describe the Poltergeist symptoms. Unfortunately, there is no explanation for these rules which makes the symptoms unclear. Llano et al [Llano 2009] give a UML specification of this antipattern in five forms introduced in [Riel 1996]. These forms are:

1. **Irrelevant classes:** these classes contain only accessor and/or print methods. They are characterized to be with no meaningful behavior in the design.

2. **Agent classes:** are classes that do not have instance variables and contain only agent methods. An agent method is a method that performs only one action, one call, which is passing a message from one class to another.

3. **Out of scope classes:** are classes that send messages to other classes but they never receive any message back, which means that the out of scope class methods are never called by other methods in the system.

4. **Operation classes:** an operation class contains only one method and all its instances are *transient* objects, the allocation site is preceded by the *transient* keyword in Java.

5. **Object classes:** are subclasses that do not add data or behavior different from the one of their super class.

Since the Poltergeist symptoms are not clear in the work of Stoianov et al [Stoianov 2010], our identification is based only on the works of Al-Rubaye [Al-Rubaye 2017] and Llano et al [Llano 2009].
In order to identify Poltergeists instances as specified by Al-Rubaye [Al-Rubaye 2017], we used *FindBugs*[10] tool in order to identify cyclic associations. Short methods are identified by counting the number of lines of code of the method of each class participating in a cyclic association. Poltergeists as defined by Llano et al [Llano 2009] are identified by a manual inspection of the source code.
Table 4.7 presents the results of *precision* and *recall* measures for the Poltergeist candidate identification. The first column reports precision and recall values when using only the Llano et al [Llano 2009] technique, the second column when using only Al-Rubaye [Al-Rubaye 2017] technique and the third column when using the two techniques, by making a union of the set of Poltergeseit candidates identified by the two techniques.
For Jext, precision values range from 0.46 using Llano et al [Llano 2009] to 0.71 by making the union of the two techniques. Recall values range from 0.46 by making the union of the two techniques to 0.65 by using Llano et al [Llano 2009] technique. In the case of JHotDraw, eight of these candidates were validated manually, using

[10]http://findbugs.sourceforge.net/

Table 4.7: Poltergeist detection results (TP for true positives, FP for false positives and FN for false negatives

System	Detection Method	[Llano 2009]	[Al-Rubaye 2017]	Union
Jext	TP	11	15	17
	FP	13	9	7
	FN	6	14	20
	Precision	0.46	0.62	0.71
	Recall	0.65	0.52	0.46
JHotDraw	TP	8	2	8
	FP	2	8	2
	FN	11	1	11
	Precision	0.8	0.2	0.8
	Recall	0.42	0.67	0.42

[Llano 2009] and [Al-Rubaye 2017] techniques, which gave a precision equal to 0.8. By using only the technique from [Al-Rubaye 2017], the recall value is equal to 0.67. However, by using only the technique from Llano et al [Llano 2009] or by combining the two techniques, the recall value becomes 0.42. For the two systems, precision and recall values are generally acceptable. We believe that this is due to the fact that some classes detected manually, using one technique or the two together, are never instantiated. That is why they are not automatically detected, using the recovered object graph.
We believe that our approach reduces the search space on Poltergeists by keeping only objects that verify the short-lived property. After that, the user can check which instances really represent Poltergeists.

4.3.4 Threats to Validity

4.3.4.1 Internal & construct validity

A potential threat concerns the extent to which the execution scenarios or test cases used to generate execution traces are representative of those actually used in practice. This threat could be mitigated if we experiment with more test cases by for example, giving the instrumented system to a number of students and ask them to use it for a period of time, and then collect the generated traces. We argue however that the selected scenarios used in the experiment reflect the main functionalities of the used systems.
Another threat concerns the use of the metric of "function coverage" to measure the coverage of the generated traces. With this metric, several traces can be generated randomly and only the ones that have a high value are used. However, it would be more practical to directly generate representative test cases. For that, we recently

started the research on more systematic ways to get representative test cases/scenarios. As a result, the approach used in the work of Delucia et al [Lucia 2018] seems to be adequate for our case. This approach is based on genetic algorithms in order to generate test cases for monitoring an instrumented code
An additional threat concerns the programming language's dynamic features, like reflection. Code that uses reflection is out of the bounds of our method. This code can reduce the accuracy of the static analysis phase. However, a large number of legacy systems do not use reflection. To determine the frequency of use of that mechanism (reflection), we implemented a parser which seeks for invocations to *newInstance* and *invoke* methods of the Java reflection API. We tested this parser on two randomly selected systems *JHotdraw*[11] and *Jitsi* [12]. We found that these method invocations represent only 0.2% for *JHotDraw* (13 out of 6375 allocation and invocation sites) and 0.24% for *Jitsi* (242 out of 99830). The small percentages for these two systems strengthen our intuition about the applicability of the described method.
Another threat is related to RQ2 where time spent and correctness results of understanding tasks are compared using an OG and a class diagram. The use of a class diagram makes the experiment somewhat unfair. A better comparison can be made using an object graph recovered using a state-of-the art publicly available method. However, class diagrams represent the most widely used diagrams to understand code structure, according to many practitioners and authors, like [Ammar 2012].
Another threat is related to the fact that the choice of tasks may have been biased to the advantage of the recovered OG. We alleviated this threat by reusing existing tasks proposed in the literature [Cornelissen 2009b], and adapting them to our particular system (Jext).

4.3.4.2 External validity

One threat concerns the use of one object system in this study. We believe that future experiments on a larger set of systems with larger sizes will yield more observations. However, since the results were positive with the studied (medium-sized) system; our intuition, on the interest of the refined and hierarchical OGs in assisting understanding, is strengthened.

4.4 Conclusion

The goal of this contribution is to support program understanding during software maintenance by recovering the as-implemented architecture. To this end, we proposed a six-step process to recover refined and hierarchical object graphs of object-oriented software systems. Compared to existing approaches for object graph recov-

[11] http://jhotdraw.org/
[12] https://jitsi.org/

ery, the graphs recovered by our approach have the following distinguishing features: i) nodes are labeled with lifespans and empirical probabilities of existence that enable a visualization with a level of detail; ii) they support the collapsing/expanding of objects to hide/show their internal structure. We reported an experiment where this process was applied to recover OGs for Java software systems and the results were in tune with our initial intuition.

Chapter 5

Identifying Modules and Services from the Source Code of Object-Oriented Software Systems

Contents

5.1 Introduction and Problem Statement

We proposed in the previous chapter an approach that improves the understandability quality attribute of object oriented legacy systems by recovering object graphs. The results of the experiment showed that the recovered OG contributes in reducing the complexity of the runtime architecture and in handling some comprehension tasks.
Another quality attribute in which we are interested is modularity. Improving modularity in object oriented class-based software systems is a challenging question. This improvement can be ensured by grouping source code artifacts into highly cohesive modules and reduce the coupling between these modules. Identifying highly cohesive modules in existing object oriented softwares is a challenging activity within the remodularization process. This consists in locating the coarse-grained software entities (groups of connected classes) that compose the system and which are related to performing a single task.
This chapter focuses on the third contribution of this thesis. We propose a process for identifying modules and services in an OO (class-based) software system based on the runtime architecture. We start our process by building an object graph, equivalent to the one presented in the previous chapter, in which nodes are objects and edges represent field-assigned references between objects. Then, we build the tree of instantiations, in which nodes are objects and an edge between two nodes means that the source node is the creator of the target node. In parallel to this, we identify the composition relationships between objects. Sometimes the creator of an object and its composite (owner, according to the definition from the previous chapter), if any, are not the same. Thus, the idea here consists in refining the extracted composition relationships by transforming the tree of instantiations in a way that each composite becomes the creator of its components. We do this, because we consider in our work that the composite structures reflect the design and the main structure of the software system. The goal of the refinement step is to increase cohesion between the composite and its components, since the composite of an object will be the one which creates it. An intermediate output of the composition refinement step is a set of tree edit operations on the tree of instantiations. This set of edit operations is then manually translated into refactoring operations applied on the classes, from which originated the modeled objects. We believe that source code artifacts with composition relationships should be clustered together in the same module since the separation of these artifacts into different modules will result in high coupling between modules. For that, the next step of our approach consists of applying a composition conservative genetic algorithm on the refactored code in order to identify modules. Once the modules have been identified and in order to

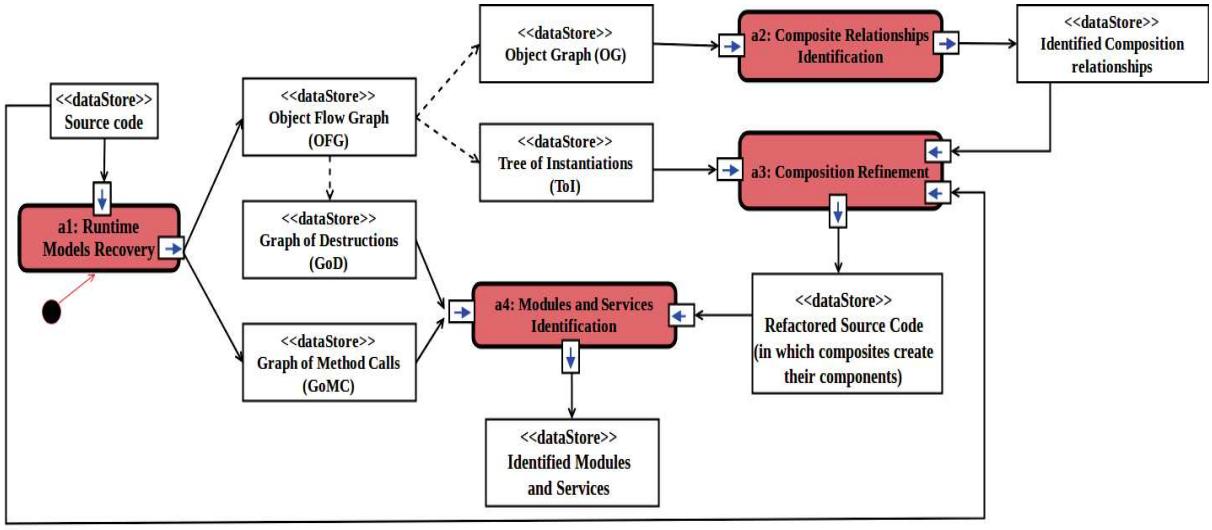

Figure 5.1: Modules and services identification Process

allow further decoupling, the last step of the process consists of identifying each module services, functionalities that are provided by a module to other modules. In order to reach this goal, two additional graphs are built: the graph of object destructions and the graph of method calls.
The rest of this chapter is organized as follows. Sections 5.2.1 to 5.2.4 detail each step in this process. Section 5.3 exposes an experimentation of the process which was conducted on real-world Java projects. The goal of this experimentation is to measure the gain in modularity brought by the process. Conclusion and perspectives are presented in Section 5.4.

5.2 Foundations of the proposed Approach

The overall process is depicted in Figure 5.1. The input of this process is the source code of an OO software system. This process produces many intermediate models (graphs and trees) to produce at the end a set of modules with their identified provided services. These modules can then be deployed and easily reused because they embed all their dependent classes, at the software system structure level. This process is composed of the following steps:

1. **Runtime Models Recovery (a1):** the goal of this step is to build five kinds of graphs based on a static analysis of the source code of the software system given as input. These graphs are: the object flow graph (OFG), which tracks objects from their creation until their (reference) storage in fields or usage in method invocations, the tree of instantiations (ToI), which models the objects of the system (o_1, o_2, ...) and their relationships of kind "o_i *creates* o_j", the object graph (OG), which represents objects and their relationships of kind

"o_i *stores in one of its slots/fields a reference to* o_j", the graph of destructions (GoD) which represents objects and their relationships of kind "o_i *destructs* o_j" and the graph of method calls (GoMC) which represents objects and their relationships of kind "o_i *invokes methods of* o_j"

2. **Composition Relationships Identification (a2):** in this step, the object graph OG is used in order to infer the composition relationships between classes. This is done using the approach proposed by Milanova et al [Liu 2007, Milanova 2007].

3. **Composition Refinement (a3):** in this step the identified composition relationships are refined. This means that the creators and composites of objects are compared in order to identify a set of tree edit operations. These edit operations are necessary to make the composite of an object the one who creates it. The identified tree edit operations are translated into refactoring operations depending on how instantiations are made in the source code.

4. **Modules and services identification (a4):** in this step, classes are grouped into modules using a genetic-algorithm based clustering. Moreover, services that represent a way of communication between these modules are identified.

In the following, each step of this process is detailed.

5.2.1 Runtime Models Recovery

The purpose of this first step is to build the aforementioned types of graphs.
The OG is recovered using the same method described in Section 4.2.2 of Chapter 4. In the same way, the ToI is generated by analyzing the creators of OFG's objects. The creator of an object is the *object identifier/class name* of the first program variable, in the OFG, to which this object was assigned, directly after its creation. For example, objects creators for the example in Listing 5.1 (objects are highlighted in crimson) are presented in Table 5.1.

Listing 5.1: OFG edges

```
MovieCatalog.main.movieCat=MovieCatalog1.MovieCatalog.this
MovieCatalog1.MovieCatalog.movieM=MovieModel1.MovieModel.this
MovieCatalog1.setModel.m=MovieCatalog1.MovieCatalog.movieM
MovieCatalog1.theModel=MovieCatalog1.setModel.m
MovieCatalog1.mainView=MainView1.MainView.this
MovieCatalog1.listView=MovieListView1.MovieListView.this
MovieCatalog1.itemView=MovieItemView1.MovieItemView.this
MovieCatalog1.showApp.this=MovieCatalog.main.movieCat
```

The other types of graphs are explained in Section 5.2.4.2.

