Migrating Object Oriented Applications into Component-Based ones

Zakarea Al-Shara

▶ To cite this version:

Zakarea Al-Shara. Migrating Object Oriented Applications into Component-Based ones. Other [cs.OH]. Université Montpellier, 2016. English. NNT : 2016MONTT254 . tel-01816975v1

HAL Id: tel-01816975
https://theses.hal.science/tel-01816975v1

Submitted on 15 Jun 2018 (v1), last revised 18 Jun 2018 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Collège Doctoral du Languedoc-Roussillon

THÈSE

Pour obtenir le grade de Docteur

Délivré par l'**UNIVERSITY OF MONTPELLIER**

Préparée au sein de l'école doctorale **I2S**
Et de l'unité de recherche **LIRMM**

Spécialité: **Software Engineering**

Présentée par **Zakarea AL SHARA**

Migrating Object-Oriented Software into Component-Based Ones

Soutenue le 17/11/2016 devant le jury composé de

Mr Flavio Oquendo	Prof.	Université de Bretagne-Sud	Rapporteur
Mr Hafedh Mili	Prof.	Université du Québec à Montréal	Rapporteur
Mr Thomas Ledoux	MCF	Ecole des Mines de Nantes	Examinateur
Mme Dalila Tamzalit	MCF	Université de Nantes	Examinateur
Mme Anne Laurent	Prof.	Université de Montpellier	Examinateur
Mr Christophe Dony	Prof.	Université de Montpellier	Co-Directeur de thése
Mr Abdelhak-Djamel Seriai	MCF	Université de Montpellier	Co-Directeur
Mr Chouki Tibermacine	MCF	Université de Montpellier	Invité
Mme Hinde Lilia Bouziane	MCF	Université de Montpellier	Invité

LIRMM

Université de Montpellier

Reuse
Component Based
Instantiation
Reuse JavaBeans
MetaModel
Java Fractal Component
Design Software Evolution
Model Driven
Design Legacy Software Component
Object Oriented EMF
Object Oriented Instantiation Fractal Service Oriented Architecture
CCM Reverse Engineering
Service Reverse Engineering
Service JavaBeans Software Migration Inheritance
Inheritance
Object Oriented
Software Maintenance
Exception Handling Model Driven
AST Software Architecture JavaBeans
CCM Java EJB AST C++
EJB Inheritance OSGi Legacy Software Service Oriented Architecture
Software Evolution Fractal
COM MetaModel Reuse
Patterns Software Transformation Interface Reverse Engineering Reuse Service Interface
AST Service Legacy Software Component
Design Java Reengineering EJB Patterns
OSGi
C++ Design Model Driven Interface EMF
Software Evolution EMF Software Migration
Software Architecture Ruls Patterns
C++ Design Legacy Software AST
EMF SOFA
CCM COM Fractal SOFA JavaBeans Service Reengineering
AST Interface Reengineering Instantiation JavaBeans
SOFA Understandability Component
EJB Software Modernization COM MetaModel Legacy Software Component
CCM
Fractal C++
Component Based Exception Handling Service Oriented Architecture COM
Interface MetaModel Reengineering Object Oriented Service SOFA
Java EMF Java Reverse Engineering Component Object Oriented Patterns
SOFA Software Migration C++ CCM
Reuse Understandability Software Modernization
Design OSGi Software Evolution SOFA
OSGi Software Architecture Component Based EJB Software Transformation Instantiation OSGi
Software Maintenance Understandability Exception Handling Instantiation Patterns Fractal Ruls Software Maintenance
Model Driven CCM Inheritance
Component Based Service Oriented Architecture Software Modernization
Ruls Java
C++ OSGi Ruls EJB
COM
Ruls COM
JavaBeans
Inheritance

Acknowledgments

After an intensive period of three years, today is the day: writing this note of thanks is the finishing touch on my thesis. It has been a period of intense learning for me, not only in the scientific arena, but also on a personal level. Writing this thesis has had a big impact on me. I would like to reflect on the people who have supported and helped me so much throughout this period.

Firstly, I would like to express my sincere gratitude to my advisor Dr. Abdelhak-Djamel Seriai for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study. Also it is with immense gratitude that I acknowledge the support and help of my advisers Dr. Chouki Tibermacine, Prof. Christophe Dony and Dr. Hinde Lilia Bouziane. They definitely provided me with advice and tools that I needed to choose the right direction and successfully complete my thesis. Without they precious support it would not be possible to conduct this research. Last but not least, I would like to thank Prof. Marianne Huchard for her supports.

Besides my advisers, I would like to thank the reviewers of my manuscript: Prof. Flavio Oquendo and Prof. Hafedh Mili for their insightful comments and encouragement, but also for the questions which incented me to widen my research from various perspectives. My sincere thanks also goes to the rest of my thesis committee: Dr. Thomas Ledoux, Prof. Dalila Tamzalit and Prof. Anne Laurent to be examiners of my thesis.

I thank my colleagues for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last three years. Besides my colleagues, I would like to thank my friends for their support , and for all the fun we have had in the last three years. We have spent beautiful moments in tours, parties, playing, cooking and love. I am very grateful to all the people I have met along the way and have contributed to the development of my life.

Finally, I would like to dedicate this thesis to my parents. I have been extremely fortunate in my life to have parents who have shown me unconditional love and support. Personally, my parents have played an important role in the development of my identity and shaping the individual that I am today. I would also like to thank my brothers and sister for supporting me spiritually throughout writing this thesis and my life in general.

Migrating Object-Oriented applications into Component-Based Ones

Abstract: Large object-oriented applications have complex and numerous dependencies, and usually do not have explicit software architectures. Therefore they are hard to maintain, and parts of them are difficult to reuse. Component-based development paradigm emerged for improving these aspects and for supporting effective maintainability and reuse. It provides better understandability through a high-level architecture view of the application. Thus, migrating object-oriented applications to component-based ones will contribute to improve these characteristics, and support software evolution and future maintenance.

In this dissertation, we propose an approach that automatically transforms object-oriented applications to component-based ones. More particularly, the input of the approach is the result provided by software architecture recovery: a component-based architecture description. Then, our approach transforms the object-oriented source code in order to produce deployable components. We focus on transforming object-oriented dependencies into interface-based ones. Moreover, we move from the concept of object to the concept of component instance. Furthermore, we provide a declarative transformation approach using domain-specific languages. We demonstrate our approach on many well-known component models.

Keywords: Component-based, Object-oriented, Software Migration, Reengineering, Reverse engineering, Transformation, Model-driven, Software evolution, Software maintenance, Software reuse, Design pattern.

Contents

1 Introduction **1**

1.1 Context . . . 1

1.1.1 Legacy Information Systems and Software Evolution . . . 1

1.1.2 Modernization of Legacy Systems . . . 3

1.2 Problem and Motivation . . . 4

1.2.1 Component-Based Software Engineering . . . 5

1.3 Contribution . . . 7

1.3.1 Recovering Component-based Architecture . . . 7

1.3.2 Transforming Object-oriented Dependencies into Interface-based Ones Using Design Patterns . . . 7

1.3.3 Reveal component Instance . . . 7

1.3.4 Model-Driven Software Migration: from Object-Oriented Models to Component-Based Models . . . 7

1.4 Structure of the Thesis . . . 8

2 Software Migration **9**

2.1 Reverse Engineering . . . 11

2.2 Transformation . . . 12

2.3 A Taxonomy for State of the Art . . . 13

2.3.1 Goal of Software Migration . . . 13

2.3.2 Input Source of Migration . . . 16

2.3.3 Reverse Engineering . . . 19

2.3.4 Transformation . . . 24

2.3.5 Direction of Transformation . . . 27

2.3.6 Output/Target of Migration . . . 29

2.4 Migrating Object-oriented Programs into Component-based ones . . . 29

2.4.1 Component-based Architectures . . . 29

2.4.2 Reconstruction Component-based Architectures . . . 32

2.4.3 ROMANTIC: an Approach for Recovering Component-based Architectures . . . 33

2.4.4 Running Example . . . 36

2.5 Discussion . . . 37

2.5.1 Input Source of Migration . . . 37

2.5.2 Reverse Engineering . . . 39

2.5.3 Transformation . . . 39

2.5.4 Output/Target of Migration . . . 40

2.5.5 Goal of Migration . . . 40

2.6 Conclusion . . . 40

3 Healing Component Encapsulation 43
3.1 Introduction . . . 43
3.2 Problem Statement . . . 44
3.2.1 Explicit component encapsulation violation . . . 44
3.2.2 Implicit component encapsulation violation . . . 44
3.3 Instance Handling Transformation . . . 46
3.3.1 Creating Object Interfaces: Uncoupling Boundary Classes . . 46
3.3.2 Using Component Interfaces through the Factory Pattern . . 47
3.4 Inheritance Transformation . . . 48
3.4.1 Replacing Inheritance by Delegation . . . 49
3.4.2 Handling Subtyping . . . 51
3.4.3 Dealing with Abstract Superclasses . . . 51
3.5 Exception handling transformation . . . 53
3.5.1 Transformation thrown exception . . . 53
3.5.2 Transforming exception handling . . . 54
3.6 Experimental Evaluation . . . 57
3.6.1 Experiment Design and Planning . . . 57
3.6.2 Results . . . 61
3.7 Conclusion . . . 69

4 Reveal Component Instance 71
4.1 Introduction . . . 71
4.2 Problem Statement . . . 72
4.3 Transforming OO Code to CB One . . . 73
4.3.1 Generating Component Descriptor and Reference of its Implementation . . . 73
4.3.2 Component Instantiation . . . 74
4.3.3 Reveal Component-based Architecture . . . 77
4.4 Mapping the Proposed Solution onto Component Models . . . 79
4.4.1 Mapping from Java to OSGi . . . 79
4.4.2 Mapping from Java to SOFA 2.0 . . . 81
4.5 Discussion . . . 82
4.6 Conclusion . . . 83

5 Model-Driven Object-Based to Component-based Software Migration 85
5.1 Introduction . . . 86
5.2 Transforming Object-oriented Applications into Component-based ones Using MDT: An Overview . . . 87
5.3 Transforming OOGM into CBGM: Defining the Source and the Target Metamodels and Rules of Transformation . . . 90
5.3.1 Metamodeling: Defining OOGMM and CBGMM . . . 90
5.3.2 Transforming OOGM to CBGM Rules . . . 98
5.4 Transforming CBGM into CBSMs . . . 103

5.4.1 Defining CBSMMs 103
5.4.2 Identifying the Variability of Transformation Rules 107
5.5 Implementation and Tools 112
5.6 Conclusion 116

6 Conclusion and Future Work 117
6.0.1 Summary of Contributions 118
6.0.2 Future Directions 119
6.0.3 Publications 121

Bibliography 123

List of Figures

1.1 Software maintenance and evolution 4

2.1 Process of software migration 10
2.2 Reverse engineering and transformation into software migration . . . 13
2.3 Taxonomy diagram for software migration 14
2.4 Object-to-component mapping model 34
2.5 Component quality measurement model 35
2.6 Information screen class diagram. 37
2.7 Architecture recovery for the information screen. 38

3.1 Flow of execution of exception-handling. 46
3.2 Transforming class instantiation based on the factory pattern. 48
3.3 Implementation of the delegation pattern at level of component. . . . 49
3.4 Replacing inheritance by delegation. 50
3.5 Handling abstract superclass based on proxy classes. 52
3.6 Transformation thrown exception by adapter. 53
3.7 Ideal exception handling model for component [Bennett 1982]. 54
3.8 Our derived exception handling model for component. 55
3.9 Transforming handling exception. 57
3.10 Relation between number of transformations with number of components. 63
3.11 The percentage of abstractness for each transformation type. 64
3.12 The mean of manual transformation time for each group. 68
3.13 The error percentage of manual transformation for each group. . . . 68

4.1 Different release of the same component instance 75
4.2 *Information-screen* architecture recovery and Darwin ADL for *DisplayedInformation* and *ContentProvider* 80

5.1 The phases of transformation process. 88
5.2 Models handled in the transformation process. 89
5.3 Model transformation. 90
5.4 The core of FAMIX metamodel. 91
5.5 The core of the object-oriented architecture recovery metamodel. . . 93
5.6 The core of the CBGMM. 94
5.7 Access attribute, object reference, method invocation in object-oriented architecture recovery metamodel. 95
5.8 Access attribute, object reference, method invocation in CBGMM. . 95
5.9 Inheritance relationship in object-oriented architecture recovery metamodel. 96

5.10 Inheritance relationship in CBGMM. 96
5.11 Exception handling in object-oriented architecture recovery metamodel. 97
5.12 Exception handling in CBGMM. 98
5.13 The order of the transformation problem into CBGM. 102
5.14 OSGi component metamodel. 103
5.15 SOFA component metamodel. 104
5.16 CCM metamodel. 105
5.17 Fractal component metamodel. 106
5.18 COM metamodel. 106
5.19 OpenCOM metamodel. 107
5.20 JavaBeans component metamodel. 108
5.21 EJB component metamodel. 108
5.22 Component-Based Generic Metamodel (CBGMM). 109
5.23 Declarative Service life cycle. 110
5.24 Feature model for specific transformation. 112
5.25 Migration process. 113

List of Tables

2.1 Goal of migration 17
2.2 Source of migration 19
2.3 Reverse engineering 23
2.4 Transformation 28
2.5 Target of migration 30

3.1 Data collection. 59
3.2 Information about people involved in the experiment. 61
3.3 Architecture recovery results. 61
3.4 Statistics of transformation types. 62
3.5 Improvement of abstractness after transformation. 63
3.6 Estimated time for manual transformation. 65

4.1 Object-based Component Model Specifications [Crnkovic 2011a] . . . 73
4.2 Composition type in object-based component models 83

5.1 Mapping between object-oriented language and IDL 116

Chapter 1
Introduction

Contents

1.1 Context **1**

1.1.1 Legacy Information Systems and Software Evolution 1

1.1.2 Modernization of Legacy Systems 3

1.2 Problem and Motivation **4**

1.2.1 Component-Based Software Engineering 5

1.3 Contribution **7**

1.3.1 Recovering Component-based Architecture 7

1.3.2 Transforming Object-oriented Dependencies into Interface-based Ones Using Design Patterns 7

1.3.3 Reveal component Instance 7

1.3.4 Model-Driven Software Migration: from Object-Oriented Models to Component-Based Models 7

1.4 Structure of the Thesis **8**

1.1 Context

1.1.1 Legacy Information Systems and Software Evolution

Many software systems live significantly longer than their developers had expected. These systems must be adapted constantly to changing situations and able to face numerous larger and smaller modifications during their lifetime. These software systems are designated as legacy systems and represent a particularly complex scenario of software maintenance and evolution. Consequently, this scenario is characterized by being time consuming, having high costs, design drift, using old programming and development languages, little or no documentation as well as few test cases. Nowadays not only old and monolithic systems, but also object-oriented applications (written in Java or C++) comply with legacy systems [Scalise 2010]. The author in [Sneed 1984] claims that systems older than five years already are legacy software systems.

Sommerville [Sommerville 2010] listed various issues and criteria that are distinctive features of a legacy system. He underlines that legacy software systems:

- do not have or only have outdated documentation.

- do not have or have few test cases to check its systems.
- no longer have available developers or users, hence orientation is time consuming and difficult.
- are expired, the third parties responsible for maintenance of hardware and software components are no longer available.
- have a very long time for compilation.
- do not have or have drifted knowledge of the internal architecture.
- are only understood to a limited extent.
- have many different versions making it difficult to distinguish their variants and hard to maintain.
- need a lot of time for small maintenance tasks.
- can not easily respond to change requests, since fixing errors has become a regular and permanent task.
- have a lot of duplicated code caused by ad-hoc reuse.
- have bad quality of the source code, errors, and many Code Smells.

From a business perspective, there are four different strategies for dealing with these legacy systems: completely replace the system, freeze the current state without further changes, hold on to maintenance despite cost or modernize these software systems by reengineering, migration, or maintenance with acceptable cost. However, the first three strategies are not suitable solutions when the cost and time are the main interest. Neither do they contribute to improving the state nor do they address the (main) causes of the problem. However, Reengineering and migration of data and their internal processing in particular are necessary requirements for both: the modernization of software and the new development [Bisbal 1999].

Reengineering and migration of software systems are also understood as mine and recovery of long-lost implicit information, as well as a bridge between this understanding and new knowledge. Since the beginning of software development, software archeology [Hunt 2002] is known and necessary. Incidental activities, depending on the complexity, are split into maintenance (minor works) and migration of software (extensive changes to a system caused by software evolution). First of all, the need for archaeological excavations within a software is because of the separation between historically developed of initial development and their maintenance. Secondly, it is due to the lack of synchronization between design artifacts and the program code. During an excavation, the Fossils found in the ground are often the only clue to the history of eras. Similarly, in legacy software systems, the source code is the only evaluable artifact of information. Their documents are usually drifted, outdated, useless or never existed. Therefore, the source code must be analyzed first to recover its meaning and function.

1.1.2 Modernization of Legacy Systems

Over the years, several different options have come into being for legacy modernization, each of them met with varying success and adoption. The most well known options are: software reengineering, software migration and new development.

Software reengineering is defined as an engineering process aiming to generate evolvable systems by Seacord et al. [Seacord 2003a]. In general, it includes all activities after the software is delivered to the customer to improve the understanding of the software and various quality parameters, for example the complexity, maintainability, extensibility and reusability. It is used to port a software system to a new platform, to extract knowledge and design, to break a monolith and to reduce the dependence from its developers. Therefore, it helps to extend the life time of the software system.

Reengineering is applied to both software maintenance and software evolution [Müller 2013]. However, scope, effort and use of various technologies differ considerably in these two areas. Evolution is based on the understanding of the whole software system. Therefore, it is expensive, while maintenance is usually limited to investigating localized problems. The generic term reengineering summarizes a variety of software techniques that are used to understand, to improve and to validate existing software [Wagner 2014].

Software migration is a variant of software reengineering in which the transformation is driven by a major technology change (e.g. migrating software written by procedural languages to object-oriented ones). Software migration is defined as analyzing, splitting and transforming software to a new platform or technology to meet new requirements and to improve future maintainability [Bisbal 1999]. The existing functionality must be preserved in order to prevent the loss of business knowledge.

The migration of software is designed exclusively for the field of software evolution. The terms maintenance and evolution must be clearly distinguished from each other. Weiderman et al. [Weiderman 1997] define maintenance as fine-grained activity that extends over a shorter period of time and involves local changes only. In addition, the structure of the system is only undergoing minor adjustments. In contrast, software evolution represents major changes, including the architecture and technology of the system. This will be urgent when the system falls short of the users satisfaction and expectations. This leads to new user requirements business areas.

Frequently, the software migration is associated with legacy software systems that have a complex and complicated restoration associated with it. As we said before, from an economic point of view, the demand to maintain the system and its inherent business logic is very high for legacy systems. However, these systems usually can not be shut down or redeveloped because of their central importance. The migration for software evolution meets the new demands and reduces the inner complexity. Thus it improves the quality parameters of the software. consequently, it can in turn lead to lower subsequent maintenance costs. The boundary between maintenance and evolution is naturally fluent. Figure 1.1 visualizes the relation-

ships just described. The maintenance, evolution, reengineering and migration of a system are illustrated. The curve (solid green line) describes the degree of requirements to be met by the system over time. The expected or required functionality is represented by a continuously increasing dash-dot line. If the gap between these two lines is too large, a system needs to be migrated.

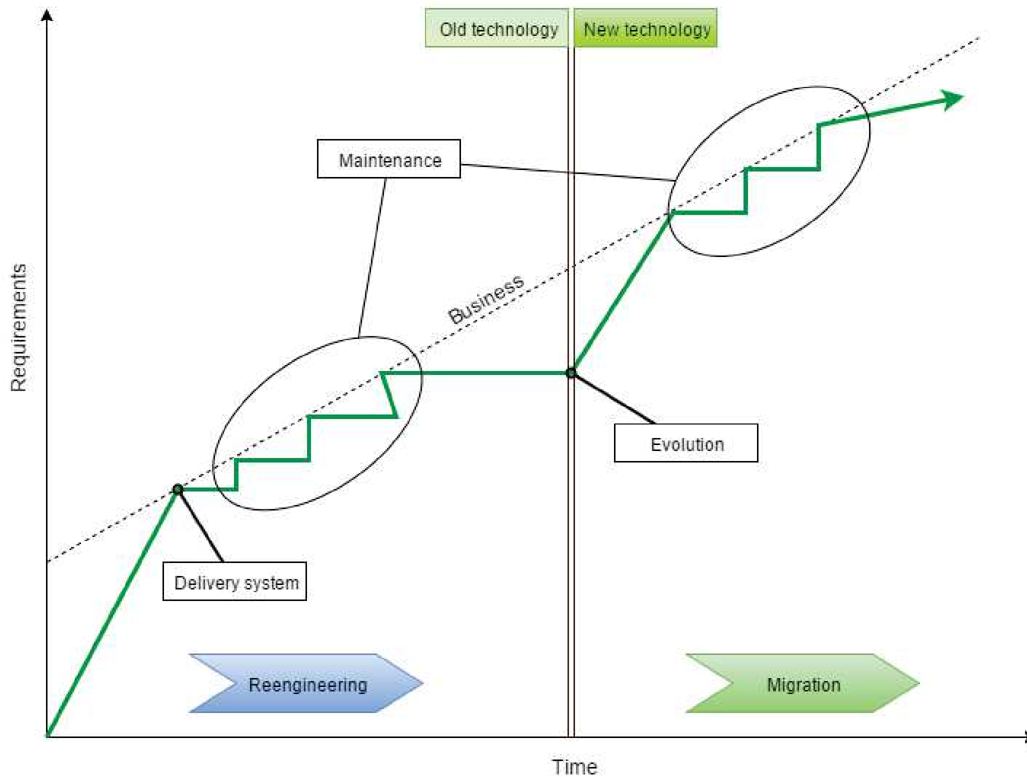


Figure 1.1: Software maintenance and evolution

The new development option is denoted in literature as Big Bang or Cold Turkey approach [Brodie]. A second and completely new software is developed in parallel to the existing one. The new development approach is correlated with huge costs (time, money and effort). Moreover, it is a risky approach where parts of the company may not be operational during the development of the new system. Consequently, the effect of new system could emerge: the over-development and over-specification of the new system [Brooks Jr 1995].

1.2 Problem and Motivation

Laws in the Software Life Cycle: The need for software evolution is due to the increasing complexity during the life cycle of software. Lehman and Belady [Lehman 1985] have formulated several laws of software evolution principle in the mid-eighties. The first law is called the law of continuous change: it states that

software will be used only if it is constantly adapted to the changing requirements. *"A large program that is used undergoes continuing change or becomes progressively less useful. The change process continues until it is judged more cost - effective to replace the system with a recreated version."* [Lehman 1985] The second principle refers to the inner complexity of software. Usually, the complexity of software increases by any change in the software. *"As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it."* [Lehman 1985] Until this day, these laws are still valid and useful, they define that any software in use is subject to evolution. Consequently, it continuously follows an increasing inner complexity.

Several empirical surveys determined the amount of software maintenance in the entire software life cycle [Müller 2013, Erlikh 2000]. These surveys infer that the percentage of software maintenance was between 50% and 80%. They point out that the lower bound of maintenance continues to rise, hence in [Seacord 2003a] estimates an effort of 90%. These studies determine the effort based on various criteria such as percentage of total budget, number of staff hours as well as estimated future expenses. They all highlight that the percentage of maintenance is extremely high and has steadily increased. Based on the principles of Lehman and Belady, it can be assumed that the ratio of maintenance is increased during the lifetime of software. Consequently, an existing software product needs to be expanded or improved. Therefore, the developer has to deal with legacy systems and to incorporate them. This emphasizes the need to focus on the evolution/maintenance phase.

1.2.1 Component-Based Software Engineering

Component Based Software Development has been recognized as a competitive principle methodology for developing modular software systems [Bertolino 2005a]. It is an approach to software development that relies on software reuse. It enforces the dependencies between components to be explicit through provided and required interfaces. Moreover, it provides coarse grained high-level architecture views for component-based applications. These views facilitate the communication between software architects and programmers during development, maintenance and evolution phases [Bertolino 2005b].

Besides, object-oriented software have fine-grained entities with complex and numerous implicit dependencies [Washizaki 2005]. Usually, they do not have explicit architectures or even have "drifted" ones. These adversely affect the software comprehension and make these software systems hard to maintain and reuse [Constantinou 2015]. Thus, migrating object-oriented software to component-based software should contribute to gain the benefits of component-based software engineering [Lau 2007].

The ultimate goal of our dissertation is supporting the evolution of legacy software. More specifically, migrating object-oriented applications into component-based ones[1]. In order to do this, we address the following research problems:

[1] The Service Component Architecture (SCA) is a set of specifications for building distributed

1. **Identifying component-based source code elements:** This problem can be divided into three sub problems:

   (a) **Identifying component-based architecture description:** CB architecture recovery aims to identify components and connectors from legacy software. The problem consists of identifying reusable components and its interfaces from legacy OO systems. A component is represented by a cluster of classes where its provided and required interfaces are represented by a set of provided and required methods respectively. The main challenge of this step is to find the best clusters compared to the component definitions which reflect the right software architecture.
   RQ1: *How to identify components and its interfaces?*

   (b) **Identifying component assembly description:** CB architecture recovery aims to identify components and connectors but not to create them. It does not transform these clusters of classes into a concrete component model. Moreover, the dependencies between clusters remain object-oriented ones. However, the dependencies between components should be through their provided and required interfaces. Therefore, we need to transform remain object-oriented dependencies to component-based ones (interface-based).
   RQ2: *How to transform object-oriented dependencies to interface-based ones?*

   (c) **Identifying component instantiation:** the recovered clusters should not be considered as simple packaging and deployment units. They should be treated as real components: true structural and behavior units that are instantiable from component descriptors and connected together to materialize the architecture of the software. Therefore, we need to reveal component descriptors, component instances and component-based architecture to materialize the recovered architecture.
   RQ3: *How to identify component instance and its descriptor?*

2. **Identifying the migration process:** The migration process from object-oriented application to component-based ones is a complex and difficult task. We need to provide a good support for the migration process by making it capable of being generic, extensible, coverable, reusable, integrated, declarative and automatic.
   RQ4: *How to describe the migration process?*

---

applications based on service-oriented architecture (SOA) and Component-Based Software Engineering (CBSE) principles [Beisiegel 2005, Beisiegel 2007]. SCA defines a component model for structuring service-oriented applications, where software components play as firstclass entities. Therefore, migrating into component-based contributes for migrating into SOA.

1.3 Contribution

1.3.1 Recovering Component-based Architecture

CB architecture recovery was largely dealt with in the literature [Ducasse 2009a, Birkmeier 2009, Kebir 2012, Chardigny 2008a, Allier 2010]. Most of these works aim to identify components as clusters of classes. They use clustering algorithms, among other techniques, aiming at maximizing intra-component cohesion and minimizing inter-component coupling to identify the architectural elements (components and connectors). Moreover, in our previous works [Kebir 2012, Chardigny 2008a], we have proposed an approach which aims to recover component-based architectures from OO source code. Therefore, we assume that the component-based architecture recovery problem is already solved. As a result, ***RQ1*** was already answered.

1.3.2 Transforming Object-oriented Dependencies into Interface-based Ones Using Design Patterns

To answer ***RQ2***, we propose an approach to automatically transform object-oriented applications to component-based ones. More particularly, the input of the approach is the result provided by software architecture recovery: a component-based architecture description. Then, our approach transforms the object-oriented source code in order to produce deployable components. We focus on transforming object-oriented dependencies between clusters (components) into component-based ones. More specifically, the approach transforms the source code related to instantiation, inheritance and exception handling dependencies between classes that are in different components into interface-based ones by using object-oriented design patterns.

1.3.3 Reveal component Instance

To answer ***RQ3***, we propose an approach for revealing component descriptors, component instances and component-based architecture to materialize the recovered architecture of an object-oriented software in component-based languages. The approach identifies component instances by inferring a component instance from a set of objects instantiated from its classes. After that, code generation and transformation are operated to materialize the recovered architecture.