Table 5.1: Objects creators of the *MovieCatalog* class example

Object	MovieCatalog1	MovieModel1	MainView1	MovieListView1
Creator	MovieCatalog	MovieCatalog1	MovieCatalog1	MovieCatalog1

Object	MovieItemView1
Creator	MovieCatalog1

5.2.2 Composition Relationships Identification

This second step enables to recover the "UML-like" composition relationships between classes using the OG. To that end, the approach proposed in [Milanova 2007] is used. This approach is based on the owners-as-dominators ownership model. Our choice for this approach is motivated by the fact that it is precise and enables inferring a high percentage of composition relationships since the used ownership model, owners-as-dominators, captures well the notion of composition [Milanova 2007].
The steps of the approach in [Milanova 2007] can be summarized in: the *ownership analysis* and the *composition inference*.
Ownership analysis step consists of calculating the boundary of each object in the OG. The boundary set of an object o consists of a subgraph rooted at o and which contains all objects that are dominated by o. The ownership analysis is applied using Algorithms 3 and 4 and can be summarized as follows:

1. For each node o in the *OG*, the set of all the reachable edges from this node is calculated.

2. For each reachable edge, two sets are calculated: the *closure* set and the *parent* set of the closure. The goal of the parent set is to ensure that the access to a given node o_j reachable from o is within the boundary of o.

3. Closures are classified into valid or non-valid. Non-valid closures refers to the situation when a given object o_j reachable from o is accessed from outside without passing through o.

4. Valid closures whose parent set is empty are added into the boundary set of the node o.

Once the boundary set of each object has been identified, a composition relationship is inferred for each edge in the boundary set of o whose source node is o. This composition relationships is identified between the class of o and the class of the target node in the edge.
In order to have clear insights on the composition relationships identification approach, Algorithm 4 is applied on the example shown in Figure 5.2. The closure and parent sets and the classification, to valid or not valid, of closures are given in Table 5.2. As an example, consider the node E_1. For this node, there are four valid closures: Closure($E_1 \rightarrow F_1$)=$\{E_1 \rightarrow F_1\}$, Closure($E_1 \rightarrow H_1$)=$\{E_1 \rightarrow H_1\}$, Closure(E_1

Algorithm 3 Closure and parent sets calculation algorithm

Input: An edge $o_i \rightarrow o_j$ reachable from o
Output: $Cl \leftarrow$ *Closure*$(o_i \rightarrow o_j)$ *and Parent(Cl)*

```
1: W ← ∅, Cl ← ∅, Parent(Cl) ← ∅
2: mark o_i → o_j, add o_i → o_j to W and to Cl
3: while W not empty do
4:    remove o_i → o_j from W
5:    for o_k → o_j such that o_k → o_i and o_k is reachable from o do
6:       if o_k → o_j is unmarked then
7:          mark o_k → o_j, add it to W and Cl
8:          add o_k → o_i to parent(Cl)
9:       end if
10:   end for
11:   for o_k → o_j such that o_i → o_k do
12:      if o_k → o_j is unmarked then
13:         mark o_k → o_j, add it to W and Cl
14:         add o_i → o_k to parent(Cl)
15:      end if
16:   end for
17: end while
```

Algorithm 4 Boundary sets calculation algorithm

Input: A node o in the OG
Output: *Boundary(o)*

```
1: for unmarked edge o_i → o_j reachable from o do
2:    Calculate closure set of o_i → o_j using Algorithm 3
3: end for
4: for edge o → o_j such that ∃ o_k such that o_k → o and o_k → o_j do
5:    Mark the closure set of o → o_j as non valid
6: end for
7: while empty Parent(Closure(o_i → o_j)) and valid Closure(o_i → o_j) do
8:    add Closure(o_i → o_j) to Boundary(o)
9:    for e ∈ Closure(o_i → o_j) do
10:      remove e from each Parent set
11:   end for
12: end while
```

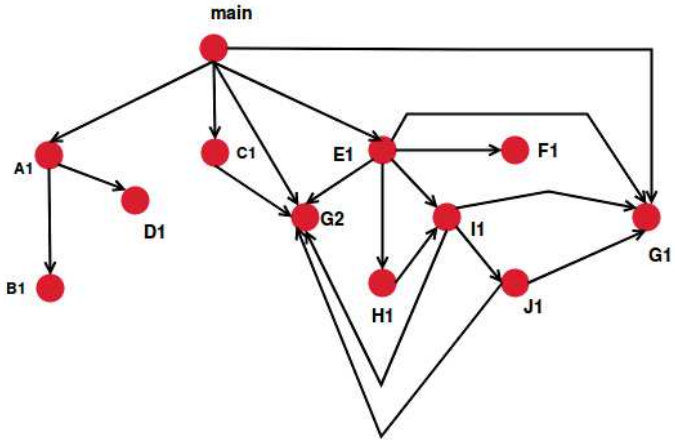

Figure 5.2: An OG example

$\rightarrow$ I_1)={$E_1 \rightarrow I_1$, $H_1 \rightarrow I_1$} and Closure($I_1 \rightarrow J_1$)={$I_1 \rightarrow J_1$}. Parent sets for these closures are respectively: { }, { }, {$E_1 \rightarrow H_1$} and { }. The algorithm 4 adds the edge $E_1 \rightarrow F_1$ to the boundary of the node E_1. After that, it adds the edge $E_1 \rightarrow H_1$ to the boundary of E_1 and deletes it from the third parent set. The third parent set becomes empty and edges $E_1 \rightarrow I_1$ and $H_1 \rightarrow I_1$ are added to the boundary of E_1. Finally, the edge $I_1 \rightarrow J_1$ is added to the boundary. The boundary sets for each node in the object graph in Figure 5.2 are given in Table 5.3.
Using results from Table 5.3, the extracted composition relationships between classes are shown in Figure 5.3.

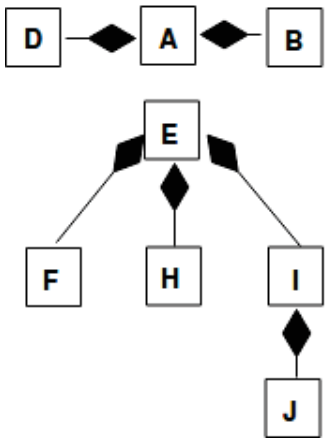

Figure 5.3: Composition relationships between classes

Once the composition relationships between classes have been identified, they will be refined in the following step.

Table 5.2: Closure and parent sets for the OG in Figure 5.2

OG nodes	Reachable edges	Closure sets	Parent sets	Non valid closure
A_1	$A_1 \rightarrow B_1$	$\{A_1 \rightarrow B_1\}$	{}	
	$A_1 \rightarrow D_1$	$\{A_1 \rightarrow D_1\}$	{}	
B_1	-	-	-	-
C_1	$C_1 \rightarrow G_2$	$\{C_1 \rightarrow G_2\}$	{}	X
D_1	-	-	-	-
E_1	$E_1 \rightarrow F_1$	$\{E_1 \rightarrow F_1\}$	{}	
	$E_1 \rightarrow H1$	$\{E_1 \rightarrow H1\}$	{}	
	$E_1 \rightarrow I_1$	$\{E_1 \rightarrow I_1, H1 \rightarrow I_1\}$	$\{E_1 \rightarrow H1\}$	
	$E_1 \rightarrow G_1$	$\{E_1 \rightarrow G_1, I_1 \rightarrow G_1, J_1 \rightarrow G_1\}$	$\{E_1 \rightarrow I_1, I_1 \rightarrow J_1\}$	X
	$E_1 \rightarrow G_2$	$\{E_1 \rightarrow G_2, I_1 \rightarrow G_2, J_1 \rightarrow G_2\}$	$\{E_1 \rightarrow I_1, I_1 \rightarrow J_1\}$	X
	$H1 \rightarrow I_1$	Already marked	-	-
	$I_1 \rightarrow J_1$	$\{I_1 \rightarrow J_1\}$	{}	
	$I_1 \rightarrow G_1$	Already marked	-	-
	$I_1 \rightarrow G_2$	Already marked	-	-
	$J_1 \rightarrow G_1$	Already marked	-	-
	$J_1 \rightarrow G_2$	Already marked	-	-
F_1	-	-	-	-
G_1	-	-	-	-
G_2	-	-	-	-
H1	$H_1 \rightarrow I_1$	$\{H_1 \rightarrow I_1, E_1 \rightarrow I_1\}$	$\{E_1 \rightarrow H_1\}$	X
	$I_1 \rightarrow J_1$	$\{I_1 \rightarrow J_1\}$	{}	
	$I_1 \rightarrow G_1$	$\{I_1 \rightarrow G_1\}$	{}	X
	$I_1 \rightarrow G_2$	$\{I_1 \rightarrow G_2\}$	{}	X
	$J_1 \rightarrow G_1$	$\{J_1 \rightarrow G_1\}$	{}	X
	$J_1 \rightarrow G_2$	$\{J_1 \rightarrow G_2\}$	{}	X
I_1	$I_1 \rightarrow J_1$	$\{I_1 \rightarrow J_1\}$	{}	
	$I_1 \rightarrow G_1$	$\{I_1 \rightarrow G_1\}$	{}	X
	$I_1 \rightarrow G_2$	$\{I_1 \rightarrow G_2\}$	{}	X
	$J_1 \rightarrow G_1$	$\{J_1 \rightarrow G_1\}$	{}	X
	$J_1 \rightarrow G_2$	$\{J_1 \rightarrow G_2\}$	{}	X
J_1	$J_1 \rightarrow G_1$	$\{J_1 \rightarrow G_1\}$	{}	X
	$J_1 \rightarrow G_2$	$\{J_1 \rightarrow G_2\}$	{}	X

5.2.3 Composition Refinement

We consider in our work the composite structure as the main structure of the software system (a model of objects and their fields) and its runtime architecture. For this reason, we believe that a composite class and its component classes should be grouped in the same module.
We believe that a composition relationship presented in the OG by, for example, $A_1 \rightarrow B_1$ requires that the class A creates the object B_1 as stated in [Seemann 1998]. What we have noticed by analyzing many OO systems, is that the composite is not always the creator of its components. We believe that editing the ToI in a way that each composite becomes the creator of its components contributes in increasing cohesion between composite and its component classes and, thus, decouples it from other classes. This is justified by the fact that an additional type of structural relationships, creation relationship, will be added between the composite and the component classes.

Table 5.3: Boundary sets of the nodes of the OG in Figure 5.2

OG nodes	Boundary Sets
$\mathbf{A}_1$	$\{A_1 \rightarrow B_1, A_1 \rightarrow D_1\}$
$\mathbf{B}_1$	-
$\mathbf{C}_1$	-
$\mathbf{D}_1$	-
$\mathbf{E}_1$	$\{E_1 \rightarrow F_1, E_1 \rightarrow H_1, E_1 \rightarrow I_1, H_1 \rightarrow I_1, I_1 \rightarrow J_1\}$
$\mathbf{F}_1$	-
$\mathbf{G}_1$	-
$\mathbf{G}_2$	-
$\mathbf{H}_1$	-
$\mathbf{I}_1$	$I_1 \rightarrow J_1$
$\mathbf{J}_1$	-

Tree edit operations are generated by comparing the ToI with regards to the extracted composition relationships in the previous step. A tree edit operation corresponds to replacing the parent of a node in the ToI, by its composite, if any, extracted in the previous step. At the source code level, these tree edit operations are translated into refactoring operations. These refactoring operations consist in moving allocation sites from a source class to a target one.
An example of this refinement step is given in Figure 5.4. The corresponding source code of the *ToI (a)* in Figure 5.4 is given in Listing 5.2. In the composition identification step, a composition relationship was extracted between ToolButton_1 and ZoomTool_1 ($\text{ToolButton}_1 \rightarrow \text{ZoomTool}_1$).
As it is shown in the *ToI (a)* in Figure 5.4, the creator of ZoomTool_1, JavaDrawApp_1, is not its composite ToolButton_1. Therefore, a tree edit operation should be applied on the ToI. This edit operation consists of removing the link between JavaDrawApp_1 and ZoomTool_1 and replace it by a link between ToolButton_1 and ZoomTool_1 (as done in *(b)*). At the source code level, this corresponds to removing the allocation site of the object whose identifier is ZoomTool_1 from the class *JavaDrawApp* and put it in the class *ToolButton* as shown in Listing 5.3. In this way, *JavaDrawApp* and *ZoomTool* have been decoupled and *ToolButton* and *ZoomTool* have been coupled further.

Listing 5.2: Source code before applying composition refinement

```
public  class JavaDrawApp {

protected void createTools(...) {
        ...
        ToolButton tb = createToolButton(..., new ZoomTool());
        ...
    }

protected ToolButton createToolButton(..., Tool tool) {

        ToolButton toolButton = new ToolButton(..., tool);;
        return toolButton;
    }

}
```

```
16
public class ToolButton {
18
Tool myTool;
20
public ToolButton(..., Tool tool)
22 {  ...
   setTool(tool);
24 ...
}
26
private void setTool(Tool newTool) {
28        myTool = newTool;
    }
30 }
```

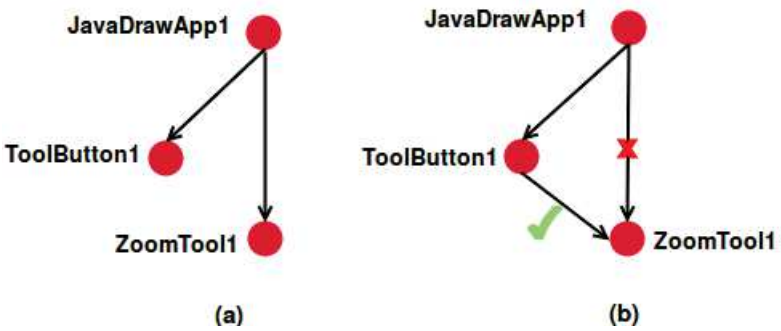

Figure 5.4: Composition refinement step example

Listing 5.3: Source code after applying composition refinement

```
public  class JavaDrawApp {
2
protected void createTools(...) {
4       ...
        ToolButton tb = createToolButton(...,
6                       ToolButton.createZoomTool());
        ...
8   }

10 protected ToolButton createToolButton(..., Tool tool) {

12      ToolButton toolButton = new ToolButton(..., tool);;
        return toolButton;
14  }

16 }

18 public class ToolButton {

20 Tool myTool;

22 public ToolButton(..., Tool tool)
24 {  ...
   setTool(tool);
   ...
26 }

28 private void setTool(Tool newTool) {
```

```
            myTool = newTool;
30      }

32  public static ZoomTool createZoomTool() {
        return new ZoomTool();
34      }
}
```

Once the composition relationships have been refined, the following step consists of identifying modules based on these identified and refined relationships.