1.3.4 Model-Driven Software Migration: from Object-Oriented Models to Component-Based Models

To answer ***RQ4***, we move onto describing our migration approach from code-centric approach to model-centric ones (models as first class entities). The metamodels for both object-oriented models and component-based models are identified. Then, we use a domain specific language transformation to describe the transformation rules between these models.

1.4 Structure of the Thesis

The rest of this thesis is organized into five chapters presented as follows:

- **Chapter 2** discusses the state-of-the-art related to the problem of legacy software modernization in the context of object-oriented, component-based and service oriented architecture.
- **Chapter 3** presents our contribution related to transforming object-oriented dependencies into interface-based ones using design patterns.
- **Chapter 4** presents our contribution aiming at revealing component descriptors, component instances and component-based architecture.
- **Chapter 5** presents our approach that aims at using model-driven transformation to transform object-oriented applications into component-based ones.
- **Chapter 6** reports conclusion remarks and future directions.

Chapter 2

Software Migration

Contents

2.1 **Reverse Engineering** **11**
2.2 **Transformation** **12**
2.3 **A Taxonomy for State of the Art** **13**
2.3.1 Goal of Software Migration 13
2.3.2 Input Source of Migration 16
2.3.3 Reverse Engineering 19
2.3.4 Transformation 24
2.3.5 Direction of Transformation 27
2.3.6 Output/Target of Migration 29
2.4 **Migrating Object-oriented Programs into Component-based ones** **29**
2.4.1 Component-based Architectures 29
2.4.2 Reconstruction Component-based Architectures 32
2.4.3 ROMANTIC: an Approach for Recovering Component-based Architectures 33
2.4.4 Running Example 36
2.5 **Discussion** **37**
2.5.1 Input Source of Migration 37
2.5.2 Reverse Engineering 39
2.5.3 Transformation 39
2.5.4 Output/Target of Migration 40
2.5.5 Goal of Migration 40
2.6 **Conclusion** **40**

In this chapter, we discuss the-state-of-the-art related to software migration and its main two steps; reverse engineering and transformation. We start by positioning our work compared to the related domains in Section 2.1 and Section 2.2. Then, a taxonomy of proposed transformation approaches are presented in Section 2.3. An example of a reverse engineering approach is presented in Section 2.4. After-that, the taxonomy results are discussed in Section 2.5. Finally, the conclusion of the chapter is presented in Section 2.6.

Legacy software systems are often built on obsolete and inefficient platforms that are difficult to maintain and enhance. Software migration is defined as reverse engineering and transforming software to a new platform or technology while reserving their existing functionalities to prevent the loss of business knowledge. The goal is (i) to reuse these software systems or part of them, (ii) to meet new requirements coming from software evolution and (iii) to improve future maintainability.

Bisbal et al [Bisbal 1999] define software migration as follows:
"Legacy Information System Migration (...) allows legacy systems to be moved to new environments that allow information systems to be easily maintained and adapted to new business requirements, while retaining functionality and data of the original legacy systems without having to completely redevelop them."

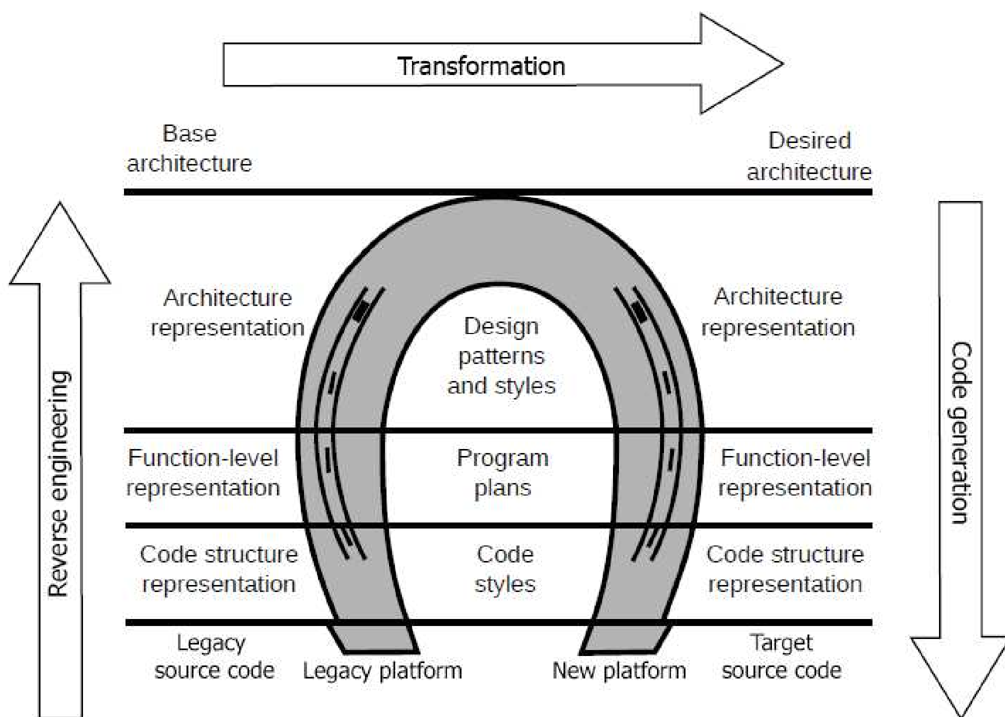


Figure 2.1: Process of software migration

Figure 2.1 extends the graphic of *Horseshoe Model* proposed in [Kazman 1998] and explains the developed methodology of software migration. On the left, the reverse engineering from the code-centric to high abstract model is depicted. On the other side, in the upper part and right part, transformations from legacy artifacts to new ones are depicted. Furthermore, it is illustrated that the target source code is generated from the high abstract models. Therefore, we consider the migration process is composed of two main steps, reverse engineering and transformation.

We consider code generation as a part of transformation step. By looking at the horseshoe from the bottom to the top, it can be noted how the migration proceeds at different levels of abstraction: code representation, function representation, and architecture representation.

2.1 Reverse Engineering

In the context of software engineering, the term *reverse engineering* was defined by [Chikofsky 1990] as *"the process of analyzing a subject system to (i) identify the system's components and their inter-relationships and (ii) create representations of the system in another form or at a higher level of abstraction"*. Also it was defined by [IEE 1998] as *"The process of extracting software system information (including documentation) from source code"*. Based on these definitions, reverse engineering has been viewed as a process of two steps: information extraction and abstract representation [CanforaHarman 2007]. Information extraction analyzes the subject software artifacts for mining raw data, whereas abstract representation creates user-oriented documents and views based on these raw data. For instance, information extraction step consists of extracting call graphs, metrics or facts from source code. Abstract representation aims at recreating design abstractions from the source code, existing documentation, human experts, knowledge and any other source of information. Its outputs can be design artifacts, software architectures, feature models, or business objects.

The source code of many software systems is the only reliable representation particularly for large long-lived software systems that have undergone many modifications during their lifetime [Chikofsky 1990]. Therefore, these software systems have substantially overgrown the initial supposed software systems and have drifted documentation. Hence, reverse engineering is recommended as a technique for redocumenting these software systems to obtain comprehensibility and maintainability [Chikofsky 1990, Vliet 2008]. However, the source code is not necessarily the only targeted artifacts for reverse engineering, any existing artifacts are candidates, for example, documentation, requirement, test case, manual pages, design diagrams or comments in a version control systems, etc. [CanforaHarman 2007]. Since the correlation between documentation and source code is not harmonized, the source code often considers the only reliable source of information [Chikofsky 1990]. This is certainly the main motivation for many reverse engineering techniques that deal with source code. The result of reverse engineering is the basis for the transformation and forward engineering steps [Kazman 1998]. Thus, reverse engineering is the foundation of the software migration and reengineering; a software system can only be changed when it is understood [Chikofsky 1990].

Reverse engineering approaches can have two major objectives: redocumentation and design recovery [CanforaHarman 2007]. Redocumentation aims at creating or revising alternate views of a given artifact at the same level of abstraction (e.g., pretty formatting source code or visualizing control flow graph). Design recovery

aims at recreating the design of abstract artifacts from the source code, existing documentation, human experts, knowledge and any other source of useful information [Biggerstaff 1989].

During the nineties, the prevalent of object-oriented languages and their advantages led to the need to reengineer existing procedural software towards object-oriented. Therefore, many reverse engineering approaches were developed to identify objects from legacy procedural software (e.g. [Cimitile 1999]). They focused on recovering high-level architectures or diagrams from procedural source code.

During the nineties and the early part of the new millennium, a new age of software development has emerged. Component-based software engineering, also known as component-based development, has emerged as a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems. Bearing this in mind, many reverse engineering approaches have been proposed to recover component-based architecture and their elements from object-oriented software systems. These approaches aim at recovering software architectures by identifying their components and the relationships (connectors) between these components.

Architecture recovery is important to improve software understandability, to promote software reuse, and to support software evolution. The most common techniques for architecture recovery depend on metrics-based [Mitchell 2006], formal concept analysis [Siff 1999], clustering metrics and heuristic-based techniques for component recovery [Kebir 2012]. One of the main problems to deal with when recovering an architecture is to capture differences between the source code and the mental model behind high-level artifacts.

2.2 Transformation

Strictly speaking, reverse engineering does not include transformation, which is the transformation from one representation form to another one (e.g., source-to-source or model-to-source transformations) nor software transformation, which encompasses the alteration of legacy software systems to reconstitute them in new forms.

Kleppe et al. [Kleppe 2003] define model transformation as follows: *A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.* The transformed model is required to be semantically equivalent to the subject model (transformed model is a refinement of subject model). In few cases, the result of transformation may be semantically different from the subject model but in predictable ways. A model can be source code or any other form of representation and can also be applicable to multiple source models and/or multiple target models. Code transformation is converting the source code into a machine readable form of target platform.

A practical requirement for source code transformation is reverse engineering of source code (e.g. source code parsing, building internal program representations of code structures like abstract syntax tree, the meaning of program symbols, etc.). After that, the transformation rules and patterns are applied onto subject source code and guided by the results of reverse engineering to obtain transformed source code representations. Finally, regeneration of valid source code from these representations as a target source code. Figure 2.2 illustrates the relation between reverse engineering and transformation into software migration.

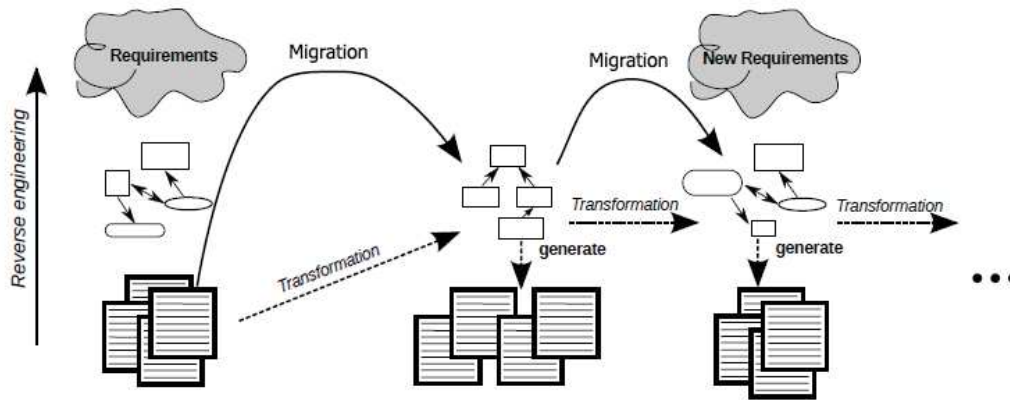


Figure 2.2: Reverse engineering and transformation into software migration

2.3 A Taxonomy for State of the Art

In this section, we propose a taxonomy for software migration that allows us to group approaches (methods, tools, techniques or formalisms) for software migration based on their common qualities. The taxonomy provides a multi-dimensional classification that allows us to group and compare software migration approaches based on the criteria of interest. The taxonomy consist of five dimensions: The goal of software migration, the input of migration, reverse engineering, transformation, and the output of migration. Figure 2.3 illustrate these diminutions and their qualities.

2.3.1 Goal of Software Migration

- **Reusability:** *ISO/IEC 25010:2011* defines reusability as *"degree to which an asset can be used in more than one system, or in building other assets"*. It is a property of a software artifact that indicates its extent of reuse. Software reuse refers to the uses of existing software artifacts for building new software systems to improve software quality and productivity [Frakes 2005] [Shiva 2007] [Leach 1997]. It can be over different levels of abstraction (i.e., requirement, design and implementation), and can concern different software

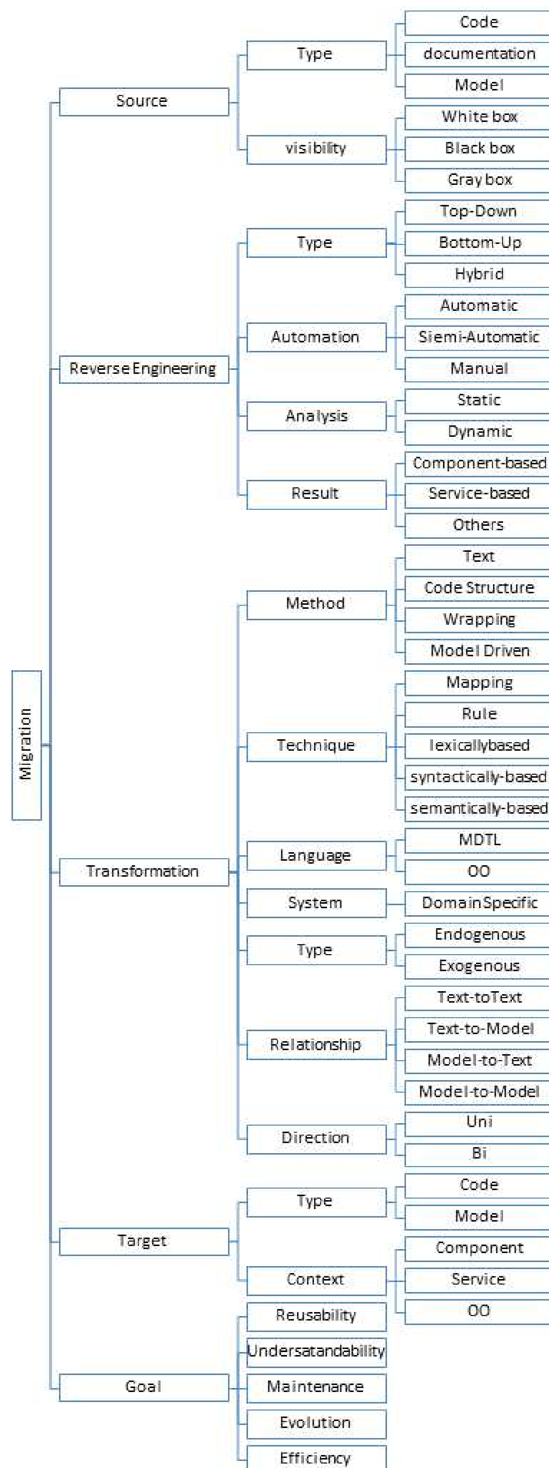


Figure 2.3: Taxonomy diagram for software migration

artifacts (e.g., feature, documentation, software architecture, software component, source code, libraries, design, etc.) [Frakes 1996] [Leach 1997].