5.2.4 Module and Service Identification

5.2.4.1 Module Identification

We define a module as a group of classes/interfaces. These classes/interfaces are highly cohesive with each other and lowly coupled with classes/interfaces of other modules. In our case, module identification is formulated as a search-based optimization problem. That is, an optimized partition of classes/interfaces on modules is searched using a genetic algorithm based on a clustering criteria (fitness function). Genetic algorithms have been found to be an effective clustering technique of large scale object oriented software systems [Islam 2018]. In the remaining of this section, the used fitness function and the genetic algorithm are described.

1. *The Used Fitness Function:*

   The determination of a fitness function/grouping criteria is a critical issue since it has a direct impact on the obtained solution (clustering result). To evaluate a clustering solution, an aggregated fitness function is used. This function is based on the maximization of both the modularity and the organization of modules values. This fitness function is defined as follows:

$$F = M^{\alpha} * O^{\beta} \tag{5.1}$$

   Where F corresponds to the fitness function, M to the modularity value and O to the module organization value of a given clustering solution. α and β are user-defined parameters $\in$ [0, 1] that define the importance of M and O according to the user viewpoint. In this work, we do not prefer M over O or the inverse. We only discuss the case where $\alpha = \beta = 1$. Product is chosen for the aggregated fitness function, instead of addition, to avoid the case when a clustering solution has a high total fitness value but low scores for M or O. For example, if a clustering solution has an M value equal to 0.8 and an O value equal to 0.1. If the addition is used in the aggregated fitness function, the value of F is equal to 0.9 (0.8 + 0.1) which is a high score. However, the module organization (O) is quite poor. In another solution, M is equal to 0.5 and O is equal to 0.4 so the F value is equal to 0.9. In this case, there is no

difference between the first and the second solution. However, if the product is used in the aggregated fitness function, the second solution is preferred over the first one ($0.2 > 0.08$). Therefore, using the product is a way of penalizing clustering solutions that have low scores of M or O.

As described earlier, low coupling and high cohesion are indicators of good modularity. For this reason, M value is calculated in terms of coupling and cohesion using the following formula as in [Allier 2011]:

$$M = \frac{SumCoh}{SumCoh + SumCoup} \tag{5.2}$$

Where $SumCoh$ is the cohesion sum of all modules cohesion values in a given clustering solution and $SumCoup$ is the coupling sum of all modules. M values are $\in [0, 1]$. The higher the value of M is, the more modular the system is. The cohesion and the coupling of a given module m composed of n classes/interfaces are calculated using the following formulas:

$$\begin{aligned} Cohesion_m &= \sum_{i=1}^{n} \sum_{j \in m} SCS_{WTFIDF}(c_i, c_j) \\ Coupling_m &= \sum_{i=1}^{n} \sum_{j \notin m} SCS_{WTFIDF}(c_i, c_j) \end{aligned} \tag{5.3}$$

Cohesion and coupling calculation is based on a a structural coupling measure SCS_{WTFIDF}, proposed by Prajapati et al [Prajapati 2017]. This coupling measure is defined in terms of eight types of structural coupling relationships that can exist between two classes: *Extends(EX), Has parameter(HP), References(RE), Calls(CA), Implements(IM), Is of Type(IT), Returns(RT) and Throws(TH).* Weights of these relationships were defined, in [Prajapati 2017], using Apache Solr[1] and Apache Tomcat[2] systems which are considered in the literature to be of good quality [Prajapati 2017]. Table 5.4 shows the weight values for each relationship. In our case, since we do not make a difference between the cases when a class creates an instance of another class to reference its fields, *RE*, or to call its methods, *CA*, the used weight values are those of *Apache Tomcat.*

In fact, the authors proposed eight different Structural Coupling Schemes (SCS). The one used in our study is the *Weighted Term Frequency Inverse Document Frequency (WTFIDF)* coupling scheme which is calculated using the following formula:

[1]http://lucene.apache.org/solr/
[2]http://tomcat.apache.org/

Table 5.4: Weight values for the eight structural relationships.

	EX	HP	RE	CA	IM	IT	RT	TH
Apache Solr	9	4	3	2	2	2	1	1
Apache Tomcat	8	3	3	3	2	2	1	1

$$SCS_{WTFIDF}(c_i, c_j) = \frac{\sum_{r \in R} w_r * n_r(c_i, c_j)}{\sum_{k=1}^{|C|} \sum_{r \in R} w_r * n_r(c_k, c_j)} log(\frac{|C|}{n_{c_j}}) + \frac{\sum_{r \in R} w_r * n_r(c_j, c_i)}{\sum_{k=1}^{|C|} \sum_{r \in R} w_r * n_r(c_k, c_i)} log(\frac{|C|}{n_{c_i}}) \quad (5.4)$$

Where: c_i and c_j are two given classes, R denotes the set of relationship types that exist between c_i and c_j, w_r the weight of the relationship r, $n_r(c_i, c_j)$ denotes the number of instances of r-type relationship from class c_i to class c_j, $|C|$ is the number of classes in the system and n_{c_i} is the number of classes that are related to class c_i.

Our choice for this coupling scheme is motivated by the fact that it outperforms all the other schemes [Prajapati 2017], defined by the same authors, since it integrates various additional aspects such as weights, the total number of classes structurally connected to a class, and the total number of classes present in the system.

Formula 5.1 is based on two measures M and O. Up to now, M value calculation was explained above. Explanations on O value calculation are given in the following.

Module organization *(O)* is taken into account in the fitness function in order to tackle the problem when no system partitioning is produced. That is, when only modularity value, M, is taken into account, the best solution is the one where all classes/interfaces reside in just one module. This is clearly not the goal of clustering techniques, as no system partitioning is done.

Module organization *(O)* is evaluated using the metric proposed by [Bouwers 2011], and which has been considered in several recent works such as the ones from Ramirez et al. [Ramírez 2016] and Ernst et al. [Ernst 2017]. In [Bouwers 2011], O is the product of two other metrics, namely *System Breakdown (SB)* which is based on the number of system modules and *Component Size Uniformity (CSU)* which is based on the *Gini coefficient*, i.e. a statistical measure of dispersion, of modules *volumes*, i.e. the size of its classes. O values are between 0 and 1. Higher values represent a better module organization. *SB, CSU* and O values are calculated using the formulas 5.5, 5.6 and 5.7 respectively.

$$SB(|M|) \begin{cases} \frac{|M|-1}{x-1} & if \quad |M| \leq x \\ 1 - \frac{|M|-x}{y-x} & if \quad x < |M| < y \\ 0 & if \quad |M| > y \end{cases} \tag{5.5}$$

$$CSU(M) = 1 - Gini(volume(m) : m \in M) \tag{5.6}$$

$$O = SB(|M|) * CSU(M) \tag{5.7}$$

Where m is a module from a given solution and *volume(m)* is the number of types in the module m. $|M|$ is the number of modules in a clustering solution. x and y correspond to the minimal and maximal numbers of modules given by the user. In our case, x and y are fixed to 8 and 16 respectively as in [Bouwers 2011][3].

2. *The Used Genetic Clustering Algorithm:*

   The general functioning of a genetic algorithm is presented in Figure 5.5. For every iteration of the algorithm, a new population is generated by applying selection, crossover and mutation operators on the current population. When the termination criteria is reached (e.g., given maximal number of iterations is reached), the best solution found is returned.

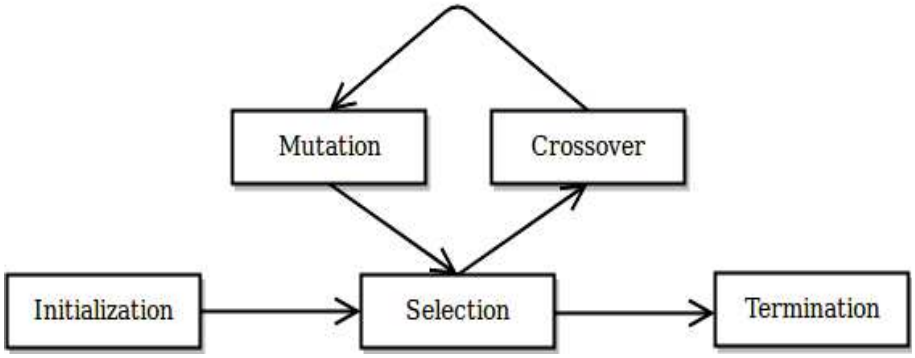

Figure 5.5: Genetic algorithms basic steps

Existing genetic algorithms are differentiated at four aspects: 1) solution representation, 2) population initialization, 3) genetic operators (selection, crossover and mutation), and 4) the fitness function used to evaluate solutions. The fourth aspect, the used fitness function, was already explained. The remaining aspects of the used genetic algorithm are explained in the following:

[3]in this work, x and y values were fixed based on the most common number of modules in a benchmark of 172 software systems.

- *Solution representation*

  A clustering solution is represented by an array of a length equal to the number of classes/interfaces of the system to be remodularized. The array cells contain numbers which correspond to module identifiers. An example of a candidate solution is shown in Figure 5.6.

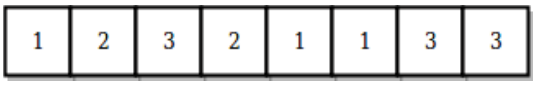

Figure 5.6: An example of a candidate solution

  In Figure 5.6, the number of classes/interfaces of the input system is 8 and the number of modules in the solution is 3. The 1st, 5th and 6th classes/interfaces (from left to right) belong to module 1. Similarly, the 2nd and 4th classes/interfaces are in module 2, and 3rd, 7th and 8th are in module 3.

- *Initial Population*

  In general, genetic algorithms start by an initial set of solutions. This set is called initial population. Most of the time, initial population is generated randomly. In our case, composition relationships identified in the second step of the process (Section 5.2.2) are taken into account in the initial population. That is, if a composition relationship exists between two classes, they must be grouped in the same module and the genetic algorithm operators (presented in the following) must not separate these classes.

  As an example, consider a software system composed of 14 classes/interfaces for which the *OG* in Figure 5.2 is recovered. An individual from the initial population of this system is depicted in Figure 5.7. In this individual, classes *A, B* and *D* are grouped in the same module (0) because there is a composition relationship between the class *A* and the classes *B* and *D* as shown in Figure 5.3. Also, classes *E, F, H, I,* and *J* are grouped in the same module (1). Modules of the remaining classes *(C, G, K, L, M and N)* are generated randomly.

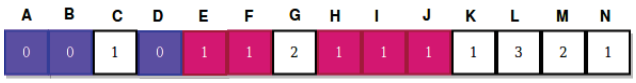

Figure 5.7: Composition relationships taken into account in the initial population

- *Selection Operator*

Selection consists of choosing solutions from a population as parents that will create an offspring for the next population. An offspring is an individual created by combining information from two individuals. There exists several selection operators [Mitchell 1998] such as: Tournament, Rank and Roulette-Wheel selection.

In this work, the Roulette-Wheel selection operator is used because of its simplicity. The principle is simple: each solution of the population is assigned a portion of a circular roulette wheel according to its fitness value. Then, a random selection is done similar to how the roulette wheel is rotated.

- *Crossover Operator*

  Crossover consists of producing new offspring solutions by combining two parent solutions. There exists different types of crossover operators [Mitchell 1998] such as: single-point crossover and heuristic crossover. In this work, the single-point crossover is used because of its simplicity. It consists of selecting randomly a crossover point. Cells to the right of that point are exchanged between the two parent solutions. This results in two offspring solutions. A simple example is shown in Figure 5.8. In this Figure, point between the 7th and the 8th classes/interfaces is selected to be the crossover point.

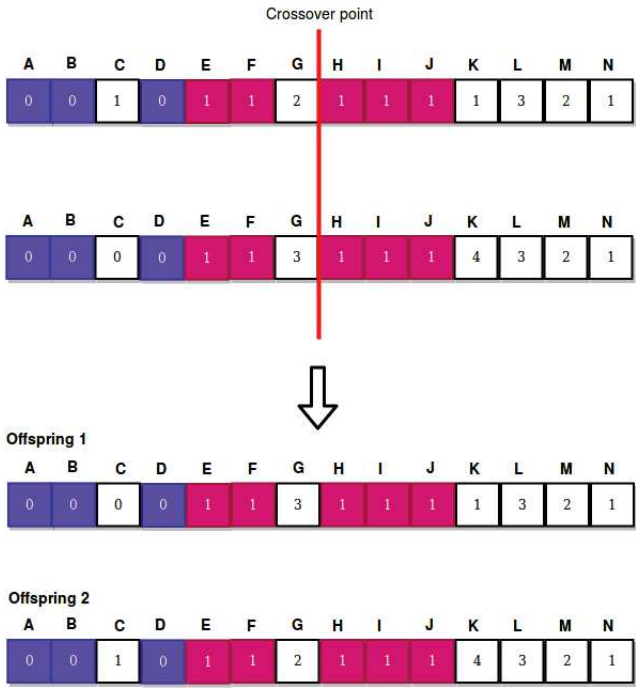

Figure 5.8: An example of single-point crossover result

- *Mutation Operator*

Mutation is the process of alternating classes/interfaces between modules within one solution to obtain a new solution. That is, module numbers of some classes/interfaces of a selected solution is changed to new module numbers. Generally, mutation process is done randomly. In our case, the used mutation operator is based on the KMeans algorithm since it has improved the clustering in many existing works [Cheng 2006]. The KMeans algorithm is based on the SCS_{WTFIDF} values between each pair of classes/interfaces as a distance measure. Classes with composition relationships between each others are not considered by the mutation. That is, they are not moved from their initial modules. An example of mutation operator is given in Figure 5.9. In this example, the 3rd class/interface was in module 1 and it is moved by the mutation operator to module 0, the 7th class/interface is moved to module 1, 11th class/interface is moved to module 2, 12th class/interface is moved to module 1, 13th class/interface is moved to module 2 and 14th class to module 0.

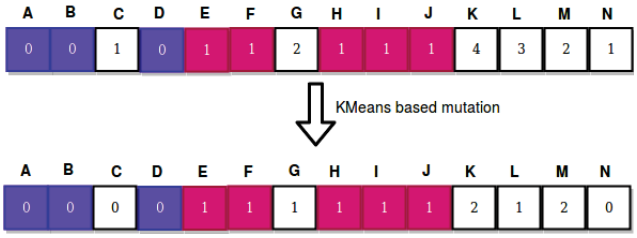

Figure 5.9: An example of KMeans based mutation

At the end of this step, modules of the OO software system are identified. The identified modules are characterized to be composition relationships conservative. In order to further decouple the identified modules, provided services of each module are identified in the following step.

5.2.4.2 Service Identification

Service identification is a crucial phase in the process of legacy to *SOA* migration. This identification is based on the service definition. In the literature, many definitions have been proposed for defining services [Brown 2002, Nakamura 2009, Erradi 2006]. Most of these definitions describe a service based on different properties (e.g. granularity and self-containment). In our work, we define a service as an object (class instance) that is registered in a service registry by a provider and can be looked up by one or several clients, as described by other authors like [Gruber 2005, Tavares 2008]. In general, services should be favored over inter-module class dependencies [McAffer 2010].

In order to identify service characteristics, we analyzed manually the source code of two object oriented software systems: Eclipse[4] and Jitsi[5]. These are well known service-oriented applications, adopters of OSGi Java framework in which a service corresponds to a standard Java object. Table 5.5 provides a brief description of these projects which are of different sizes, varying from 3218 to 12360 types (classes and interfaces). This table shows also the number of the registered services in each project. Roughly speaking, service registries and service references are collected based on an analysis of the call sites that correspond to the *registerService* and *getServiceReference(s)* OSGi method invocations respectively.

Table 5.5: Eclipse and Jitsi services

System	#Types	#Bundles	#Registered services
Eclipse 3.1	12360	171	10
Jitsi	3218	183	228

We summarize below the most important observations/properties made during the analysis of these systems and which we consider in our service identification process:

1. *Property 01:* bundles/modules register/create and unregister/destroy services. The bundle/module which registered a given service is the one to unregister it.

2. *Property 02:* every registered service has a unique *ServiceRegistration* object, for the private use of the service inside its bundle/module, and has one or more *ServiceReference* objects that refer to it to be used outside its bundle/module.

Based on these properties, we defined the following conditions that a given object must verify in order to be considered as a service:

1. **Condition 01** *(using property 01):* the creator/registrant and destructor/unregistrant of this object belongs to its module.

2. **Condition 02** *(using property 02):* the object has at least one client, an object that uses it, outside its module.

Service identification requires other types of information that does not exist yet in the recovered graphs. For this reason, we need two additional types of graphs: graph of destructions (GoD) and graph of method calls (GoMC).
GoD is represented by a pair (O, E_{GoD}) where O represents the set of the system running objects. These objects are the same which exist in OG. Each edge $e \in E_{GoD}$ between o_1 and o_2 means that o_1 is the destructor of o_2. The destruction relationship is captured using Algorithm 5. The input of this algorithm is the object flow graph edges and its output is the E_{GoD} set.

[4]We analyzed version 3.1 available in: http://archive.eclipse.org/eclipse/downloads/. It represents the first version of Eclipse that adopts the notions of modules (bundles) and services.
[5]https://github.com/jitsi

Algorithm 5 Destruction relationship identification

Input: OFG edges
Output: E_{GoD}

```
1:  E_GoD ← ∅
2:  for o ∈ O_GoD do
3:     Ref_o ← o.getReferences()
4:     Scopes ← ∅
5:     i ← 0
6:     for r ∈ Ref_o do
7:        assign_r ← r.getAssignment()
8:        if assign_r.getRightHandSide() = null ∨ assign_r.getRightHandSide() ∉ Ref_o
          then
9:           Scopes ← Scopes ∪ assign_r.getScope()
10:          i ← i + 1
11:       end if
12:    end for
13:    if i = Ref_o.size() then
14:       E_GoD ← E_GoD ∪ (Scopes.getLast(), o)
15:    else
16:       E_GoD ← E_GoD ∪ (o.getCreator(), o)
17:    end if
18: end for
```

Destruction relationship identification is not a trivial task since an object can have several references, which is known by the *aliasing problem* in the literature [Clarke 2013]. In this case, an object is destroyed if *null* or a reference to a new object are assigned to all variables that reference it.
In order to identify destruction relationships, Algorithm 5[6] collects all the references to each object running in the system (Line 3). Then, for each reference r, the algorithm gets the first assignment in which r is in the left hand side (Line 7). If the right hand side of this assignment is equal to null or to another reference different from the ones of the current object, the algorithm adds the scope of the assignment to the set of scopes (Line 9). We mean here by scope the object of the class where the assignment statement exists. After that, the algorithm tests if all the references were reassigned (Line 12). In this case, the scope of the assignment of the last reference represents the destructor of the current object (Line 13), otherwise, we consider that the creator and the destructor of the current object are the same (Line 15).

[6]It is a kind of the marking step of the garbage collector which allows classifying objects into usable or unusable

GoMC is represented by a pair (O, E_{GoMC}). *O* represents the set of objects running in the system. Each edge $e \in E_{GoMC}$ between o_1 and o_2 means that o_1 calls methods of o_2. Several algorithms have been proposed in order to recover call graphs of object oriented software systems. Examples of these algorithms are:

- **Reachability Analysis (RA):** is the first proposed algorithm for call graphs recovery. This algorithm starts the analysis from the `main` method and for each call site in a visited method, an edge is added, to the set of the call graph edges, between the visited method and each method having the same name as the one in the call site. Taking only the method names into account, no parameters and/or return types, makes the algorithm not precise since some edges between methods are not really used.
- **Class Hierarchy Analysis (CHA) [Dean 1995]:** is an extended version of the *RA* algorithm. However, the results are more precise than RA since it takes class hierarchy information into account. CHA looks at the declared type the receiver object used to call the method. This reduces call edges to the declared type implementation methods and the methods declared in the subtype hierarchy of the declared type of the receiver object.
- **Rapid Type Analysis (RTA) [Bacon 1996]:** in order to make CHA more efficient, the instantiation information of the whole system is considered by the RTA algorithm to have a better estimate of the runtime type of the receiver object used in a call site. In this case, the edges of the call graph are restricted to methods of classes, from the type hierarchy, which were instantiated in the system. Two versions exist for this algorithm: *pessimistic and optimistic*. The *pessimistic* version looks at all instantiations in the whole program. In the *optimistic* version, the call graph is iteratively created and only instantiations in methods already in the call graph are considered.

Our analysis for the *GoMC* recovery is a form of *RTA* algorithm optimistic version which uses information about instantiated classes to reduce the size of the call graph. At the end of this step, provided services of each module are identified.

5.3 Evaluation & Experimental Results

5.3.1 Data Collection

In order to evaluate the proposed approach, we conducted an experimentation on three open source Java projects namely *Jext* and *JHotDraw* used previously in the previous Chapter (Section 4.3). In addition to these systems, we applied our approach on MiniDraw [Christensen 2011] which is a pedagogical object-oriented

framework that is a scaled-down version of *JHotDraw*. It comes with several applications. The one chosen for our study is *LogoPuzzle* board game which is a puzzle on a University Logo. *MiniDraw* comprises 41 types, classes/interfaces containing a total of 1265 LOC.
In the remaining of this section, we present first the research question that we wanted to answer, then the setup of the experiments, and at last the results.

5.3.2 Research question

Since the proposed approach targets to identify modular clusters from an object-oriented system, the evaluation criteria needs to address modularity improvement. The experiments was conducted to answer the third research question presented in Chapter 1. As a reminder, the aim of the research question is to evaluate the output of the proposed approach and to compare it with the output of existing remodularization/clustering techniques.

5.3.3 Experiments setup

A prototype tool of the approach named *CACAO (a Composition conservAtive geneticC Algorithm for the remOdularization of OO software system)* was implemented. The tool is broken down into three parts: a part for the recovery of the object graph, implemented using Spoon [Pawlak 2015a], a part for the identification of the composition relationships and a part for the calculation of the solution based on a genetic algorithm. The solution is visualized using JxBrowser[7] which is an *API* that enables displaying *HTML, CSS* and *JavaScript* content in a Java application.
In order to test if the three systems, *JHotDraw*, *MiniDraw* and *Jext*, behave the same before and after applying the third step of the approach, the versions before and after refactoring were given to three Ph.D students, who where not involved in this work before, and who were asked to reproduce the same scenarios on the two versions of each system. The participants reported that the three systems behave correctly after refactoring.

5.3.4 Results and discussion

The research question is answered by assessing the accuracy of the remodularization solutions produced by three state-of -the-art techniques on the studied systems and comparing it to the output of our approach (CACAO). The techniques in question are listed below:

[7]https://www.teamdev.com/jxbrowser

1. **Bunch:** our choice for Bunch is motivated by the fact that it represents a well-known state-of-the-art architecture recovery tool and it was the subject of many recent comparison studies of software architecture recovery techniques [Lutellier 2015, Lutellier 2018, Paixão 2018]. We consider in this work two variations of Bunch's hill-climbing algorithm: next ascent hill climbing (NAHC) and steep ascent hill climbing (SAHC).

2. **ACDC (Algorithm for Comprehension-Driven Clustering):** the accuracy of this technique has been evaluated and confirmed in several comparative studies such as the ones of [Garcia 2013a] and [Stavropoulou 2017]

3. **ARC (Architectural Recovery using Concerns):** it was proven to be accurate and it was used as subject of comparison in several studies such as the ones of [Garcia 2013a] and [Langhammer 2016].

We obtained ACDC and Bunch implementations from their authors websites. For ARC, thanks to the ARCADE [Le 2015] workbench developers who welcomed our request and accepted to send us their implementation of ARC.
The first part to answer the research question consists of doing a comparative study based on the extent to which the clusters produced by the clustering approaches resembles clusters produced by the system's architect. For that, the authoritative, ground-truth, partition of a system must be obtained to compare the difference between the authoritative partition and the partition computed by the clustering approach. However, obtaining authoritative partition is a difficult task that requires heavy efforts [Kobayashi 2012, Garcia 2013c] since it is difficult for researchers to find original system developers. Therefore, we used the technique proposed by [Wu 2005] and used in several recent works such as [Corazza 2016] [Kobayashi 2012] and [Prajapati 2017]. This technique is summarized as follows:

- Each cluster corresponds to a package
- A cluster with a size less than to five types is merged into its parent cluster

In this first part of the research question, the *MoJoPlus* metric [Tzerpos 1999, Stroulia 2003] is used. *MoJoPlus* is a variation of the *MoJo* metric proposed by [Tzerpos 1999]. The input of this metric is two architectures of a given system. The metric calculates the minimal number of *move* and *join* operations, needed to transform one architecture into another. The *move* operation moves an entity from one cluster to another. The *join* operation merges two clusters into one cluster.
More precisely, given two architectures A and B, *MoJo* is calculated as follows:

$$MoJo(A, B) = min(mno(A, B), mno(B, A)) \tag{5.8}$$

Where *mno(A,B)* represents the number of *join* and *move* operations to go from A to B. In *MoJoPlus*, an additional type of operation is added: multi-move operation. This operation can be used to move a group of types from one module/cluster to another as a single operation that costs only 1. In order to evaluate the accuracy of the clustering techniques, we used the equation 5.9 [Tzerpos 1999]:

$$Q_{MoJoPlus(A,B)} = (1 - \frac{MoJoPlus(A,B)}{n}) * 100 \quad (5.9)$$

$Q_{MoJoPlus(A,B)}$ values ranges between 0% and 100%. High values indicate high correspondence between A and B.
Table 5.6 shows the $Q_{MoJoPlus(A,B)}$ scores for each of the studied clustering approaches and our approach. In our case, A varies to represent all the studied clustering approaches and B always represents the ground-truth architecture.
Since genetic algorithms are not deterministic (this is due to several factors such as the initial population and the selection operator), our proposed approach was run ten times on each of the three systems. Moreover, Due to the non-determinism of the clustering algorithm used by Bunch, we ran the clustering sixty times, thirty for each of its variations, *SAHC* and *NAHC*, and reported only the best results of $Q_{MoJoPlus(A,B)}$. In addition, ARC can vary in the number of concerns, so in its case we also ran the clustering twenty times with different numbers of concerns and we selected the best values of $Q_{MoJoPlus(A,B)}$.
In total, in generating Table 5.6, we computed $Q_{MoJoPlus(A,B)}$ values for 273 architectures (243 generated by the state-of-the-art clustering approaches and 30 generated by our approach).