Systematic reuse is a strategy for increasing productivity and improving quality in the software industry [Tomer 2004] [Frakes 1996]. The reusing of existing artifacts that are already tested, evaluated and proven in advance enhances the quality of software [Jacobson 1997]. And reusing preexisting software artifacts instead of developing them from scratch improves the software productivity. Consequently, it reduces development cost and the time to market [Frakes 2005, Mohagheghi 2007].

Since the first time systematic software concept was presented by [M. D. McIlroy 1968] , many software reuse approaches have been proposed to reach a potential degree of software reuse [Frakes 2005, Shiva 2007]. For examples, Component-Based Software Engineering (CBSE) [Heineman 2001a], Software Product Line Engineering (SPLE) [Clements 2002], service-oriented software engineering [Stojanović 2005] and aspect-oriented software engineering [Jacobson 2004], etc.

- **Understandability:** According to *ISO/IEC 9126-1*, understandability is a sub-characteristic of usability, and it is defined as the capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use [Fenton 2014]. The understandability sub-characteristic allows to draw conclusions about how subject users can recognize the logical concepts of software and its applicability [Stevanetic 2015]. It correlates with metrics which measure attributes of software that are responsible for the users' efforts for recognizing the logical concepts and applicability. Users should be able to select a software product which is suitable for their intended use. In other words, we can see understandability as the extent to which the user can comprehend the software.

  Usually, maintenance developers should not have less knowledge than the developers of the original system. He needs to make the system his own for a successful maintenance process. Large software systems generally consist of a landscape of different software products that are developed or bought and grown over many years and by different owners. The concept of system understanding is brought to examine the connection and the interaction between different products. The result is an overview (knowledge) of the functions and the role of the subsystems: system understanding [Thomas 2001].

- **Maintainability:** *ISO/IEC 25010:2011* defines maintainability as *degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers.* In addition, the ISO 9126 standard defines maintainability as the capability of the software product to be modified, including corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. Moreover, it proposed that the maintainability consists of the following charac-

teristics: analyzability, changeability, stability, and testability. These characteristics can rely on quality indicators such as coupling, cohesion, and complexity [Li 1993, Smith 2011, Zou 2002].

- **Evolution:** It is the process of developing a software and repeatedly updating or modifying it for various reasons during the software life cycle [Lehman 1980]. The software evolution is consider the longest phase in the software life cycle [Lehman 1980, Mens 2008]. It begins with the delivery of the application to the customer and until the system is dead or replaced. Lehman et al. formulated a series of software evolution laws [Lehman 1980, Lehman 1997]. The laws describe a balance between factors driving new developments and that slow down progress. Modern development technologies (e.g software product line, component-based development) can help to minimize the problems that inevitably arise in the context of software evolution.
- **Efficiency:** *ISO/IEC 25010:2011* defines efficiency as *resources expended in relation to the accuracy and completeness with which users achieve goals.* Inefficient causes are often found in bad architectural and violations of coding practice [Jones 2008, Charnes 1978]. These causes can be detected by measuring the static quality attributes of an application that predict potential operational performance bottlenecks (i.e. requiring high execution speed) and future scalability problems (i.e. huge volumes of data) [Fenton 2014, Jones 2008].

2.3.2 Input Source of Migration

2.3.2.1 Source Type

- **Source Code:** source code is taken to mean a collection of computer executable instructions that are describing a software system [Harman 2010]. It includes machine code, very high level languages and executable graphical representations of systems. The instructions are written usually by developers in a computer programming language, such as Java, C++, and C.

  Object-oriented source code mainly consists of a set of classes that belong to a set of organized packages [Rumbaugh 1991a]. Methods and attributes constitute the main building units of a class. The dependencies between classes are recognized via method invocations, access attributes, inheritance, exception handling, etc. Source code is the most software artifact that is commonly used by the existing migration approaches, due to its availability and reliability [Chikofsky 1990]. In the case of migrating object-oriented applications to component-based ones, the approaches start by analyzing and recognizing the dependencies between classes in order to identify high cohesive and low coupled clusters. After that, the transformation process takes place to transform these clusters to components [Wagner 2014]. For example, [Akers 2004], [Poch 2009] and [Allier 2011] proposed approaches that migrate object-oriented applications into component-based ones by having source code as input artifacts.

Table 2.1: Goal of migration

Approach	Reusability	Understandability	Maintenance	Evolution	Efficiency
[Akers 2004]	√			√	
[Boshernitsan 2006]				√	√
[Ding 2011]		√			
[Wang 2006]				√	√
[Xue 2011]	√				
[Clavreul 2010]	√				
[Nguyen 2014]	√		√		
[Santos 2015]			√		
[Bruneliere 2010]	√	√	√		√
[Fuhr 2011]	√	√		√	
[Ahmad 2014]	√	√	√		
[Winter 2007]				√	
[Matos 2011]				√	
[Selim 2013]				√	
[Hunold 2009]				√	
[Poch 2009]		√	√		
[Karim 2014]				√	
[Hunold 2008]					√
[De Lucia 2008]	√			√	
[Ying 2013]					√
[Binkley 2006]			√		
[Seriai 2014]		√			
[Canfora 2008]			√		
[Tilevich 2009]				√	√
[Kegel 2008a]					√
[Allier 2011]	√	√	√		
[Kapur 2010]			√		
[Eysholdt 2010]					√
[Axelsen 2012]					√
[Einarsson 2012]			√		
[Kjolstad 2011]			√	√	
[Gligoric 2014]				√	

- **Documentation:** Software documentation is text, model or illustration that accompanies computer software to explain how it operates or how to use it [Pohl 2010]. For example, It could be requirements documentation like use cases or design documentation like software architecture and class diagrams. Documentation is recommended to improve development and help maintenance. In the context of reengineering, redocumenting legacy software is useful to continue to maintain them or migrate them to new platforms [Forward 2002]. Moreover, Documentations are used as truth based for the reverse engineering approach.

- **Metamodel:** In general, it is a model that is used to describe another model by using a modeling language.

  According to OMG standards, *"A metamodel is a special kind of model that specifies the abstract syntax of a modeling language. It can be understood as the representation of the class of all models expressed in that language. Metamodels in the context of MDA are expressed using MOF."*

  According to Mellor et al [Mellor 2004], *"A metamodel is a model of a modeling language. The metamodel defines the structure, semantics and constraints for a family of models."*

  According to Clark et al [Clark 2008, Clark 2015], *"A metamodel is a model of a language that captures its essential properties and features. These include the language concepts it supports, its textual and/or graphical syntax and its semantics (what the models and programs written in the language mean and how they behave)."*

  Both source and target metamodels are provided for migration. Models confirm to source metamodel (instances of source metamodel) are also provided. Then the migration approach is responsible for generating target models that must confirm to the target metamodel. The advantages of using metamodels that are that they are technology independent. Moreover, it can be language-independent like FAMIX [Ducasse 2011a].

2.3.2.2 Source Visibility

We can classify the source of migration into three types based on their visibility. Whitebox, blackbox and graybox refer to the visibility of an implementation but not its interface [Szyperski 2002].

- **Whitebox:** In a whitebox software artifact, the source of the artifact is fully available for clients. Thus it can be studied and analyzed to enhance the understanding of what the abstraction does. Whitebox software reuse refers to using a software through its interfaces by relying on the understanding gained from studying and analyzing the actual implementation. For example, class libraries and frameworks are delivered in source form, and developers study the classes implementation to reuse or inherit them.

Table 2.2: Source of migration

Approuch	Type			Visibility		
	Code	Document	Model	White box	Black box	Gray box
[Akers 2004]	C++			√		
[Boshernitsan 2006]	Java			√		
[Ding 2011]		Use cases		√		
[Wang 2006]	C++			√		
[Xue 2011]	x			√		
[Clavreul 2010]	x				√	
[Nguyen 2014]	Java			√		
[Santos 2015]	OO			√		
[Bruneliere 2010]	Java, J2EE	Use cases	XML	√		
[Fuhr 2011]	Java			√		
[Ahmad 2014]	OO			√		
[Winter 2007]	Java			√		
[Matos 2011]	OO			√		
[Selim 2013]			General Motors Models		√	
[Hunold 2009]	OO			√		
[Poch 2009]	Java			√		
[Karim 2014]	JavaScript			√		
[Hunold 2008]	Java			√		
[De Lucia 2008]	ACUCOBOL-GT			√		
[Ying 2013]	AJAX-XML			√		
[Binkley 2006]	Java			√		
[Seriai 2014]	Java			√		
[Canfora 2008]	OO	Use cases + FSA			√	
[Tilevich 2009]	Java			√		
[Kegel 2008a]	Java			√		
[Allier 2011]	Java			√		
[Kapur 2010]	Java			√		
[Eysholdt 2010]	XML/UML			√		
[Axelsen 2012]	Core Package Templates			√		
[Einarsson 2012]	UML Model			√		
[Kjolstad 2011]	Java mutable class			√		
[Gligoric 2014]	Build Script			√		

- **Blackbox:** In an ideal blackbox software artifact, clients do not know any details about the source of the artifact beyond the interface and its specification. Blackbox reuse refers to the concept of reusing implementations without relying on anything but their interfaces and specifications. For example, application programming interfaces (APIs) do not reveal anything about the underlying implementation. They can only be reused through their interfaces.

- **Graybox:** In a graybox artifact, the source of the artifact is partially available for clients (reveal a controlled part of their implementation). The interface may still enforce encapsulation and limit what clients can do, although implementation inheritance allows for substantial interference. For example, in applications that have graphical user interfaces (GUIs), implementation of their GUIs could be available but not others.

2.3.3 Reverse Engineering

2.3.3.1 Type of Reverse Engineering

Basically, there are two different methods for the investigation of software: Top-down and Bottom-up [Szyperski 2002, Waters 1994]. In addition, combinations of the two methods are possible (Hybrid).

- **Top-down:** The study of the high level software artifacts (e.g. software architecture, use cases, etc.) starts with a look from the top. After that, The initial results are refined from top to bottom with the help of various hypotheses to identify low level software artifacts. Examples of using this kind of method are proposed by [Ding 2011] and [Canfora 2008]. In [Ding 2011], the authors used use cases to recover a service-oriented architecture from execution traces. In [Canfora 2008], the approach starts analyzing use cases and finite state automation to identify services form object-oriented source code. Use cases are used to extract the rules of interactions between users and legacy software systems. Then, a wrapper is implemented as an interpreter of a finite state automaton that encapsulates these interaction rules.

- **Bottom-up:** This method starts with the low level software artifacts (source code). It extracts useful information, structures and other interesting artifacts which are necessary to form an abstract representation at the next higher level (e.g. software architecture). For example, the approach proposed in [Akers 2004] analyzes source code that is written in C++ to extract components. In [Wang 2006], the authors targeted C++ applications to extract components using the cluster technology. Component interfaces and clusters of the components are demonstrated on J2EE distribute environment.

- **Hybrid:** It is the combination of the two previous methods; top-down and bottom-up. This method uses the top-down approach for system understanding; high level software artifacts of a software system is sufficient to recognize their relationships. The bottom-up method is essential for a detailed analysis of low level software artifacts. The approach proposed in [Bruneliere 2010] applied bottom-up method on Java applications source code and top-down method on their use cases. It extracts metrics, models and graphs from these applications. In [Eysholdt 2010], the authors used XML that represents a source code as low level artifacts and UML as high level artifacts.

2.3.3.2 Automation

- **Automatic:** The fully automated approaches do not need any human experts. Usually, fully automatic approaches compromise their precision or recall and the obtained information is only partially usable by the maintainer [CanforaHarman 2007]. Actually, there are no one hundred percent automated approaches, but we consider approaches that do not have a high impact of human experts on their results as automatic ones.

- **Semi-automatic:** It requires inputs from human experts to complement or correct the information that is automatically extracted. Semi-automatic approaches have used human experts feedback to improve the extracted artifacts and views, not the production process itself. Future activities in reverse engineering should push towards learning from human experts feedback to

automatically produce results by using machine learning, meta-heuristics and artificial intelligence [CanforaHarman 2007]. In other words, human experts feedback should be fully integrated into the software migration and reverse engineering to gain the benefits of both automatic and semi-automatic approaches.

- **Manual:** Manual approaches fully depend on human experts. These ones only provide guidelines and rules for human experts that allow them to identify elements and transform them to the target.

2.3.3.3 Type of Analysis

- **Static analysis:** It examines program source code over all possible behaviors that might arise at run time without actually executing the program, no matter on what inputs or in what environment the program is run [Ernst 2003]. Software metrics and reverse engineering are extensively using this type of analysis. For example, many works have been proposed to identify components from object-oriented source code using static analysis to obtain cluster of classes that have high cohesion and low coupling. Static analysis operates by building a model of the situation of the program, then determining how the program reacts to this situation [Ernst 2003]. A program has many possible executions, then the analysis must keep track of multiple different possible situations. However, it is usually not reasonable to reflect every possible run-time situation of the program; for example, there may be arbitrarily many different states or user inputs at runtime. Moreover, some information can not be obtained statically like dynamic binding but at run time [Rumbaugh 1991b]. Therefore, static analyses usually use an abstracted model of program situations that loses some details and information. As a result, the static analysis output may be less precise than the ideal ones.