Table 5.6: $Q_{MoJoPlus(A,B)}$ results

System	ACDC	ARC	SAHC	NAHC	CACAO
MiniDraw	88%	85%	85%	85%	59%
JHotDraw	89%	72%	87%	86%	92%
Jext	90%	72%	90%	91%	91%

In the case of *MiniDraw*, *ACDC* produces the most similar architecture to the chosen ground-truth architecture where the $Q_{MoJoPlus(A,B)}$ value is 88%. Followed by *Bunch-SAHC*, *Bunch-NAHC* and *ARC* with a $Q_{MoJoPlus(A,B)}$ value of 85%. Our approach is ranked last with a $Q_{MoJoPlus(A,B)}$ value of 59%.
In the case of *JHotDraw*, our approach outperforms all the architectures produced by the four state-of-the-art clustering approaches. It produces the most similar architecture to the chosen ground-truth architecture where the $Q_{MoJoPlus(A,B)}$ value is 92%. *ACDC*, *Bunch-NAHC* and *Bunch-SAHC* approaches produce architectures with so close, to our approach, $Q_{MoJoPlus(A,B)}$ values.
For *Jext*, our approach and Bunch-NAHC produce architectures with a $Q_{MoJoPlus(A,B)}$ value equal to 91%. *ACDC* and *Bunch-SAHC* also produce so close results. However, *ARC* is ranked last with a $Q_{MoJoPlus(A,B)}$ value equal to 72%.
For the three systems, *ACDC's* $Q_{MoJoPlus(A,B)}$ values range from 88% to 90%, *ARC* values from 40% to 85%, *Bunch-SAHC* from 72% to 90% and *Bunch-NAHC* from 68% to 91%. From the 243 selected architectures, our approach outperforms, from $Q_{MoJoPlus(A,B)}$ value viewpoint, 161 architectures (66% of the selected architectures).

For the chosen ground-truth architecture, $Q_{MoJoPlus(A,B)}$ values in Table 5.6 denote that all the clustering approaches can be of help to recover modular OO system architectures since they appear to be able to correctly cluster a large portion of the software system at hand. Moreover, the $Q_{MoJoPlus(A,B)}$ values can provide valuable intuition into which clustering approach is better suited for a particular software system. For example, *ACDC* appear to perform best on *MiniDraw*, our approach performs best on *JHotDraw*, while *Bunch-NAHC* and our approach do better with *Jext*. However, the $Q_{MoJoPlus(A,B)}$ value is not the only factor one needs to consider when choosing a clustering approach. Another factor that should be taken into account is the quality, in terms of modularity, of the solution produced by the clustering approach since the goal of this latter is to improve modularity. For this reason, modularity of the solutions produced by the different clustering approaches are compared in the following.
First, M, O and F values of the authoritative architectures of the three systems are calculated using the Formulas 5.2, 5.7 and 5.1 respectively and depicted in Table 5.7.

Table 5.7: M, O and F values of the authoritative architectures

System	M	O	F
MiniDraw	0.52	0.4	0.21
JHotDraw	0.52	0.45	0.23
Jext	0.59	0	0

The results of cohesion, coupling and modularity measures after applying our approach (*CACAO*) and all the chosen clustering approaches on *MiniDraw*, *JHotDraw* and *Jext* are depicted in Table 5.8. In this table, the best and the second best modularity values for each system, each modularity row, are highlighted in dark gray and light gray respectively.
From the results of Table 5.7 and 5.8, modularity values are improved by most approaches. The improvement ranges from from 12% (ARC) to 52% (Bunch-NAHC) for *MiniDraw*, from 21% (Bunch-SAHC) to 44% (ACDC and Bunch-NAHC) for *JHotDraw* and from 27% (our approach) to 45.8 (ACDC, Bunch-NAHC and Bunch-SAHC). The ARC technique produced a solution which is less modular than the ground truth architecture with a decrease of 38% in the modularity value in the case of *JHotDraw* and a decrease of 42.3% in the case of *Jext*. This result is justified by the fact that module identification in ARC is based on semantic relationships between classes and not on structural relationships (on which our fitness function is based).
Once the modularity values have been calculated, we pass now to the calculation of Module organization (O) measures. As stated before, in this study, x and y are fixed to 8 and 16 respectively as in [Bouwers 2011]. SB, CSU and O values are shown in Table 5.9. Also, the best and the second best module organization (O) values for

Table 5.8: Cohesion and Coupling measures of the different architectures

System		ACDC	ARC	NAHC	SAHC	CACAO
MiniDraw	**Cohesion**	25.73	28.59	39.33	38.15	34.05
	Coupling	23.87	20.98	10.28	11.27	14.4
	#Modules	4	8	4	4	8
	Modularity	0.52	0.58	0.79	0.77	0.71
JHotDraw	**Cohesion**	322.37	166.42	360.30	345.23	357.9
	Coupling	105.38	358.5	122.47	200.49	166.8
	#Modules	9	62	12	12	8
	Modularity	0.75	0.32	0.75	0.63	0.69
Jext	**Cohesion**	507.09	228	498.48	499.41	503.72
	Coupling	81	434	90.49	89.56	163.91
	#Modules	20	137	10	9	9
	Modularity	0.86	0.34	0.85	0.85	0.75

each system, each O row, are highlighted in dark gray and light gray respectively. In some cases, the O value is equal to 0 (for ARC on JHotDraw and ACDC & ARC on Jext). This is principally due to the high number of modules in the generated architecture.

Table 5.9: Module organization measures

System		ACDC	ARC	NAHC	SAHC	CACAO
MiniDraw	**SB**	0.43	1	0.43	0.43	1
	CSU	0.7	0.29	0.7	0.7	0.58
	O	0.3	0.29	0.3	0.3	0.58
JHotDraw	**SB**	0.88	0	0.5	0.5	1
	CSU	0.6	0.28	0.56	0.6	0.24
	O	0.53	0	0.28	0.3	0.24
Jext	**SB**	0	0	0.75	0.88	0.88
	CSU	0.36	0.21	0.34	0.52	0.4
	O	0	0	0.26	0.46	0.35

The last part of the experimentation consists of the calculation of the fitness function, based on Formula 5.1, for the three systems. Results are depicted in Table 5.10. In this study, α and β are fixed to 1 which means that the user is not interested in a measure (modularity and module/cluster organization) over the other.
Table 5.10 provides some insight into the performance of the used clustering approach on each used system. It indicates that our approach performs well on *MiniDraw* as compared to other approaches. In the case of *JHotDraw*, *ACDC* performs significantly better than other approaches. For *Jext*, *Bunch-SAHC* outperforms all the other approaches and our approach produces the next best remodularization solution. Overall, our approach produces the best and next best

Table 5.10: Fitness function measures

System	ACDC	ARC	NAHC	SAHC	CACAO
MiniDraw	0.16	0.17	0.24	0.23	0.41
JHotDraw	0.4	0	0.21	0.19	0.17
Jext	0	0	0.22	0.39	0.26

remodularization solutions in two, out of three, cases, Bunch-NAHC produces the second best solution in two cases and ACDC and Bunch-SAHC produce the best solution in one case. These results show that our approach is competitive to the state-of-the-art most used clustering approaches and that there is a potential to use it in remodularizing OO software systems.

5.3.5 Threats to validity

5.3.5.1 Internal validity

As in the work presented in the previous chapter, a potential threat concerns the programming language's dynamic features, like reflection. As mentioned previously, code that uses reflection is out of the bounds of our method. This code can reduce the accuracy of the static analysis phase (as discussed in Section 4.3.4.1).
An additional threat concerns the fact of using only one ground-truth architecture in order to answer the research questions. It is well known that for a single system, there are several architectures that can be deemed correct by the system's architects [Bowman 1999, Garcia 2013b]. This means that there can be other ground-truth architectures for the studied systems, that will obviously change the analysis results. Therefore, the use of this specific architecture may have been biased to the advantage of our approach. In order to narrow this threat, three metrics are used and two of them, *module organization (O)* and *modularity*, do not rely on the ground-truth architecture.
Another threat concerns the use of only four clustering approaches in our analysis study from a large body of works in the literature. It is sure that including additional approaches can strengthen our study results. However, this threat was mitigated by using techniques that take as input different types of information and use different clustering mechanisms. Moreover, these techniques were deemed accurate in several comparative studies.

5.3.5.2 External validity

One threat concerns the use of three object systems in this study. We believe that future experiments on a larger set of systems with larger sizes will yield more observations. However, since the results were positive with the studied systems; our intuition, on the interest of the proposed appraoch in improving modularity, is strengthened.

5.4 Conclusion

We presented in this chapter a remodularization approach that groups source code artifacts into highly cohesive modules and reduces the coupling between modules. In our work module identification is performed on runtime architecture models, in which we find the concrete interacting objects and the runtime coupled entities. To assess the validity of our proposal, we have conducted an empirical assessment of three open source software systems written in Java. This assessment was particularly conducted to compare the output of our approach with the one of the state-of-theart approaches in terms of improvement in modularity. Results showed that our approach is competitive to these state-of-the-art approaches.

Chapter

6

Conclusions And Perspectives

The goal of the work presented in thesis is the improvement of object-oriented legacy systems maintainability, in order to theoretically lengthen these systems lifetimes. To reach this goal, three approaches were proposed. In the remaining of this chapter, a summary of the proposed approaches and future directions that we are planning to do are presented.

6.1 Summary of Contributions

Our first contribution is an approach for migrating object-oriented legacy software systems, written in Java, into equivalent component-based equivalent ones. In contrast to exiting approaches that consider a component descriptor as a cluster of classes, each class in the legacy system is migrated into a component descriptor in our approach. The process is decomposed into two steps, the first step focuses on identifying symptoms of violation of two maintainability principles namely decoupling and unanticipated instantiations. After that, these symptoms are eliminated from the source code by applying a set of refactoring operations.
We believe in this thesis that there exist two ways to reason about maintainability. The first way consists of quantifying it using metrics that are known as predictive of maintainability. For this reason, the maintainability of the resulted refactored code in our first contribution is assessed using the *Maintainability Index (MI)* metric which is widely used in the literature. The results of *MI* have been improved

after applying the refactoring operations which strengthen our hypothesis about the usefulness of the proposed approach. The second way of reasoning about maintainability is to link it to other software quality attributes. In this thesis, maintainability is linked to two quality attributes namely understandability and modularity.
In order to improve understandability, we proposed an approach that enables to recover runtime architecture models of object-oriented legacy systems, written in Java, and manage the complexity of the resulted model. The runtime architecture recovery starts with a static analysis of the source code in order to build an object flow graph (OFG), which traces the objects from their creation until their storage in fields or use in method invocations, from which an object graph is extracted. Nodes in the object graph represent concrete interacting entities in the system. Edges between nodes represent field references between objects. Using a dynamic analysis, the object graph is refined by adding labels on its nodes. To manage the complexity of the resulted object graph, the composite structures are identified and the visualized graph is filtered based on this composite structure and by identifying thresholds on the values of the labels added on nodes. The gain in understandability is measured by answering three research questions concerning the complexity of the resulted runtime architecture, the time spent and the correctness in handling comprehension tasks using the graph and the possibility to identify refactoring opportunities.
In order to improve modularity, we proposed an approach for enhancing the structure of object-oriented software systems by identifying modules and services in the source code. In contrast to existing works in the literature, the process starts by a step of runtime models recovery. These models represent the concrete interacting objects that compose the running system and their inter-dependencies. These models are exploited in order to identify composition relationships between source code artifacts. Once these composition relationships have been identified, a composition conservative genetic algorithm is applied on the software system in order to identify modules. At last, services that allow modules to communicate are identified, based on the runtime models, in order to further improve decoupling. An experimentation of this process has been conducted in order to evaluate the gain in modularity.
From a practical (tooling) point of view, the contributions of this thesis are the following:

1. A prototype tool implementing the refactoring operations proposed in the first contribution;

2. The *DIALOG (refineD and hIerArchicaL Object Graph)* prototype tool for the recovery and visualization of runtime architecture models;

3. The *CACAO (Composition conservAtive genetiC Algorithm for the remOdularization of OO software system)* prototype tool;

4. A dataset of 273 architecture descriptions, in RSF format, that can be used by other researchers in their experiments.

6.2 Future Directions

The presented work makes emerge some limitations which have been judged as deserving a deeper study. In the threats to validity section of each contribution, some limitations that are the subject of short-term perspectives have been presented. In the following, mid-term and long-term perspectives are presented.

6.2.1 Enrich the recovered OG with other kinds of information

In order to further improve understandability, we can study the extraction of other kinds of information that can be used in the visualization with a level of detail, like the number of dependencies between visualized objects. This can help in detecting anti-patterns whose identification is based on the properties of objects. Besides this, we can enrich the recovered graphs with variability aspects, like in feature models of software product lines. The idea is to include, in these OGs, variability points and their associated constraints (or, xor, ...), and exploit them in the visualization with a level of detail.

6.2.2 Consider semantic relationships between classes to improve modularity

In order to further improve coupling/cohesion estimation, semantic/lexical relationships between classes should be taken into account in the proposed fitness function in the third contribution. Semantic relationships are derived based on similarity between comments and identifiers of the source code artifacts of the studied systems. Similarity measures between source code tokens can be calculated by resorting to text mining techniques.

6.2.3 Adapt the proposed approaches to other programming languages

As you have probably noticed, all the subject systems in the experimentation sections of the proposed approaches are implemented in Java. An improvement that can be applied to the proposed approaches consists of adapting them to support other object-oriented programing languages, supporting other object-oriented constructs, like *Prototypes* in *JavaScript* and *Embeddings* in *Go* (*golang*).

6.2.4 Identification of modules and services as a machine learning problem

Besides this and in the emerging research field of the application of machine/deep learning to software engineering, we can develop a new approach that considers the identification of modules and services from object-oriented legacy systems as a machine learning problem. A training data needs to be collected based on the module and service characteristics and the ways existing systems have been migrated to these paradigms (JDK9 and modern Java applications to the module system, for example). A classifier can then be built using the training data in order to assist module or service identification.

6.2.5 Develop a framework that groups the proposed approaches

As another perspective to this work, we can build a framework that combines all our proposed approaches. The framework can also be enriched by other state-of-the-art approaches and a set of metrics for assessing the improvement of maintainability and comparing the different approaches. The framework can be integrated within a software development setting (an IDE for instance) in order to acquire real-life case studies on the value brought to maintainers.

Bibliography

[Abdellatif 2018] Manel Abdellatif, Geoffrey Hecht, Hafedh Mili, Ghizlane El-Boussaidi, Naouel Moha, Anas Shatnawi, Jean Privat and Yann-Gaël Guéhéneuc. *State of the Practice in Service Identification for SOA Migration in Industry*. In Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings, pages 634–650, 2018. (Cité à la page 6.)

[Abi-Antoun 2009] Marwan Abi-Antoun and Jonathan Aldrich. *Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations*. In Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 2009. (Cité aux pages 27 and 32.)

[Adjoyan 2014] Seza Adjoyan, Abdelhak-Djamel Seriai and Anas Shatnawi. *Service Identification Based on Quality Metrics Object-Oriented Legacy System Migration Towards SOA*. In SEKE: Software Engineering and Knowledge Engineering, pages 1–6. Knowledge Systems Institute Graduate School, 2014. (Cité aux pages 20, 30, and 32.)

[Ahn 2018] Hwi Ahn, Sungwon Kang and Seonah Lee. *Reconstruction of execution architecture view using dependency relationships and execution traces*. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018, pages 1417–1424, 2018. (Cité à la page 4.)

[Aho 1972] Alfred V Aho and Jeffrey D Ullman. The theory of parsing, translation, and compiling, volume 1. Prentice-Hall Englewood Cliffs, NJ, 1972. (Cité à la page 68.)

[Akroyd 1996] Michael Akroyd. *Anti patterns session notes*. Object World, 1996. (Cité à la page 78.)

[Al Qutaish 2005] Rafa E Al Qutaish and Alain Abran. *An analysis of the design and definitions of Halstead metrics*. In IWSM, 2005. (Cité à la page 47.)

[Al-Rubaye 2017] Samer Raad Azzawi Al-Rubaye and Yunus Emre Selcuk. *An investigation of code cycles and Poltergeist anti-pattern*. In Software Engineering and Service Science (ICSESS), 2017 8th IEEE International Conference on, pages 139–140. IEEE, 2017. (Cité aux pages 78, 79, and 80.)

[Alahmari 2010] Saad Alahmari, Ed Zaluska and David De Roure. *A service identification framework for legacy system migration into SOA*. In Services Computing (SCC), 2010 IEEE International Conference on, pages 614–617. IEEE, 2010. (Cité à la page 18.)

[Alimadadi 2018] Saba Alimadadi, Ali Mesbah and Karthik Pattabiraman. *Inferring hierarchical motifs from execution traces*. In Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pages 776–787, 2018. (Cité à la page 74.)

[Alkazemi 2013] Basem Y. Alkazemi, Mohammed K. Nour and Abdulqader Qada Meelud. *Towards a Framework to Assess Legacy Systems*. In IEEE International Conference on Systems, Man, and Cybernetics, Manchester, SMC 2013, United Kingdom, October 13-16, 2013, pages 924–928, 2013. (Cité à la page 3.)

[Allen 1970] Frances E Allen. *Control flow analysis*. In ACM Sigplan Notices, volume 5, pages 1–19. ACM, 1970. (Cité à la page 68.)

[Allen 1972] Frances E Allen and John Cocke. Graph-theoretic constructs for program control flow analysis. IBM Thomas J. Watson Research Center, 1972. (Cité à la page 68.)

[Allier 2011] Simon Allier, Salah Sadou, Houari Sahraoui*et al. From object-oriented applications to component-oriented applications via component-oriented architecture*. In Working IEEE/IFIP Conference on Software Architecture (WICSA), 2011. (Cité aux pages 16, 30, 32, and 94.)

[Alshara 2015] Zakarea Alshara*et al. Migrating large object-oriented Applications into component-based ones: instantiation and inheritance transformation*. In GPCE, 2015. (Cité aux pages 16, 30, and 32.)

[Alshara 2016] Zakarea Alshara, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde-Lilia Bouziane, Christophe Dony and Anas Shatnawi. *Materializing Architecture Recovered from Object-Oriented Source Code in Component-Based Languages*. In - 10th European Conference on Software Architecture ECSA 2016, Copenhagen, Denmark, November 28 - December 2, 2016, Proceedings, pages 309–325, 2016. (Cité à la page 16.)

[Ammar 2012] Nariman Ammar and Marwan Abi-Antoun. *Empirical evaluation of diagrams of the run-time structure for coding tasks*. In Working Conference on Reverse Engineering (WCRE), 2012. (Cité aux pages 8, 31, 69, and 81.)

[Anjos 2017] Eudisley Gomes dos Anjos. *Assessing Maintainability in Software Architectures*. PhD thesis, Universidade de Coimbra, 2017. (Cité à la page 5.)

[Arnold 1993] Robert S Arnold. Software reengineering. IEEE Computer Society Press, 1993. (Cité à la page 4.)

[Athanasopoulos 2017] Dionysis Athanasopoulos. *Usage-Aware Service Identification for Architecture Migration of Object-Oriented Systems to SoA*. In International Conference on Database and Expert Systems Applications, pages 54–64. Springer, 2017. (Cité à la page 4.)

[Bacon 1996] David F Bacon and Peter F Sweeney. *Fast static analysis of C++ virtual function calls*. ACM Sigplan Notices, vol. 31, no. 10, pages 324–341, 1996. (Cité à la page 102.)

[Balogh 2013] Gergo Balogh and Árpád Beszédes. *CodeMetropolis - code visualisation in MineCraft*. In 13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013, pages 136–141, 2013. (Cité à la page 28.)

[Balogh 2015] Gergo Balogh, Attila Szabolics and Árpád Beszédes. *CodeMetropolis: Eclipse over the city of source code*. In 15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015, pages 271–276, 2015. (Cité à la page 28.)

[Basili 1996] Victor R Basili, Lionel C. Briand and Walcélio L Melo. *A validation of object-oriented design metrics as quality indicators*. IEEE Transactions on software engineering, vol. 22, no. 10, pages 751–761, 1996. (Cité à la page 38.)

[Benaroch 2013] Michel Benaroch. *Primary drivers of software maintenance cost studied using longitudinal data*. 2013. (Cité à la page 5.)

[Bennett 1999] Keith H Bennett, Magnus Ramage and Malcolm Munro. *Decision model for legacy systems*. IEE Proceedings-Software, vol. 146, no. 3, pages 153–159, 1999. (Cité à la page 2.)

[Bernardi 2016] Mario Luca Bernardi, Marta Cimitile and Giuseppe A. Di Lucca. *Mining static and dynamic crosscutting concerns: a role-based approach.* Journal of Software: Evolution and Process, vol. 28, no. 5, pages 306–339, 2016. (Cité à la page 24.)

[Bertolino 2005] Antonia Bertolino*et al. An architecture-centric approach for producing quality systems.* In Quality of Software Architectures and Software Quality. Springer, 2005. (Cité à la page 7.)

[Biggerstaff 1993] Ted J Biggerstaff, Bharat G Mitbander and Dallas Webster. *The concept assignment problem in program understanding.* In Proceedings of the 15th international conference on Software Engineering, pages 482–498. IEEE Computer Society Press, 1993. (Cité à la page 6.)

[Binkley 2006] David W. Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca and Paolo Tonella. *Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects.* IEEE Trans. Software Eng., vol. 32, no. 9, pages 698–717, 2006. (Cité aux pages 24 and 30.)

[Blei 2003] David M Blei, Andrew Y Ng and Michael I Jordan. *Latent dirichlet allocation.* Journal of machine Learning research, vol. 3, no. Jan, pages 993–1022, 2003. (Cité à la page 30.)

[Boehm 1976] Barry W. Boehm, John R. Brown and M. Lipow. *Quantitative Evaluation of Software Quality.* In Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, USA, October 13-15, 1976., pages 592–605, 1976. (Cité à la page 2.)

[Börstler 2016] Jürgen Börstler, Michael E Caspersen and Marie Nordström. *Beauty and the Beast: on the readability of object-oriented example programs.* Software quality journal, vol. 24, no. 2, pages 231–246, 2016. (Cité à la page 46.)

[Bouwers 2011] Eric Bouwers, José Pedro Correia, Arie van Deursen and Joost Visser. *Quantifying the analyzability of software architectures.* In Working IEEE/IFIP Conference on Software Architecture (WICSA), 2011. (Cité aux pages 95, 96, and 106.)

[Bowman 1999] Ivan T Bowman, Richard C Holt and Neil V Brewster. *Linux as a case study: Its extracted software architecture.* In Proceedings of the 21st international conference on Software engineering, pages 555–563. ACM, 1999. (Cité à la page 108.)

[Breu 2004] Silvia Breu and Jens Krinke. *Aspect Mining Using Event Traces.* In 19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria, pages 310–315, 2004. (Cité à la page 24.)

[Briand 2006] Lionel C Briand, Yvan Labiche and Johanne Leduc. *Toward the reverse engineering of UML sequence diagrams for distributed Java software.* IIEEE Trans on Soft Eng, 2006. (Cité aux pages 27 and 28.)

[Brown 1998] William J Brown*et al.* *Refactoring Software, Architectures, and Projects in Crisis*, 1998. (Cité à la page 78.)

[Brown 2002] Alan Brown, Simon Johnston and Kevin Kelly. *Using service-oriented architecture and component-based development to build web service applications.* Rational Software Corporation, vol. 6, pages 1–16, 2002. (Cité à la page 99.)

[Canfora 2008] Gerardo Canfora, Anna Rita Fasolino, Gianni Frattolillo and Porfirio Tramontana. *A wrapping approach for migrating legacy system interactive functionalities to Service Oriented Architectures.* Journal of Systems and Software, vol. 81, no. 4, pages 463–480, 2008. (Cité à la page 19.)

[Ceccato 2007] Mariano Ceccato. *Migrating Object Oriented code to Aspect Oriented Programming.* In 23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France, pages 497–498, 2007. (Cité aux pages 25 and 30.)

[Ceccato 2008] Mariano Ceccato. *Automatic Support for the Migration Towards Aspects.* In 12th European Conference on Software Maintenance and Reengineering, CSMR 2008, April 1-4, 2008, Athens, Greece, pages 298–301, 2008. (Cité aux pages 25 and 30.)

[Chatterji 2013] Debarshi Chatterji, Jeffrey C. Carver, Nicholas A. Kraft and Jan Harder. *Effects of cloned code on software maintainability: A replicated developer study.* In 20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013, pages 112–121, 2013. (Cité à la page 30.)

[Cheng 2006] Shih-Sian Cheng, Yi-Hsiang Chao, Hsin-Min Wang and Hsin-Chia Fu. *A Prototypes-Embedded Genetic K-means Algorithm.* In 18th International Conference on Pattern Recognition (ICPR 2006), 20-24 August 2006, Hong Kong, China, pages 724–727, 2006. (Cité à la page 99.)

[Chidamber 1994] Shyam R Chidamber and Chris F Kemerer. *A metrics suite for object oriented design.* IEEE Transactions on software engineering, vol. 20, no. 6, pages 476–493, 1994. (Cité à la page 38.)

[Chikofsky 1990] Elliot J. Chikofsky and James H Cross. *Reverse engineering and design recovery: A taxonomy.* IEEE software, vol. 7, no. 1, pages 13–17, 1990. (Cité aux pages xiii, 3, 4, and 5.)

[Christensen 2011] Henrik B Christensen. Flexible, reliable software: using patterns and agile development. CRC Press, 2011. (Cité aux pages 70 and 102.)

[Clarke 2013] Dave Clarke, James Noble and Tobias Wrigstad, editeurs. Aliasing in object-oriented programming. types, analysis and verification, volume 7850 of *Lecture Notes in Computer Science*. Springer, 2013. (Cité à la page 101.)

[Constantinou 2015] Eleni Constantinou, Athanasios Naskos, George Kakarontzas*et al.* *Extracting reusable components: A semi-automated approach for complex structures*. Information Processing Letters, 2015. (Cité aux pages 16, 30, and 32.)

[Cooper 2001] Keith D Cooper, Timothy J Harvey and Ken Kennedy. *A simple, fast dominance algorithm*. Soft Prac & Exper, 2001. (Cité à la page 68.)

[Corazza 2016] Anna Corazza, Sergio Di Martino, Valerio Maggio and Giuseppe Scanniello. *Weighing lexical information for software clustering in the context of architecture recovery*. Empirical Software Engineering, vol. 21, no. 1, pages 72–103, 2016. (Cité à la page 104.)

[Cornelissen 2009a] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen and Rainer Koschke. *A systematic survey of program comprehension through dynamic analysis*. IEEE Transactions on Software Engineering, vol. 35, no. 5, pages 684–702, 2009. (Cité à la page 6.)

[Cornelissen 2009b] Bas Cornelissen, Andy Zaidman, Arie Van Deursen and Bart Van Rompaey. *Trace visualization for program comprehension: A controlled experiment*. In International Conference on Program Comprehension (ICPC), 2009. (Cité aux pages 29, 74, and 81.)

[Crotty 2017] James Crotty and Ivan Horrocks. *Managing legacy system costs: A case study of a meta-assessment model to identify solutions in a large financial services company*. Applied computing and informatics, vol. 13, no. 2, pages 175–183, 2017. (Cité à la page 2.)