- **Dynamic analysis:** It is observing the executions of a program by executing programs on a real or virtual processor (e.g testing and profiling)[Ernst 2003]. Dynamic analysis operate on a program with high code coverage has been more precise because the analysis can examine the actual and exact run-time behavior of the program[Chen 2002]. The program must be executed with sufficient test inputs to produce interesting behavior and maximize its code coverage. The results obtained by dynamic analysis are not comprehensive, they may not generalize to future executions. That is because there is no guarantee that the test suite over which the program was run (e.g use cases) is covering of all possible program executions (code coverage)[Ernst 2003].

- **Hybrid static-dynamic analysis:** Static and dynamic analysis can be applied to a single problem to complement and support one another. They can enhance one another by providing information that would otherwise be unavailable. For example, to identify all possible dependencies in object-oriented

source code, The static analysis can identify static dependencies like method invocations, inheritance and exception handling. Then dynamic operate in its turn to identify dynamic dependencies like ones related to dynamic binding and dependency injection.

2.3.3.4 Result of Reverse Engineering

The core of reverse engineering consists of extracting information from legacy software to represent it in abstract ways which are more easily understandable by humans. These abstract ways mean design recovery from the source code, existing documentation, experts knowledge and any other source of information. Based on the types of reverse engineering results, we study three types of the design recovery related works; Component-based, service-based, and others.

- **Component-based:** The key aspect of this is analyzing legacy software systems to identify components of them and inter-relationships between these components. Representations of these systems are created at a higher level of abstraction (Component-based architecture). Component-based architecture recovery aims at identifying components and the relationships (connectors) between them from legacy software based on specific techniques such as metrics-based, Formal Concept Analysis, clustering metrics and heuristic-based. Usually, a component identified as a cluster of classes that have high cohesion and low coupling with other clusters.

- **Service-based:** This type of results are used for migrating legacy systems towards service-oriented architecture. A service Oriented Architecture (SOA) can be viewed as a set of components that are separating their interfaces from their internal implementation, and their interface descriptions can be published and discovered [Haas 2004]. A key aspect of this is identifying software services from legacy software systems. A service is a unit of functionalities that is used (provided) to achieve desired end results for a service consumer with the policies that should control its usage. Service-oriented architecture is an architectural software concept that defines the use of services in a standardized way.

- **Others:** In addition to extract high level abstraction in contexts of component and service, other abstractions and information can be obtained as mediators to complete the migration process. These can be models (feature model, metamodel, model), trees and graphs (dependency graph, AST, call graph), patterns, templates, quality metrics, etc. A combination of all previously mentioned is possible. For example, many approaches, like the one proposed in [Kebir 2012], aim to recover component-based architectures based on a fitness function. The fitness function is based on quality measurements (e.g. coupling and cohesion). Therefore, quality measurements behind component-based architectures are extracted by these approaches.

Table 2.3: Reverse engineering

Approuch	Type			Automation			Analysis		Results		
	Top-Down	Bottom-Up	Hybrid	Automatic	Simi-Automatic	Manual	Static	Dynamic	Component-based	Service-based	Others
[Akers 2004]		√		√			√		√		
[Boshernitsan 2006]		√			√		√				Patterns
[Ding 2011]	√			√			√			√	
[Wang 2006]		√			√		√		√		Java
[Xue 2011]			√	√			√				Feature Model
[Clavreul 2010]		√			√		√				Meta Model
[Nguyen 2014]		√		√			√				Model
[Santos 2015]					√		√				Patterns
[Bruneliere 2010]			√	√			√				Metrics, Models, Graphs
[Fuhr 2011]		√		√			√	√		√	Graphs
[Ahmad 2014]		√			√		√			√	
[Winter 2007]		√		√			√				Patterns
[Matos 2011]		√		√	√		√			√	Graphs
[Selim 2013]		√		√			√				Model
[Hunold 2009]		√		√			√				Patterns
[Poch 2009]		√					√		√		
[Karim 2014]		√		√			√		√		
[Hunold 2008]		√		√			√				Patterns
[De Lucia 2008]			√	√			√			√	
[Ying 2013]		√		√			√				AST
[Binkley 2006]		√			√		√				Concerns
[Seriai 2014]		√		√				√	√		Call Graph
[Canfora 2008]	√			√			√			√	
[Tilevich 2009]		√		√				√			Partition Program
[Kegel 2008a]		√		√			√				Type Constraints
[Allier 2011]		√		√				√	√		
[Kapur 2010]		√		√			√				Differantial Code
[Eysholdt 2010]			√	√			√				Ecore Model
[Axelsen 2012]		√				√	√				Template
[Einarsson 2012]		√		√			√				Model
[Kjolstad 2011]		√		√			√				Graph
[Gligoric 2014]		√		√				√			Dependency Graph

2.3.4 Transformation

2.3.4.1 Transformation Method

- **Text:** Source code are the main and only artifacts that are used during transformation. It is not represented in other formats (e.g. XMI format, graphs, trees, models, etc.). The transformation is written in a classical programming languages (e.g. Java, C++).

- **Code structure:** An intermediate structural representation of the source code used to deal with it during transformation. Toolkits are used to enable the parsing of source code to produce structural documents. For example, we can represent each class as an abstract syntax tree [Pfenning 1988]. Then the transformation deals with these abstract syntax trees to complete its task.

- **Wrapping:** This method uses a thin layer of code which wraps legacy source code, the interactions with surrounding environments will be through this layer [Sneed 2000]. It helps two incompatible interfaces to work together, for example, the adapter pattern is a simple wrapper that allows the interface of an existing class to be used as another interface without modifying its source code[Freeman 2004]. The advantage of wrapping methods is that software components can be reused with lowest error prone because they have been known and tested for years. It is suitable for migrating black-box source code (black-box modernization), it requires merely knowledge of the external interfaces [Seacord 2003b]. Unfortunately, this method has only a short term solution character, the original problems are not necessarily solved. Moreover, typical problems occur like data persistence, thread and synchronization, as well as exception and error handling.

- **Model driven:** Defines the process of converting source models into target models by using standard model representation (e.g. meta-model) [Jouault 2008]. It relies on models as first class entities, the source and target models must be presented in a standard model. They are far more abstract than the first three methods and do not require detailed knowledge about structured elements, interfaces, or their representation at the model level.

2.3.4.2 Transformation Technique

- **Mapping:** A mapping transformation between source and target models consists of a set of declarative relations that should hold between their domains [Jouault 2008]. A domain is an element of a specific type and has a pattern, a set of properties and conditions. Such a relation is defined by two (or more) domains and a couple of pre and post conditions. Mapping Technique is bi-directional transformation: the transformation can be in the reverse direction. That means the source model can be the target one and the target model can be the source one.

- **Rule-based:** The transformation between source and target models is defined as a sequence of imperative rules[Porres 2003]. It is usually used to build target models of a complex structure. In addition, when there is no direct correspondence between individual elements of the source and target models, then it is difficult to describe the transformation declaratively[Visser 2005]. Unlike the mapping technique, rule-based is uni-directional, it does not support the transformation in the reverse direction.

- **lexically-based:** It is a simple technique based on search-and-replace features like traditional rename refactoring tools presented in IDEs [Kapur 2010]. This type is simple to the point that it can not discrimination references from declarations.

- **syntactically-based:** Syntactically-based transformation approaches based on syntactic representation of the program [Kapur 2010]. For example, we can parse classes and store all foundations of their syntactical information in abstract syntax trees (ASTs) [Pfenning 1988]. After that, a transformation is applied based on these ASTs like, the transformation could be used to locate references and refactor them.

- **semantically-based:** This technique demands the presence of formal specifications of the source code [Kapur 2010]. The formal specifications tend to be absent in industrial settings and require a mathematical expertise and the analytical skills to understand and apply them effectively [Nummenmaa 2011].

2.3.4.3 Transformation Language

- **Model driven transformation languages:** These languages are intended specifically for model transformation which is central to model-driven development. They provide rule-based and pattern-based transformation languages for manipulating models. Many model driven transformation languages are proposed like the language that is standardized by Object Management Group (OMG): Query View Transformation (QVT). QVT is part of the Meta-Object Facility (MOF) and describes a model-to-model transformation. It is similar to a programming language, functions and variables can be defined to access models elements and their attributes. The metamodels of the source and target model are always necessary. Other transformation languages of this kind are ATLAS Transformation Language (ATL) developed by the INRIA [Jouault 2008] and Kafka was proposed in [Weis 2003].

- **Object-oriented languages:** Transformations are written using classical programming languages like Java programming language. It is possible to manipulate the models using these languages by means of extracting information from the source models and mapping them to the target model. Models, rules and patterns are imperatively described by source code.

2.3.4.4 Transformation Specificity

It indicates whether a transformation approach is specialized or designed for a specific application or domain. The transformation approach could be domain-specific application like the approach proposed by [Karim 2014], where the transformation is specific to Firefox extensions. In addition, it could be Domain-specific frameworks like the approach proposed by [Selim 2013], where the transformation is specific to Vehicle Control Software (VCS) development.

2.3.4.5 Type of Transformation

Based on the nature of languages and technologies used to express the source and target models of a transformation, two types of transformation can be distinguished; endogenous and exogenous [Mens 2006a].

- **Endogenous :** Endogenous transformations are transformations between models expressed in the same programming language and technology; the source and target models are written by the same language. refactoring, normalization and optimization are examples of endogenous transformation [Visser 2001].

- **Exogenous :** Exogenous transformations are transformations between models expressed using different programming languages and/or technologies; the source and target models are written by the different languages and/or designed for different technologies. software migration and code translation are examples of exogenous transformation [Visser 2001].

2.3.4.6 Relationship between Transformation's Source & Target

The source and the target artifacts of transformation could be model or text (source code, structured documents like XML, etc.) [Favre 2005]. Therefore, four possibilities of the source and target artifact could be faced by transformation; text-to-text, text-to-model, model-to-text and model-to-model.

- **Text-to-text (T-to-T):** This kind of transformation transforms from text to text, the text are usually source code. This transformation is also referred to as source-to-source or code-to-code.

- **Text-to-model (T-to-M):** This kind of transformation is primarily used in reverse engineering of software. Information that is inspected in the source code will be represented at model level. This transformation is also referred to as code-to-model.

- **Model-to-text (M-to-T):** This kind of transformation generates a model from a textual description, the generated code does not necessarily have to be executable. This transformation is also referred to as model-to-code or code generation.

- **Model-to-model (M-to-M):** This kind of transformation translates a model from one modeling language to another. It is considered a key aspect of model-driven development.

2.3.5 Direction of Transformation

2.3.5.1 Unidirectionality

Transformation languages or tools that have the property of unidirectionality usually require complex and imperative transformation rules, since each transformation can not be used in two different directions: the inverse transformation to transform the target model(s) into source model(s).

2.3.5.2 Bidirectionality

Transformation languages or tools that have the property of bidirectionality require fewer and declarative transformation rules, since each transformation can be used in two different directions: to transform the source model(s) into target model(s), and the inverse transformation to transform the target model(s) into source model(s).

Table 2.4: Transformation

Approach	Method				Technique		Transformation language		Type		Domain	Source-to-Target			
	Text	Code Structure	Wrapping	Model Driven	Rule	Pattern	OO	MDTL	Endogenous	Exogenous	System specific	T-to-T	T-to-M	M-to-T	M-to-M
[Akers 2004]		AST			√		√			√		√			
[Boshernitsan 2006]		AST				√	Java		√			√			
[Ding 2011]				√	√			ATL		√					√
[Wang 2006]		AST					√			√		√			
[Xue 2011]	√				√		√		√			√			
[Clavreul 2010]		AST		√	√			√	√				√		
[Nguyen 2014]		AST			√		√			√		√			
[Santos 2015]	√					√	√		√		√	√			
[Bruneliere 2010]		√		√	√	√	√	√	√			√	√	√	
[Fuhr 2011]		√		√	√		√	√		√		√			
[Ahmad 2014]		GML			√	√				√		√	√		
[Winter 2007]		Tree				√	TL		√		Java libraries	√			
[Matos 2011]	√	AST			√		Java (Eclipse plugin)			√			√		
[Selim 2013]				√		√		ATL	√		VCS				√
[Hunold 2009]		AST				√	language transformation processor		√			√			
[Poch 2009]		AST	√		√		EMF			√		√			
[Karim 2014]		AST			√		JavaScript			√	Firefox extensions	√			
[Hunold 2008]		AST					TXL-based		√			√			
[De Lucia 2008]			√				Java (Eclipse plugin)			√		√			
[Ying 2013]		AST			√		Extended Backus-Naur Form)		√		Web application	√			
[Binkley 2006]	√				√		Java		√			√			
[Seriai 2014]	√				√		√			√		√			
[Canfora 2008]			√		√		√			√				√	
[Tilevich 2009]	√				√		Java		√			√			
[Kegel 2008a]	√				√		Java		√			√			
[Allier 2011]			√		√		Java			√		√			
[Kapur 2010]	√				√		Eclipse API Tooling		√			√			
[Eysholdt 2010]				√		√	EMF			√				√	√
[Axelsen 2012]	√				√					√		√			
[Einarsson 2012]				√	√			QVTo		√					√
[Kjolstad 2011]	√				√		Eclipse refactoring engine		√			√			
[Gligoric 2014]	√						Metamorphosis Tool			√		√			

2.3.6 Output/Target of Migration

2.3.6.1 Target type

Target transformation artifacts can range from abstract analysis representations of the system to very concrete models of source code. Common tools are obviously needed such as code generators and parsers.

- **Code :** The output of the transformation is textual software artifacts that can be compiled and executed over target platforms (i.e., source code, byte-code, or machine code).

- **Model :** The output of the transformation is model software artifacts that can not be compiled or executed.