[Davis 1990] J. Steve Davis. *Effect of Modularity on Maintainability of Rule-Based Systems*. International Journal of Man-Machine Studies, vol. 32, no. 4, pages 439–447, 1990. (Cité à la page 6.)

[de Brito 2013] Hugo de Brito, Humberto Torres Marques-Neto*et al.* *On-the-fly extraction of hierarchical object graphs*. Journal of the Brazilian Computer Society (JBCS), 2013. (Cité aux pages 27, 28, and 32.)

[Dean 1995] Jeffrey Dean, David Grove and Craig Chambers. *Optimization of object-oriented programs using static class hierarchy analysis*. In European Conference on Object-Oriented Programming, pages 77–101. Springer, 1995. (Cité à la page 102.)

[Dromey 1995] R. Geoff Dromey. *A Model for Software Product Quality.* IEEE Trans. Software Eng., vol. 21, no. 2, pages 146–162, 1995. (Cité à la page 2.)

[Eder 2012] Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas and Karl-Heinz Prommer. *How much does unused code matter for maintenance?* In 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, pages 1102–1111, 2012. (Cité à la page 30.)

[Ernst 2017] Neil A Ernst, Stephany Bellomo, Ipek Ozkaya and Robert L Nord. *What to Fix? Distinguishing between design and non-design rules in automated tools.* In International Conference on Software Architecture(ICSA), 2017. (Cité à la page 95.)

[Erradi 2006] Abdelkarim Erradi, Sriram Anand and Naveen Kulkarni. *SOAF: An architectural framework for service definition and realization.* In Services Computing, 2006. SCC'06. IEEE International Conference on, pages 151–158. IEEE, 2006. (Cité à la page 99.)

[Escobar 2016] Daniel Escobar, Diana Cárdenas, Rolando Amarillo, Eddie Castro, Kelly Garcés, Carlos Parra and Rubby Casallas. *Towards the understanding and evolution of monolithic applications as microservices.* In Computing Conference (CLEI), 2016 XLII Latin American, pages 1–11. IEEE, 2016. (Cité aux pages 21, 30, and 32.)

[Fabresse 2008] Luc Fabresse*et al. Foundations of a simple and unified component-oriented language.* Computer Languages, Systems & Structures, vol. 34, no. 2, pages 130–149, 2008. (Cité à la page 36.)

[Fittkau 2015] Florian Fittkau, Santje Finke, Wilhelm Hasselbring*et al. Comparing trace visualizations for program comprehension through controlled experiments.* In International Conference on Program Comprehension (ICPC), 2015. (Cité à la page 74.)

[Fittkau 2017] Florian Fittkau, Alexander Krause and Wilhelm Hasselbring. *Software landscape and application visualization for system comprehension with ExplorViz.* Information & Software Technology, vol. 87, pages 259–277, 2017. (Cité à la page 29.)

[Flanagan 2006] Cormac Flanagan and Stephen N Freund. *Dynamic architecture extraction.* In FATES/RV. Springer, 2006. (Cité aux pages 27, 28, and 32.)

[Fowler 1999] Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts. Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999. (Cité aux pages 4, 25, 31, and 37.)

[Fuhr 2011] Andreas Fuhr, Tassilo Horn and Volker Riediger. *Using Dynamic Analysis and Clustering for Implementing Services by Reusing Legacy Code*. In 18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011, pages 275–279, 2011. (Cité aux pages 20 and 30.)

[Galorath 2006] Daniel D Galorath and Michael W Evans. Software sizing, estimation, and risk management: when performance is measured performance improves. Auerbach Publications, 2006. (Cité à la page 5.)

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design patterns: Elements of reusable software architecture*. Reading: Addison-Wesley, 1995. (Cité à la page 37.)

[Garcia 2011] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic and Yuanfang Cai. *Enhancing architectural recovery using concerns*. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, pages 552–555. IEEE Computer Society, 2011. (Cité aux pages 30 and 32.)

[Garcia 2013a] Joshua Garcia, Igor Ivkovic and Nenad Medvidovic. *A comparative analysis of software architecture recovery techniques*. In Automated Software Engineering (ASE), 2013. (Cité à la page 104.)

[Garcia 2013b] Joshua Garcia, Ivo Krka, Chris Mattmann*et al.* *Obtaining ground-truth software architectures*. In International Conference on Software Engineering (ICSE), 2013. (Cité à la page 108.)

[Garcia 2013c] Joshua Garcia, Ivo Krka, Chris Mattmann and Nenad Medvidovic. *Obtaining ground-truth software architectures*. In 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, pages 901–910, 2013. (Cité à la page 104.)

[Gligoric 2013] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz and Darko Marinov. *Systematic testing of refactoring engines on real software projects*. In European Conference on Object-Oriented Programming, pages 629–653. Springer, 2013. (Cité à la page 42.)

[Grady 1992] Robert B Grady. Practical software metrics for project management and process improvement. Prentice-Hall, Inc., 1992. (Cité à la page 2.)

[Griffiths 2010] Nathan Griffiths and Kuo-Ming Chao. Agent-based service-oriented computing. Springer, 2010. (Cité à la page 18.)

[Gruber 2005] Olivier Gruber, B. J. Hargrave, Jeff McAffer, Pascal Rapicault and Thomas Watson. *The Eclipse 3.0 platform: Adopting OSGi technology*. IBM Systems Journal, vol. 44, no. 2, pages 289–300, 2005. (Cité à la page 99.)

[Gysel 2016] Michael Gysel, Lukas Kölbener, Wolfgang Giersche and Olaf Zimmermann. *Service cutter: A systematic approach to service decomposition*. In European Conference on Service-Oriented and Cloud Computing, pages 185–200, 2016. (Cité aux pages 22 and 32.)

[Hu 2014] Tao Hu and Tu Peng. *Multi-angle Evaluations of Test Cases Based on Dynamic Analysis*. In International Conference on Advanced Data Mining and Applications (ADMA), 2014. (Cité aux pages 65 and 70.)

[Huang 2009] Weidong Huang, Peter Eades and Seok-Hee Hong. *Measuring effectiveness of graph visualizations: A cognitive load perspective*. Information Visualization, 2009. (Cité aux pages 69 and 73.)

[Islam 2018] Md Zahidul Islam, Vladimir Estivill-Castro, Md Anisur Rahman and Terry Bossomaier. *Combining K-Means and a genetic algorithm through a novel arrangement of genetic operators for high quality clustering*. Expert Syst. Appl., vol. 91, pages 402–417, 2018. (Cité à la page 93.)

[ISO.25010 2008] ISO.25010. *Systems and software Quality Requirements and Evaluation (SQuaRE)*. Rapport technique, 2008. (Cité aux pages 2 and 6.)

[Jain 2001] Hemant K. Jain, Naresh Chalimeda, Navin Ivaturi and Balarama Reddy. *Business Component Identification - A Formal Approach*. In 5th International Enterprise Distributed Object Computing Conference (EDOC 2001), 4-7 September 2001, Seattle, WA, USA, Proceedings, pages 183–187, 2001. (Cité aux pages 14, 30, and 32.)

[Kebir 2012] Selim Kebir, Abdelhak-Djamel Seriai, Sylvain Chardigny and Allaoua Chaoui. *Quality-centric approach for software component identification from object-oriented code*. In European Conference on Software Architecture(ECSA), 2012. (Cité à la page 16.)

[Kerdoudi 2016] Mohamed Lamine Kerdoudi, Chouki Tibermacine and Salah Sadou. *Opening web applications for third-party development: a service-oriented solution*. Service Oriented Computing and Applications, vol. 10, no. 4, pages 437–463, 2016. (Cité à la page 20.)

[Khadka 2011] Ravi Khadka, Gijs Reijnders, Amir Saeidi, Slinger Jansen and Jurriaan Hage. *A method engineering based legacy to SOA migration method*. In IEEE 27th International Conference on Software Maintenance, ICSM, pages 163–172, 2011. (Cité à la page 18.)

[Khadka 2013] Ravi Khadka, Amir Saeidi, Slinger Jansen and Jurriaan Hage. *A structured legacy to SOA migration process and its evaluation in practice*. In 7th IEEE International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems, MESOCA 2013, Eindhoven, The Netherlands, September 23, 2013, pages 2–11, 2013. (Cité à la page 18.)

[Kim 2004] Soo Dong Kim and Soo Ho Chang. *A Systematic Method to Identify Software Components*. In 11th Asia-Pacific Software Engineering Conference (APSEC 2004), 30 November - 3 December 2004, Busan, Korea, pages 538–545, 2004. (Cité aux pages 15, 30, and 32.)

[Kobayashi 2012] Kenichi Kobayashi, Manabu Kamimura, Koki Kato, Keisuke Yano and Akihiko Matsuo. *Feature-gathering dependency-based software clustering using Dedication and Modularity*. In 28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012, pages 462–471, 2012. (Cité à la page 104.)

[Koenig 1998] Andrew Koenig. *Patterns and antipatterns*. The patterns handbook: techniques, strategies, and applications, vol. 13, page 383, 1998. (Cité à la page 78.)

[Koteska 2018] Bojana Koteska, Anastas Mishev and Ljupco Pejov. *Quantitative Measurement of Scientific Software Quality: Definition of a Novel Quality Model*. International Journal of Software Engineering and Knowledge Engineering, vol. 28, no. 3, page 407, 2018. (Cité à la page 46.)

[Kruchten 2012] Philippe Kruchten, Robert L Nord and Ipek Ozkaya. *Technical debt: From metaphor to theory and practice*. Ieee software, vol. 29, no. 6, pages 18–21, 2012. (Cité à la page 4.)

[Labiche 2013] Yvan Labiche, Bojana Kolbah and Hossein Mehrfard. *Combining static and dynamic analyses to reverse-engineer scenario diagrams*. In International Conference on Software Maintenance (ICSM), 2013. (Cité à la page 28.)

[Langelier 2005] Guillaume Langelier, Houari A. Sahraoui and Pierre Poulin. *Visualization-based analysis of quality for large-scale software systems*. In Automated Software Engineering (ASE), 2005. (Cité à la page 28.)

[Langhammer 2016] Michael Langhammer, Arman Shahbazian, Nenad Medvidovic and Ralf H Reussner. *Automated extraction of rich software models from limited system information*. In Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on, pages 99–108. IEEE, 2016. (Cité à la page 104.)

[Lanza 2007] Michele Lanza and Radu Marinescu. Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer Science & Business Media, 2007. (Cité à la page 78.)

[Le 2015] Duc Minh Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian and Nenad Medvidovic. *An empirical study of architectural change in open-source software systems*. In Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on, pages 235–245. IEEE, 2015. (Cité à la page 104.)

[Lee 2003] Eunjoo Lee, Byungjeong Lee, Woochang Shin and Chisu Wu. *A reengineering process for migrating from an object-oriented legacy system to a component-based system*. In International Computer Software and Applications Conference (COMPSAC), pages 336–341. IEEE, 2003. (Cité aux pages 15, 30, and 32.)

[Lee 2008] Seonah Lee, Gail C. Murphy, Thomas Fritz and Meghan Allen. *How can diagramming tools help support programming activities?* In VL/HCC, 2008. (Cité aux pages 8 and 31.)

[Lehman 1997] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry and Wladyslaw M Turski. *Metrics and laws of software evolution-the nineties view*. In Software metrics symposium, 1997. proceedings., fourth international, pages 20–32, 1997. (Cité à la page 2.)

[Lengauer 1979] Thomas Lengauer and Robert Endre Tarjan. *A fast algorithm for finding dominators in a flowgraph*. Programming Languages and Systems (TOPLAS), 1979. (Cité à la page 68.)

[Levcovitz 2016] Alessandra Levcovitz, Ricardo Terra and Marco Tulio Valente. *Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems*. CoRR, vol. abs/1605.03175, 2016. (Cité à la page 22.)

[Li 2006] Shimin Li and Ladan Tahvildari. *A Service-Oriented Componentization Framework for Java Software Systems*. In 13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy, pages 115–124, 2006. (Cité aux pages 18, 30, 32, and 33.)

[Liu 2007] Yin Liu and Ana Milanova. *Ownership and Immutability Inference for UML-Based Object Access Control*. In 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, pages 323–332, 2007. (Cité à la page 86.)

[Liu 2018] Cong Liu, Boudewijn F. van Dongen, Nour Assy and Wil M. P. van der Aalst. *Component interface identification and behavioral model discovery from software execution data*. In Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018, pages 97–107, 2018. (Cité à la page 17.)

[Llano 2009] Maria Teresa Llano and Rob Pooley. *UML Specification and Correction of Object-Oriented Antipatterns*. In International Conference on Software Engineering Advances (ICSEA), 2009. (Cité aux pages 79 and 80.)

[Lucia 2001] Andrea De Lucia, Anna Rita Fasolino and Eugenio Pompella. *A Decisional Framework for Legacy System Management*. In 2001 International Conference on Software Maintenance, ICSM 2001, Florence, Italy, November 6-10, 2001, page 642, 2001. (Cité aux pages xiii and 3.)

[Lucia 2018] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino and Michele Risi. *Detecting the Behavior of Design Patterns through Model Checking and Dynamic Analysis*. ACM Trans. Softw. Eng. Methodol., vol. 26, no. 4, pages 13:1–13:41, 2018. (Cité à la page 81.)

[Lutellier 2015] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic and Robert Kroeger. *Comparing Software Architecture Recovery Techniques Using Accurate Dependencies*. In 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2, pages 69–78, 2015. (Cité aux pages 4 and 104.)

[Lutellier 2018] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic and Robert Kroeger. *Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques*. IEEE Trans. Software Eng., vol. 44, no. 2, pages 159–181, 2018. (Cité à la page 104.)