2.3.6.2 Target Context

- **Component-based :** The aim of the transformation approaches is for supporting software evolution. Most of these approaches migrate object-oriented applications into component-based ones. The level of transformation is varied, where some approaches merely aim to identify component implementation or recover its architecture, and others aim to produce an executable component but on a specific component model.

- **Service-based :** The aim of the transformation approaches is for supporting software evolution. Most of these approaches migrate object-oriented applications or component-based ones into services or service oriented architecture (standard service and web service).

- **Object-oriented :** The aim of these approaches is for supporting software maintenance and enhance its software quality for object-oriented applications by reengineering or transformation. These approaches aim to translate from a programing language to another one, refactoring source code, or to extract high level representation of the source code.

2.4 Migrating Object-oriented Programs into Component-based ones

2.4.1 Component-based Architectures

Software architecture is the representation of a software system at high level structures to link between business requirements and technical implementations [Ducasse 2009b]. It plays a key role as a bridge between software requirements and implementation [Maqbool 2007]. At least six aspects of software development

Table 2.5: Target of migration

Approuch	**Type**		Context		
	Code	Model	Component	Service	OO
[Akers 2004]	CCM		√		
[Boshernitsan 2006]	Java				√
[Ding 2011]		√		√	
[Wang 2006]	J2EE		√		
[Xue 2011]	x				√
[Clavreul 2010]	x				√
[Nguyen 2014]	C#				√
[Santos 2015]	OO				√
[Bruneliere 2010]	Java, J2EE	Graph			√
[Fuhr 2011]	Wep service (IBM's SOMA)			√	
[Ahmad 2014]	SOA development of cloud-enabled			√	
[Winter 2007]	Java				√
[Matos 2011]	Annotated OO			√	
[Selim 2013]		AUTOSAR	√		
[Hunold 2009]	OO				√
[Poch 2009]	SOFA		√		
[Karim 2014]	Jetpack framework.		√		
[Hunold 2008]	Java				√
[De Lucia 2008]	Web service			√	
[Ying 2013]	AJAX_JSON				
[Binkley 2006]	AspectJ				√
[Seriai 2014]	Spring Framework		√		
[Canfora 2008]	Wep Service			√	
[Tilevich 2009]	Java RMI				√
[Kegel 2008a]	Java				√
[Allier 2011]	OSGi		√		
[Kapur 2010]	Java				√
[Eysholdt 2010]	Java	Xtext/GMF			√
[Axelsen 2012]	Java				√
[Einarsson 2012]		UML Diagram			
[Kjolstad 2011]	Java Immutable class				√
[Gligoric 2014]	CloudMake				

that software architecture plays an important role in it [Garlan 2000] : understanding, reuse, construction, evolution, analysis, and management. Many definitions have been presented in the literature to define software architecture.

Perry and Wolf [Perry 1992] defined software architecture as a set of architectural elements that have a particular form. They distinguish three different elements: processing elements, data elements, and connecting elements. The processing elements are components that supply data elements that contain the information. The connecting elements are the glue that holds the processing elements together.

Bass et al [Bass 2012] proposed the following general definition: *"The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both."* The definition describes the system structure in terms of describing software elements (e.g. classes, components, subsystems) and their relationships (e.g. dependencies, connectors).

Moriconi [Moriconi 1994] proposed that a software architecture is represented using the six concepts: component, interface, connector, configuration, mapping, and architecture style. Component is an autonomous object like module, process, procedure, or variable. Interface is a logical point of interaction between components and their environment. Connector is defines the roles of component interactions and interface points. Configuration is a collection of constraints that wire component, connectors and interfaces into a specific architecture. Mapping is a correlation between the entities (vocabularies and formulas) of an abstract and a concrete architecture. Architecture style is a set of constraints that must be satisfied by an architecture that is written in the style , and a semantic interpretation of the connectors.

Consequently, based on the above definitions of software architecture, components, interfaces, and connectors are treated as first-class objects in software architecture.

Many definitions have been presented in the literature to define a software component. Each definition describes the software component from a different level of abstraction [Birkmeier 2009]. These definitions range from high level of abstraction to a technical one: domain-oriented component definitions like [Baster 2001], architecture-focused software component definitions like [Szyperski 2002] and Technical component definitions like [Lüer 2002].

Baster et al. [Baster 2001]: *"We define components as abstract, self-contained packages of functionality performing a specific business function within a technology framework. These business components are reusable with well-defined interfaces."*

Szyperski [Szyperski 2002]: *"A software component is 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."*

Luer et al. [Lüer 2002]: *"A component is a software element that (a) encapsulates a reusable implementation of functionality, (b) can be composed without*

modification, and (c) adheres to a component model."

We can realize that the domain-oriented component definitions focus on business components and domain specific functionality. while the architecture-focused software component definitions focus on component compositions and interfaces. Finally, the Technical component definitions focus on deployment and implementation aspects of the component.

2.4.2 Reconstruction Component-based Architectures

Successful applications evolve over time, they have many changes and modifications during their life cycle to meet new requirements [Ducasse 2009a]. So their architecture inevitably drifts, in other words the code does not conform to its architecture. A different kind of scenario when these applications did not already have software architectures particularly in legacy applications. Whatever, software architecture is not explicitly represented in the code, then the software architecture should be reconstructed to be synchronized with its implementation (code).

Software architecture reconstruction is a reverse engineering approach that aims at reconstructing feasible architectural views of a software application [Ducasse 2009a]. Software architecture reconstruction has been used in the literature through several other terms like reverse architecting, architecture extraction, mining software architecture, architecture recovery, or architecture discovery. Architecture recovery and architecture discovery terms are more specific than the others [Medvidovic 2003]: recovery refers to a bottom-up process, while discovery refers to a top-down process.

The most obvious goals of Software architecture reconstruction is to identify architectural views or elements. Software architecture reconstruction has encountered many challenges: It should support multiple architectural viewpoints because stakeholders have various concerns such as reusability, reliability, portability, or performance. Complex software systems sometimes are difficult to analyze and reverse engineer. For example, language concepts such as polymorphism, late-binding, delegation, or inheritance make these software system harder to analyze the source code [Dunsmore 2000, Wilde 1991]. So architecture reconstruction approaches need to be interactive, iterative, and parameterizable [Grundy 2000]. Moreover, large and long-living software systems have methods, languages, and technologies that are often heterogeneous which should be handled. Thus The major challenges are abstracting, identifying, and presenting higher level views from lower level (e.g. source code) and heterogeneous information.

Software architecture play important roles in software development [Garlan 2000]. These roles define the motivations for architecture reconstruction which are [Ducasse 2009a]:

- **Redocumentation and Understanding:** by reestablishing software abstractions views and help reverse engineers understand them.
- **Reuse and Migration:** by identifying components from existing systems

that can be reused or transformed into other system or technologies.

- **Conformance:** to check conformance between the software architecture and implementation code. Therefore, bridging the gap between high-level architectural models and the system's source code [Medvidovic 2006, Yan 2004].
- **synchronization:** software architecture and implementation evolve at different speeds, they should be synchronized to avoid architectural drift.
- **Analysis:** by measuring architectural quality analyzes by providing required architectural views that can assist stakeholders in their decision-making processes.
- **Evolution and Maintenance:** software architecture restructuring considered as a first step toward software evolution and maintenance. It reduces the system scope which should evolve [Medvidovic 2006].

As a result, the Software architecture restructuring is a first step toward migrating object-oriented software to component-based and even service-based ones. However, it was largely studied in the literature, several approaches and techniques have been proposed to support it [Birkmeier 2009, Ducasse 2009a, O'Brien 2002, Mendonça 1996]. In these works a software component is recovered as a cluster (set) of classes that collaborate with each other to provide the component functionalities [Crnkovic 2011b]. Next (subsection 2.4.3), we present our previous works: ROMANTIC [Chardigny 2008a, Kebir 2012], we have proposed an approach which aims to recover component-based architectures form object-oriented source code.

2.4.3 ROMANTIC: an Approach for Recovering Component-based Architectures

In [Chardigny 2008c] and [Kebir 2012], the authors presented an approach called ROMANTIC (Re-engineering of Object-oriented systeMs by Architecture extractioN and migraTIon to Component based ones). ROMANTIC aims to automatically recover a component-based architecture from the source code of a single object-oriented software. It is mainly based on two models:

1. Object-to-component mapping model that allows to link object-oriented concepts, e.g. package and class, to component-based ones, e.g. component and interface.
2. Quality measurement model that is used to evaluate the quality of recovered architectures and their architectural-elements.

2.4.3.1 Object-to-Component Mapping Model

ROMANTIC defines a software component as a set of classes that may belong to different object-oriented packages. The component classes are organized based on

two parts: internal and external structures. The internal structure is implemented by a set of classes that have direct links only to classes that belong to the component itself. The external structure is implemented by a set of classes that have direct links to other component classes. Classes that form the external structure of a component define provided and required interfaces. Figure 2.4 shows the object-to-component mapping model.

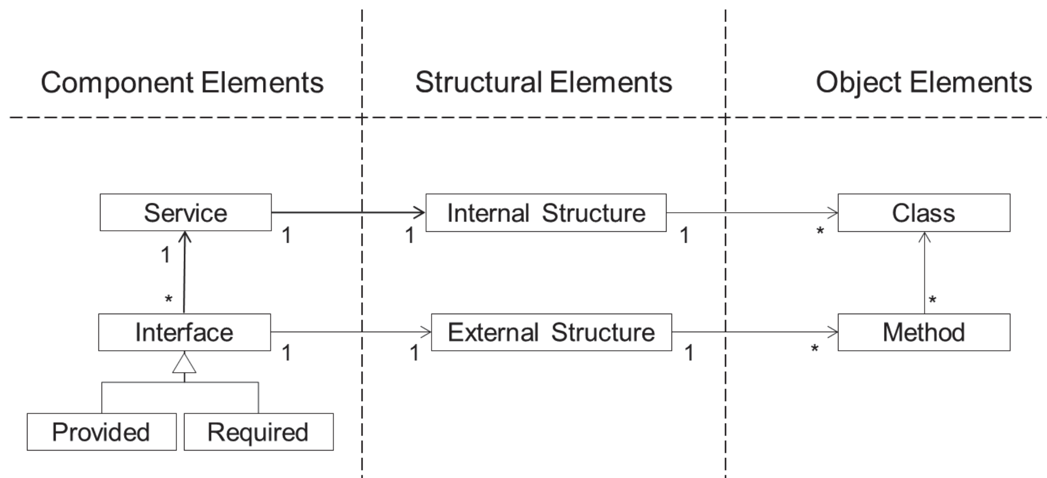


Figure 2.4: Object-to-component mapping model

2.4.3.2 Quality Measurement Model

According to [Szyperski 2002] [Lüer 2002] and [Heineman 2001b], a component is defined as "a software element that (a) can be composed without modification, (b) can be distributed in an autonomous way, (c) encapsulates the implementation of one or many functionalities, and (d) adheres to a component model "[Kebir 2012]. Based on this definition, ROMANTIC identifies three quality characteristics of a component: composability, autonomy and specificity [Chardigny 2008c]. Composability is the ability of a component to be composed without any modification. Autonomy means that it can be reused in an autonomous way. Specificity characteristic is related to the fact that a component must implement a limited number of closed functionalities.

Similar to the software quality model ISO 9126 [Iso 2001], ROMANTIC proposes to refine the characteristics of the component into sub-characteristics. Next, the sub-characteristics are refined into the properties of the component (e.g. number of 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). Lastly, these properties are refined into object-oriented metrics (e.g. coupling metric). Figure 2.5 shows how the component characteristics are refined following the proposed measurement model.

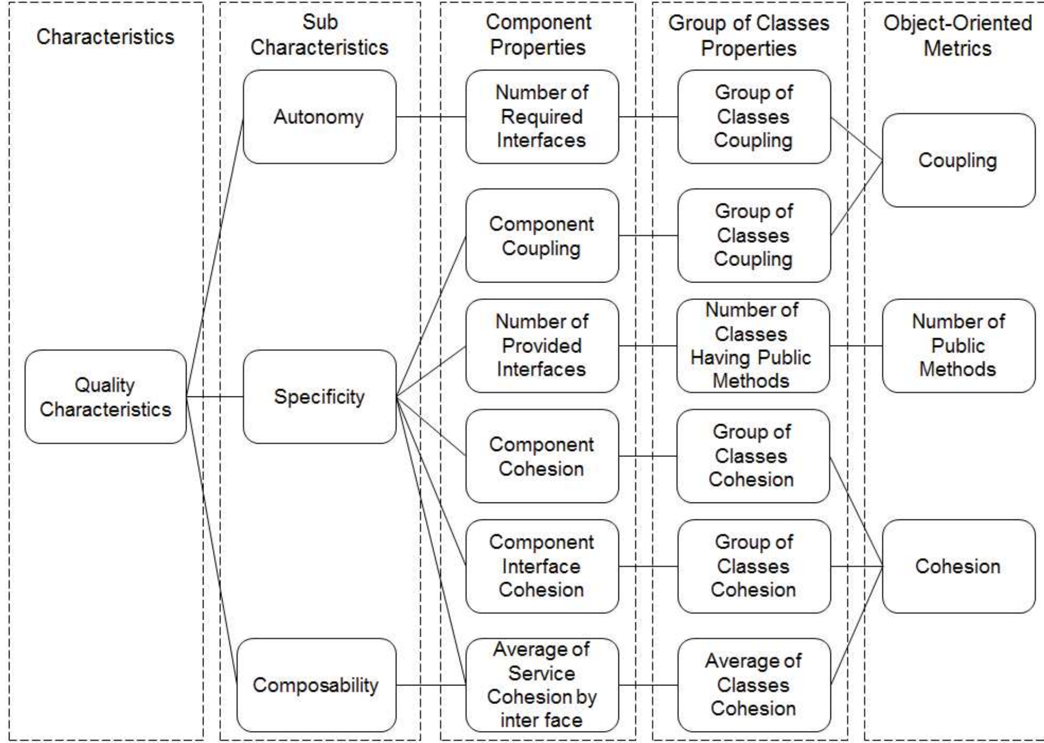


Figure 2.5: Component quality measurement model

Based on this measurement model, a quality function has been proposed to measure the quality of an object-oriented component based on its characteristics. This function is given bellow:

$$Q(E) = \frac{1}{\sum_i \lambda_i} \cdot (\lambda_1 \cdot S(E) + \lambda_2 \cdot A(E) + \lambda_3 \cdot C(E)) \quad (2.1)$$

Where:

- E is an object-oriented component composed of a group of classes.
- S(E), A(E) and C(E) refer to the specificity, autonomy, and composability of E respectively.
- λ_i are weight values, situated in [0-1]. These are used by the architect to weight each characteristic as needed.

ROMANTIC proposes a specific fitness function to measure each of these characteristics. For example, the specificity characteristic of a component is calculated as follows:

$$S(E) = \frac{1}{5} \cdot (\frac{1}{|I|} \cdot \sum_{i \in I} LCC(i) + LCC(I) + LCC(E) + Couple(E) + noPub(I)) \quad (2.2)$$

This means that the specificity of a component E depends on the following object-oriented metrics: the cohesion of classes composing the internal structure of E (*LCC(E)*), the cohesion of all classes composing the external structure of E (*LCC(I)*), the average cohesion of all classes composing the external structure of E ($\frac{1}{|I|} \cdot \sum_{i \in I} LCC(i)$), the coupling of internal classes of E (*Coupl(E)* which is measured based on the number of dependencies between the classes of E), and the number of public methods belonging to the external structure of E (*noPub(I)*). *LCC* (*Loose Class Cohesion*) is an object-oriented metric that measures the cohesion of a set of classes [Bieman 1995]. For more details about the quality measurement model please refer to [Chardigny 2008c] and [Kebir 2012].

This component quality function is applied in a hierarchical clustering algorithm [Kebir 2012, Chardigny 2008c] as well as in search-based algorithms [Chardigny 2008b] to partition the object-oriented classes into disjoint groups, where each group represents a component. In addition, it has been extended by [Adjoyan 2014] to be able to identify service-oriented architectures.

2.4.4 Running Example

To better illustrate the problem and solutions related to the object-oriented to component-based migration, we introduce an example of a simple Java application. This application simulates the behavior of an information screen (e.g. a software system which displays on a bus's screen information about stations, time, etc.).

In Figure 2.6, *ContentProvider* class implements methods which send text messages (instances of *Message*), and time information obtained trough *Clock* instances based on the data returned by *TimeZone* instances. The *DisplayManager* is responsible for viewing the provided information through a *Screen*.

Figure 2.7 shows the result of architecture recovery step applied on our example. The recovery step identifies four clusters (components), where each cluster may contain one or several classes. We consider a component-based architecture as a set of components connected via interfaces, where interfaces are identified from boundary classes. For example, the component *DisplayedInformation* connected to the *ContentProvider* component through two interfaces. The first interface declares *getCurrentTime* method which is placed in class *Clock* and *getContent* method from class *Clock*. The second one declares *getContent* method from class *Message*.

A cluster is composed of two types of classes: internal classes and boundary classes. Internal classes are classes that do not have dependencies (e.g. a method invocation or an inheritance relationship) with other classes placed into other clusters (e.g. *GpsLocation* and *Screen*). And the boundary classes are classes that have dependencies with classes placed into other clusters (e.g. *TimeZone* and *Clock*). We consider a component-based architecture as a set of components connected via interfaces, where interfaces are identified from boundary classes.

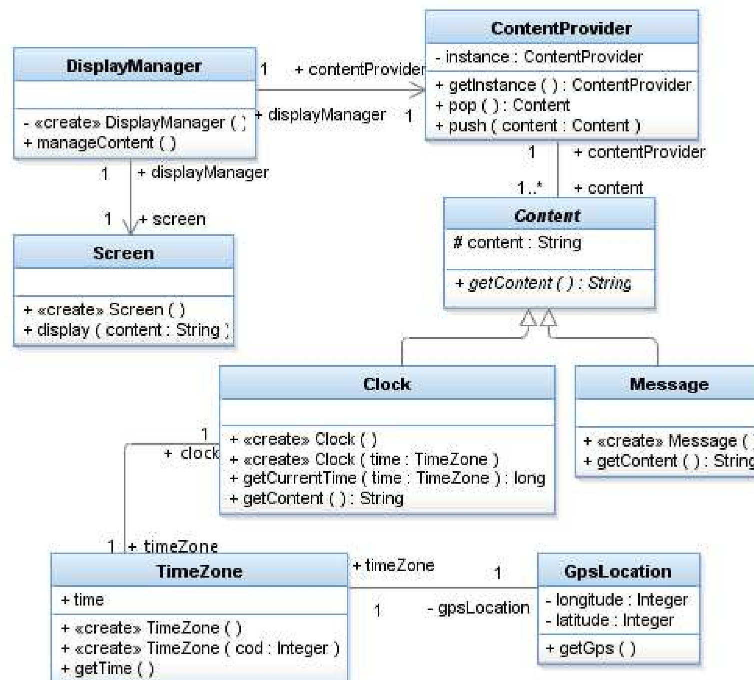


Figure 2.6: Information screen class diagram.

2.5 Discussion

In this section, we discuss findings that are obtained from our taxonomy. These findings are organized based on the taxonomy diminutions.

2.5.1 Input Source of Migration

Almost all classified approaches depends on the source code as the input artifact to their migration process. 31 approaches out of 32 rely on the source code, 5 approaches of them used other artifacts in addition to source code (3 approaches used documents and 2 approaches used models). Regarding the visibility of the input sources, 90% of the approaches depend on whitebox model where just 10% used blackbox. Graybox model does not have a chance on the classified approaches, to the best of our knowledge, there is no object-oriented migration approach using this model as an input.

This finding (regarding the input type and visibility) confirms the fact that the source code is the most available artifact in legacy software systems. Moreover, source code is the real implementation of legacy software systems, while their documentations and architectures are usually out of date (drifted). They are drifted because of modifications and maintenance during the systems life.

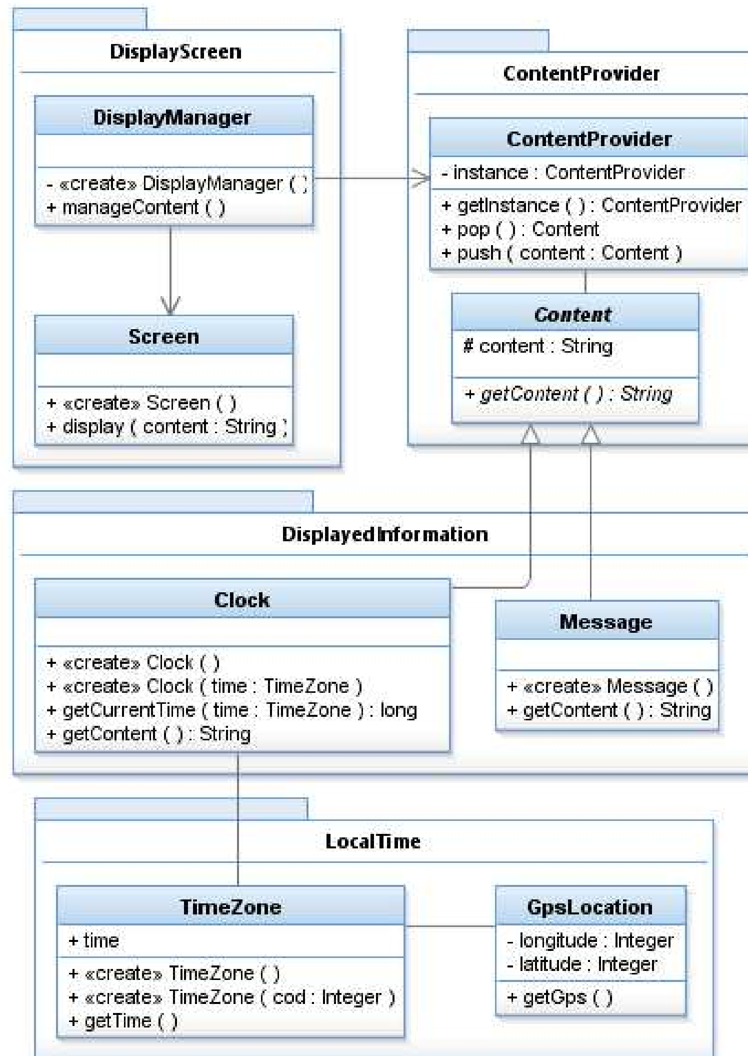


Figure 2.7: Architecture recovery for the information screen.

2.5.2 Reverse Engineering

Most classified approaches used the bottom-up method for investigating source code (81%), while 6.5% of the approaches used the top-down method for the investigation of software. The big difference between the previous two ratios relate to the availability and reliability of the source code rather than their abstract representations (e.g. software architecture). Combinations of the bottom-up and top-down methods are possible and recommended. 12.5% used the both methods, top-down for system understanding (i.e. a general overview of the architecture of the system is sufficient to recognize relationships) and bottom-up for detailed information.

In addition to saving time and efforts, The correct and stable automatic approach avoids further maintenance and development of the code and at the same time conserves the knowledge and quality contained within a well-tested proven code. 75% of the approaches are automatic, 22% are semi-automatic, and just 3% are manual.

Properties of subject software which are to be investigated may be static or dynamic in nature [Pressman 1986]. 84% of the approaches relied on the static analysis and 16% of them relied on the dynamic one. We have only one approach that relied on both analysis types. The superiority of static analysis over dynamic is in terms of availability, cost-efficiency, lower-risk, and it can by fully automation [Ernst 2003]. Hence the most static analysis tools depend only on the source code of subject software rather than dynamic ones, which need full knowledge of execution traces that are usually do not available in legacy software as well as the need for human executors or experts.

19% of the approaches recover high level abstraction in context of component-based like component-based architecture recovery or component identification approaches. In the context of service, 25% of the approaches targeted service-oriented architecture. The rest of the approaches (66%) present the analyzed software artifacts into other abstractions like call graphs or control flow graphs.

2.5.3 Transformation

One third (34%) of the approaches did not use any other presentation for source code. Most of these approaches used simple transformation (refactoring) that do not need other source code representations like find-and-replace transformation. While 47% of approaches used structural representation of the source code like abstract syntax tree. 12% of the approaches used wrapping methodology for transformation. in the context of model-driven software development, 22% of approaches used meta-models and their instances model to operate the transformation.

Most approaches (84%) used classical object-oriented languages to implement their transformation. Many tools and APIs are developed for code transformation (e.g. Spoon for Java, Eclipse JDT, ASF+SDF, CIL ,and Coccinelle) and generator (e.g. Acceleo, T4 by Microsoft, Telosys Tools). The rest of approaches used well-known transformation languages such as QVT or ATL. 3% of these percentages used

both languages.

The ratio between endogenous and exogenous transformation is very close. Hence, 47% of approaches are endogenous and 53% are exogenous. Endogenous transformation are developed usually for Optimization, Refactoring, Simplification and normalization [Mens 2006a]. In exogenous transformation, the goal is usually translating languages and/or migrating to new technology or platform [Mens 2006a].

Talking about the genericity of transformations, 16% of the approaches are domain specific. They can not be operated outside their purpose transformation. While 84% of the approaches are generic ones.

Based on the nature of the input and output transformation artifacts, 78% of the approaches transform source code to another one, and 13% transform it to model. 3% of them transform the source code into both forms, code and model. 22% of approaches take models as input, 9% transform the model to code and 13% to other models.

2.5.4 Output/Target of Migration

The output artifacts of the approaches are 91% concrete code and 19% models. 10% of them have mixed both code and models. 54% more than one half (54%) of the approach produced Java code. 22% produced code for component-based and the same for service-based (7% web service). The other outputs are 49% object-oriented code and 7% script code.

2.5.5 Goal of Migration

The most interesting goal of migration based on the classified approaches is evolution (41%). Followed by maintainability with 34%. Third place went to both reusability and efficiency with 28% for each. Finally, understandability with 22%.

2.6 Conclusion

In this chapter, we present the state-of-the-art related to migrate legacy software written in object oriented languages. We concentrate our effort to illustrate the recovery of software architecture and software component identification. This includes positioning our dissertation compared to the domain concepts and the related works. Related works are classified based on five dimensions. These are the inputs, reverse engineering approaches, transformation approaches, the output of migration and its goals. The chapter is concluded with the following remarks:

- First, a lot of of approaches have been proposed to recover component-based software architecture from object-oriented software systems.

- Second, few approaches have been proposed for transforming into component-based models.

- Third, these few approaches did not implement their transformation by using a well-known design like design patterns.
- Fourth, these few approaches did not treat all transformation from object-oriented to component-based like transforming object-oriented dependencies to interface-based ones.
- Fifth, none of these approaches have produced pure component (component that can be executable over an existing component model and confirm to standard component model).
- Sixth, none of them proposed a generic solution for migrating object-oriented software into many component models.

Chapter 3

Healing Component Encapsulation

Contents

3.1 Introduction . . . 43
3.2 Problem Statement . . . 44
3.2.1 Explicit component encapsulation violation . . . 44
3.2.2 Implicit component encapsulation violation . . . 44
3.3 Instance Handling Transformation . . . 46
3.3.1 Creating Object Interfaces: Uncoupling Boundary Classes . . . 46
3.3.2 Using Component Interfaces through the Factory Pattern . . . 47
3.4 Inheritance Transformation . . . 48
3.4.1 Replacing Inheritance by Delegation . . . 49
3.4.2 Handling Subtyping . . . 51
3.4.3 Dealing with Abstract Superclasses . . . 51
3.5 Exception handling transformation . . . 53
3.5.1 Transformation thrown exception . . . 53
3.5.2 Transforming exception handling . . . 54
3.6 Experimental Evaluation . . . 57
3.6.1 Experiment Design and Planning . . . 57
3.6.2 Results . . . 61
3.7 Conclusion . . . 69

3.1 Introduction

Most existing large legacy applications are object-oriented (OO) [Washizaki 2005]. These applications have complex and numerous dependencies. However, as we explain before (see Sec. 2.4.2), architecture recovery approaches identified components as clusters of classes. The dependencies between these clusters are still object-oriented one. Therefore we need to transform these dependencies to interface-based ones.