[Marin 2004] Marius Marin, Arie van Deursen and Leon Moonen. *Identifying Aspects Using Fan-In Analysis*. In 11th Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8-12, 2004, pages 132–141, 2004. (Cité à la page 24.)

[Mazlami 2017] Genc Mazlami, Jürgen Cito and Philipp Leitner. *Extraction of Microservices from Monolithic Software Architectures*. In 2017 IEEE International Conference on Web Services, ICWS 2017, Honolulu, HI, USA, June 25-30, 2017, pages 524–531, 2017. (Cité aux pages 4, 23, and 30.)

[McAffer 2010] Jeff McAffer, Paul VanderLei and Simon Archer. Osgi and equinox: Creating highly modular java systems. Addison-Wesley Professional, 2010. (Cité à la page 99.)

[McCall 1977] Jim A McCall, Paul K Richards and Gene F Walters. *Factors in software quality. volume i. concepts and definitions of software quality*. Rapport technique, GENERAL ELECTRIC CO SUNNYVALE CA, 1977. (Cité à la page 2.)

[McClure 1992] Carma McClure. The three rs of software automation: re-engineering, repository, reusability. Prentice Hall Englewood Cliffs, NJ, 1992. (Cité à la page 4.)

[Milanova 2007] Ana Milanova. *Composition inference for UML class diagrams.* Autom. Softw. Eng., vol. 14, no. 2, pages 179–213, 2007. (Cité aux pages 86 and 87.)

[Mitchell 1998] Melanie Mitchell. An introduction to genetic algorithms. MIT press, 1998. (Cité à la page 98.)

[Mitchell 2006] Brian S Mitchell and Spiros Mancoridis. *On the automatic modularization of software systems using the bunch tool.* IEEE Transactions on Software Engineering, vol. 32, no. 3, pages 193–208, 2006. (Cité aux pages 29 and 32.)

[Nakamura 2009] Masahide Nakamura, Hiroshi Igaki, Takahiro Kimura and Ken-ichi Matsumoto. *Extracting service candidates from procedural programs based on process dependency analysis.* In Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific, pages 484–491. IEEE, 2009. (Cité à la page 99.)

[Newman 2015] Sam Newman. Building microservices: designing fine-grained systems. " O'Reilly Media, Inc.", 2015. (Cité à la page 21.)

[Ouni 2013] Ali Ouni, Marouane Kessentini, Houari A. Sahraoui and Mounir Boukadoum. *Maintainability defects detection and correction: a multi-objective approach.* Autom. Softw. Eng., vol. 20, no. 1, pages 47–79, 2013. (Cité aux pages 26 and 31.)

[Pacione 2004] Michael John Pacione, Marc Roper and Murray Wood. *A novel software visualisation model to support software comprehension.* In Working Conference on Reverse Engineering (WCRE), 2004. (Cité aux pages 74 and 75.)

[Paixão 2018] Matheus Paixão, Mark Harman, Yuanyuan Zhang and Yijun Yu. *An Empirical Study of Cohesion and Coupling: Balancing Optimization and Disruption.* IEEE Trans. Evolutionary Computation, vol. 22, no. 3, pages 394–414, 2018. (Cité à la page 104.)

[Pawlak 2015a] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez*et al. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code.* Software: Practice and Experience, 2015. (Cité aux pages 70 and 103.)

[Pawlak 2015b] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera and Lionel Seinturier. *Spoon: A Library for Implementing Analyses and Transformations of Java Source Code*. Software: Practice and Experience, vol. 46, pages 1155–1179, 2015. (Cité à la page 45.)

[Peiris 2016] Manjula Peiris and James H. Hill. *Automatically Detecting "Excessive Dynamic Memory Allocations" Software Performance Anti-Pattern*. In Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016, pages 237–248, 2016. (Cité à la page 32.)

[Prajapati 2017] Amarjeet Prajapati and Jitender Kumar Chhabra. *Improving modular structure of software system using structural and lexical dependency*. Information and Software Technology, 2017. (Cité aux pages 94, 95, and 104.)

[Rajlich 1997] Vaclav Rajlich and George S Cowan. *Towards standard for experiments in program comprehension*. In International Workshop on Program Comprehension (IWPC), 1997. (Cité à la page 74.)

[Ramírez 2016] Aurora Ramírez, José Raúl Romero and Sebastián Ventura. *A comparative study of many-objective evolutionary algorithms for the discovery of software architectures*. Empirical Software Engineering, 2016. (Cité à la page 95.)

[Ransom 1998] Jane Ransom, Ian Sommerville and Ian Warren. *A Method for Assessing Legacy Systems for Evolution*. In 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR '98), 8-11 March 1998, Florence, Italy, pages 128–134, 1998. (Cité à la page 3.)

[Riel 1996] Arthur J Riel. Object-oriented design heuristics. Addison-Wesley Longman Publishing Co., Inc., 1996. (Cité à la page 79.)

[Schmidt 2018] Frédéric Schmidt, Stephen G. MacDonell and Andy M. Connor. *Multi-Objective Reconstruction of Software Architecture*. International Journal of Software Engineering and Knowledge Engineering, vol. 28, no. 6, page 869, 2018. (Cité à la page 4.)

[Schneidewind 1987] Norman F. Schneidewind. *The state of software maintenance*. IEEE Transactions on Software Engineering, no. 3, pages 303–310, 1987. (Cité à la page 5.)

[Seacord 2003] Robert C Seacord, Daniel Plakosh and Grace A Lewis. Modernizing legacy systems: software technologies, engineering processes, and business practices. Addison-Wesley Professional, 2003. (Cité à la page 3.)

[Seemann 1998] Jochen Seemann and Jürgen Wolff von Gudenberg. *Pattern-based design recovery of Java software.* In ACM SIGSOFT Software Engineering Notes, volume 23, pages 10–16. ACM, 1998. (Cité à la page 90.)

[Selmadji 2018] Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde-Lilia Bouziane, Christophe Dony and Rahina Oumarou Mahamane. *Re-architecting OO Software into Microservices - A Quality-Centred Approach.* In Service-Oriented and Cloud Computing - 7th IFIP WG 2.14 European Conference, ESOCC 2018, Como, Italy, September 12-14, 2018, Proceedings, pages 65–73, 2018. (Cité aux pages 23, 30, and 32.)

[Shah 2013] Syed Muhammad Ali Shah, Jens Dietrich and Catherine McCartin. *On the automation of dependency-breaking refactorings in java.* In Software Maintenance (ICSM), 2013 29th IEEE International Conference on, pages 160–169. IEEE, 2013. (Cité aux pages 26 and 31.)

[Shahmohammadi 2010] Gholamreza Shahmohammadi, Saeed Jalili and Seyed Mohammad Hossein Hasheminejad. *Identification of System Software Components Using Clustering Approach.* Journal of Object Technology, vol. 9, no. 6, pages 77–98, 2010. (Cité à la page 4.)

[Shatnawi 2015] Anas Shatnawi, Abdelhak Seriai, Houari A. Sahraoui and Zakarea Alshara. *Mining Software Components from Object-Oriented APIs.* In Software Reuse for Dynamic Systems in the Cloud and Beyond - 14th International Conference on Software Reuse, ICSR 2015, Miami, FL, USA, January 4-6, 2015. Proceedings, pages 330–347, 2015. (Cité à la page 4.)

[Shatnawi 2016] Anas Shatnawi, Abdelhak-Djamel Seriai, Houari Sahraoui and Zakarea Alshara. *Reverse engineering reusable software components from object-oriented APIs.* Journal of Software and Systems(JSS), 2016. (Cité à la page 17.)

[Solms 2015] Fritz Solms. *A Systematic Method for Architecture Recovery.* In ENASE 2015 - Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering, Barcelona, Spain, 29-30 April, 2015., pages 215–222, 2015. (Cité à la page 4.)

[Sommerville 2011] Ian Sommerville*et al.* Software engineering. Boston: Pearson,, 2011. (Cité à la page 2.)

[Spiegel 2002] André Spiegel. *Automatic distribution of object oriented programs.* PhD thesis, Free University of Berlin, Dahlem, Germany, 2002. (Cité aux pages 27 and 31.)

[Stavropoulou 2017] Ioanna Stavropoulou, Marios Grigoriou and Kostas Kontogiannis. *Case study on which relations to use for clustering-based software architecture recovery.* Empirical Software Engineering, vol. 22, no. 4, pages 1717–1762, 2017. (Cité à la page 104.)

[Steimann 2006] Friedrich Steimann*et al.* *Decoupling classes with inferred interfaces.* In SAC, 2006. (Cité aux pages 26, 31, and 37.)

[Stoianov 2010] Alecsandar Stoianov and Ioana Şora. *Detecting patterns and antipatterns in software using Prolog rules.* In Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on, pages 253–258. IEEE, 2010. (Cité à la page 79.)

[Stroulia 2003] Eleni Stroulia, Mohammad El-Ramly, Paul Iglinski and Paul Sorenson. *User interface reverse engineering in support of interface migration to the web.* Automated Software Engineering, vol. 10, no. 3, pages 271–301, 2003. (Cité à la page 104.)

[Szyperski 1999] Clemens Szyperski, Jan Bosch and Wolfgang Weck. *Component-oriented programming.* Lecture Notes in Computer Science, 1999. (Cité à la page 7.)

[Szyperski 2002] Clemens A. Szyperski, Dominik Gruntz and Stephan Murer. Component software - beyond object-oriented programming, 2nd edition. Addison-Wesley component software series. Addison-Wesley, 2002. (Cité à la page 16.)

[Tamai 1992] Tetsuo Tamai and Yohsuke Torimitsu. *Software lifetime and its evolution process over generations.* In Software Maintenance, 1992. Proceerdings., Conference on, pages 63–69. IEEE, 1992. (Cité à la page 2.)

[Tavares 2008] André Luiz Camargos Tavares and Marco Tulio de Oliveira Valente. *A gentle introduction to OSGi.* ACM SIGSOFT Software Engineering Notes, vol. 33, no. 5, 2008. (Cité à la page 99.)

[Tilley 1995] Scott R Tilley and Dennis Smith. *Perspectives on legacy system reengineering.* 1995. (Cité à la page 4.)

[Tonella 2004] Paolo Tonella and Mariano Ceccato. *Aspect Mining through the Formal Concept Analysis of Execution Traces.* In 11th Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8-12, 2004, pages 112–121, 2004. (Cité à la page 24.)

[Tonella 2005] Paolo Tonella. *Reverse engineering of object oriented code.* In International Conference on Software Engineering (ICSE), 2005. (Cité à la page 53.)

[Tourwé 2003] Tom Tourwé and Tom Mens. *Identifying Refactoring Opportunities Using Logic Meta Programmin*. In 7th European Conference on Software Maintenance and Reengineering (CSMR 2003), 26-28 March 2003, Benevento, Italy, Proceedings, pages 91–100, 2003. (Cité aux pages 26 and 31.)

[Tourwé 2004] Tom Tourwé and Kim Mens. *Mining Aspectual Views using Formal Concept Analysis*. In 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004), 15-16 September 2004, Chicago, IL, USA, pages 97–106, 2004. (Cité à la page 24.)

[Tzerpos 1997] Vassilios Tzerpos and Richard C Holt. *The orphan adoption problem in architecture maintenance*. In Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on, pages 76–82. IEEE, 1997. (Cité à la page 29.)

[Tzerpos 1999] Vassilios Tzerpos and Richard C Holt. *MoJo: A distance metric for software clusterings*. In Reverse Engineering, 1999. Proceedings. Sixth Working Conference on, pages 187–193. IEEE, 1999. (Cité à la page 104.)

[Tzerpos 2000] Vassilios Tzerpos and Richard C Holt. *Acdc: An algorithm for comprehension-driven clustering. In wcre*, 2000. (Cité aux pages 29 and 32.)

[Van Vliet 1993] Hans Van Vliet, Hans Van Vliet and JC Van Vliet. Software engineering: principles and practice, volume 3. Wiley New York, 1993. (Cité à la page 5.)

[Vanciu 2013] Radu Vanciu and Marwan Abi-Antoun. *Object graphs with ownership domains: An empirical study*. In Aliasing in Object-Oriented Programming. Types, Analysis and Verification. Springer, 2013. (Cité à la page 71.)

[Wampler 2002] Bruce E Wampler. The essence of object-oriented programming with java and uml. Addison-Wesley, 2002. (Cité à la page 58.)

[Wang 2008] Lei Wang and Michael Franz. *Automatic partitioning of object-oriented programs for resource-constrained mobile devices with multiple distribution objectives*. In International Conference on Parallel and Distributed Systems (ICPADS), 2008. (Cité aux pages 28 and 32.)

[Washizaki 2003] Hironori Washizaki, Hirokazu Yamamoto and Yoshiaki Fukazawa. *A Metrics Suite for Measuring Reusability of Software Components*. In 9th IEEE International Software Metrics Symposium (METRICS 2003), 3-5 September 2003, Sydney, Australia, page 211, 2003. (Cité à la page 19.)

[Washizaki 2005] Hironori Washizaki and Yoshiaki Fukazawa. *A technique for automatic component extraction from object-oriented programs by refactoring*. Science of Computer programming, 2005. (Cité aux pages 15, 30, and 32.)

[Welker 2001] Kurt D Welker. *The software maintainability index revisited.* CrossTalk, vol. 14, pages 18–21, 2001. (Cité à la page 46.)

[Wettel 2011] Richard Wettel, Michele Lanza and Romain Robbes. *Software systems as cities: a controlled experiment.* In International Conference on Software Engineering (ICSE), 2011. (Cité à la page 28.)

[Wu 2005] Jingwei Wu, Ahmed E. Hassan and Richard C. Holt. *Comparison of Clustering Algorithms in the Context of Software Evolution.* In 21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary, pages 525–535, 2005. (Cité à la page 104.)