This chapter proposes a method that automatically transforms an OO application code to a CB one. We assume that an existing CB architecture recovery method provides us architecture descriptions as an input to our method. Based on the taxonomy of component models proposed in [Lau 2005], we chose, as the target

of our transformation, an object-based component model (i.e. components implemented based on OO source code). These component models are implemented as an extension of mainstream OO programming languages (e.g. OSGI is an extension of Java [Platform 2015, Lau 2005]). This choice allows us to reuse the OO source code to be migrated. In this work, we experimented the proposed solution on the transformation of Java applications into the OSGi framework.

The remainder of this chapter is organized as follows. Section 3.2 presents the migration process and its related issues. Section 3.3 explains the proposed solution to transform class instantiation dependencies. Section 3.4 describes our solution to transform OO inheritance relationships. Section 3.6 presents implementation and experimental results. We present the conclusion in Section 3.7.

3.2 Problem Statement

In this chapter we use clusters of classes obtained based on recovery approaches as an input of the source code transformation step. To transform clusters of classes to components we need to solve two main problems: Explicit component encapsulation violation and Implicit component encapsulation violation.

3.2.1 Explicit component encapsulation violation

The component must hide its internal structure and behavior [Szyperski 2002]. It should provide its services without exposing the classes that implement it. Two source code expressions fall under this category. First, "class instantiation", where a class (in one component) creates an instance of another class (residing in a different component). For example, *Clock* class creates an object of *TimeZone* class, while these classes belong to two different clusters. Second, "method invocation", where a method defined in a given class of a cluster invokes a method defined in a class placed in another cluster.

3.2.2 Implicit component encapsulation violation

It is related to implicit dependency between components caused by OO mechanisms. Two OO mechanisms that belong to this problem: inheritance mechanism and exception handling mechanism.

3.2.2.1 Inheritance

In the inheritance mechanism, a class and its subclasses cannot necessarily be placed in the same cluster. This is the case in Figure 2.7 for *Clock* and *Message* subclasses of *Content* Class. In this case, the inheritance relationship between these classes crosses component boundaries, facing an implicit dependency between the underlying components. Since component models do not all support inheritance (e.g. ComponentJ, COM, etc.) [Spacek 2012], source code related to inheritance needs to be transformed.

3.2.2.2 Exception handling

An exception corresponds to an abnormal state in the execution of a program. An exception is raised when such a state is detected. An exception handler is a lexical region of code that is executed in response to an exception occurrence. Different programming languages have different rules for matching an exception occurrence to a specific handler. An exception is handled when the execution of the handler is complete [Miller 1997]. The control flow of a program after a handler is executed is determined by an exception-handling model [Yemini 1985]. Three exception-handling models are commonly referred to in the literature [Buhr 2000, Miller 1997, Yemini 1985]. In the termination model, the lexical scope raising an exception is destroyed, and, if a handler is found and executed, control resumes at the first syntactic unit following this handler. In the resumption model, once an exception is handled, control continues where the exception was raised. Finally, in the retry model, when an exception is handled, the syntactic block raising the exception is terminated and then retried. There are a number of variants of exception-handling mechanisms: many variants can be distinguished by the exception model supported, and by the rules used to bind a handler to an exception occurrence. In this chapter, we focus on class-based [Abadi 1996, Abadi 2012] object-oriented languages that implement the termination model of exception handling, and in which handler selection is based on object types [Dony 1990]. Two common programming languages that fit this description are Java and C++ [Stroustrup 1991]. We show the transformation of this exception-handling model from object-oriented applications into component-based ones.

Figure 3.1 shows that an error occurs within a method m2, the method creates an object (exception) of type E1 and hands it off to the runtime system. After a method throws an exception, the runtime system attempts to find a handler to handle it. The set of possible handlers to handle the exception is the ordered list of methods that had been called. Therefore, the exception propagated to method m1. Method m1 has a handler of exception E1 so the handler is executed. Finally the system returns to the normal control flow. However, what if Client class and Server class are placed into different clusters? Two types of problems that must be transformed from OO concepts to component-based ones: (i) The thrown exception; the type of thrown exception maybe paced into another cluster. (ii) The propagation of the exception will be done implicitly by runtime system. Since components interact with each other's by their interfaces, source code related to exception needs to be transformed.

In this chapter we focus on solutions related to source-code transformations of explicit component encapsulation violation (instantiation and method invocation). In addition, we propose solutions for transforming OO inheritance and OO exception handling, which are cases of implicit component explicit violation, to other forms that conform to component principles.

flow.png flow.png flow.png

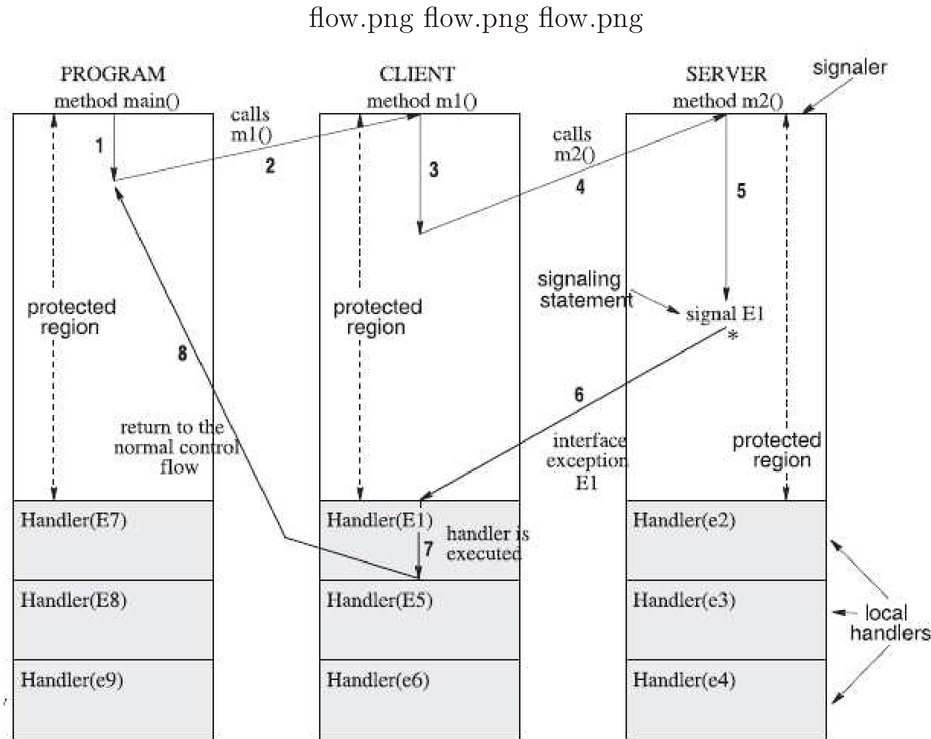


Figure 3.1: Flow of execution of exception-handling.

3.3 Instance Handling Transformation

Considering the result of the recovery step, a class belonging to a component (cluster) can be instantiated in a method of class belonging to another component by using directly these class constructors. This causes a violation of the principle of component encapsulation. Our approach proposes two steps to transform direct instantiation dependencies: (i) Uncoupling classes belonging to different components (clusters) by creating object interfaces. (ii) Defining specific component interfaces playing the role of object factories.

3.3.1 Creating Object Interfaces: Uncoupling Boundary Classes

We transform direct references (method calls) between classes of different components to interface-based calls. Thus when a class A uses class B where A and B are parts of two different components, we create a couple of the same provided and required interfaces (IB). The provided one will be defined in the component of the class B and the required one in the component of the class A. These interfaces define the same methods of all public methods of class B. In addition they define other methods to access public attributes of this class (i.e. setter and getter methods). Each direct use of class B in the class A will be refactored as a use of the required interface (IB) added to the component of A.

To illustrate this, consider our illustrative example, where *Clock* creates an instance of *TimeZone*. This is depicted in Listing 3.1. We create *ITimeZone* interface for class *TimeZone*. *ITimeZone* specifies the signatures of all public methods in *TimeZone*. Moreover, it declares setter and getter methods for its public attribute (*time*). Listing 3.2 shows the result of our transformation in both *Clock* and *TimeZone* classes.

Listing 3.1: Instantiation dependency in Java code.

```
public class Clock extends Content{
  public Clock() {
    TimeZone timeZone = new TimeZone();
    String time = timeZone.getTime();}
    ...
}

public class TimeZone {
  public String time;

  public TimeZone() {...}
  public String getTime(){...}
}
```

Listing 3.2: Creating object interfaces.

```
public class Clock extends Content{
  public Clock() {
    ITimeZone timeZone = new TimeZone();
    String time = timeZone.getTime();
    }
    ...
}

public class TimeZone implements ITimeZone{ ... }

public interface ITimeZone {
  public String setTime();
  public String getTime();
}
```

3.3.2 Using Component Interfaces through the Factory Pattern

The second step of this transformation is based on the Factory design pattern. Thus the expression in the source code related to the instantiation of a class B by a class A where these classes are parts of two different components is transformed to a use of a component interface playing the role of an object factory. This interface is defined as provided by the component of class B. It contains methods that return objects instantiated from classes of the component of class B. Each method of this interface corresponds to an existing class constructor. The methods of this interface are implemented in a factory class which is added to the classes of the component of the class B.

In Figure 3.2, we create a provided factory interface whose methods are implemented in the factory class. It has a method *createTimeZone()* that returns a new

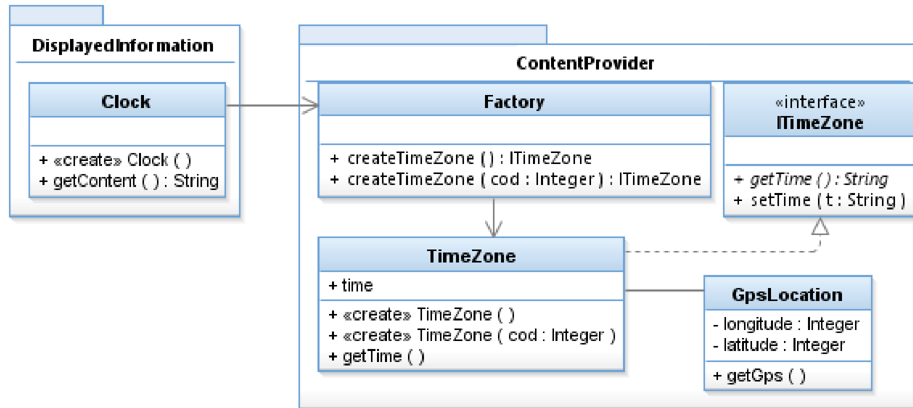


Figure 3.2: Transforming class instantiation based on the factory pattern.

ITimeZone() instance. The *Clock* class invokes this method instead of creating the instance. It does not expose the class that implements the interface, but exposes only the interface. Thereby, client code does not know the internal structure of *DisplayedInformation*. Listing 3.3 shows how *Clock* class gets a new instance of type *ITimeZone* using the *Factory* class.

Listing 3.3: Transforming class instantiation based on the factory pattern in OSGi code.

```
public class Clock extends Content{

  public Clock() {
    ITimeZone timeZone = Factory.createTimeZone();
    }
  ...
}

public class Factory{

  public static ITimeZone createTimeZone() {
    ITimeZone timeZone = new TimeZone();
    return timeZone;
    }

  public static ITimeZone createTimeZone(int cod) {...}
  ... // other provided factory methods
}
```

3.4 Inheritance Transformation

Inheritance links between classes belonging to different components need to be transformed. Our solution to transform these inheritance dependencies is based on the delegation pattern [Vlissides 1995]. In the case of object-oriented code, delegation

pattern related to two objects A and B corresponds to an explicit transfer (forward) for all method invocations received by A (called delegator) to B (called delegatee) through methods of B. All internal method invocations in methods of B related to this transfer must be transfered to delegator. This avoids the problem of the loss of the initial receiver [Kegel 2008b, Weck 1996].

3.4.1 Replacing Inheritance by Delegation

Our solution to transform the inheritance link consists in implementing the delegation pattern, but at component level (see Figure 3.3). Thus, inheritance link between two classes A and B (A subclass of B) which belong to two different components is transformed as follows. On the one hand, all methods invoked on the class A are transferred (delegated) to the component of B (considered as the delegatee) through a required interface. The required interface is implemented by the component of A (considered as the delegator) which is connected to a provided interface defined by the delegatee component. The provided interface defines all methods of the superclass B. On the other hand, all internal method invocations in the superclass B must be transferred to the delegator component through a required interface. This interface is implemented by the delegatee component and connected to a provided interface defined by the delegator component. This interface defines all methods of the subclass A.

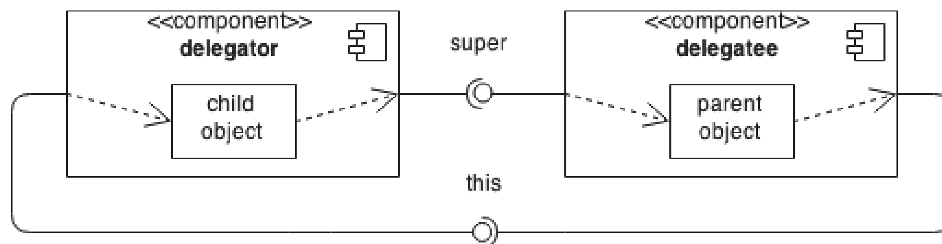


Figure 3.3: Implementation of the delegation pattern at level of component.

At class level, the transfer of method invocations between delegator and delegatee components and vice versa is realized by creating an instance of superclass B in the subclass A and by invoking the concerned method in this instance. This instance is created each time the class A is instantiated. Attributes of the instance of the class B are initialized using values given in the constructor of class A. The transfer of a method invocation from the delegatee to the delegator is realized in the same way by invoking this method on the instance of A. The reference of the instance of A is communicated to the instance of B as an additional parameter in each method invocation transferred from delegator to delegatee.

Figure 3.4 describes the transformation of inheritance between *Message* subclass and *Content* superclass. A new interface *IContent* is created for superclass *Content* into delegatee component (ContentProvider). Then, a variable is added to assign the initial receiver of type *IMessage* (*this*). Finally, the factory design pattern is

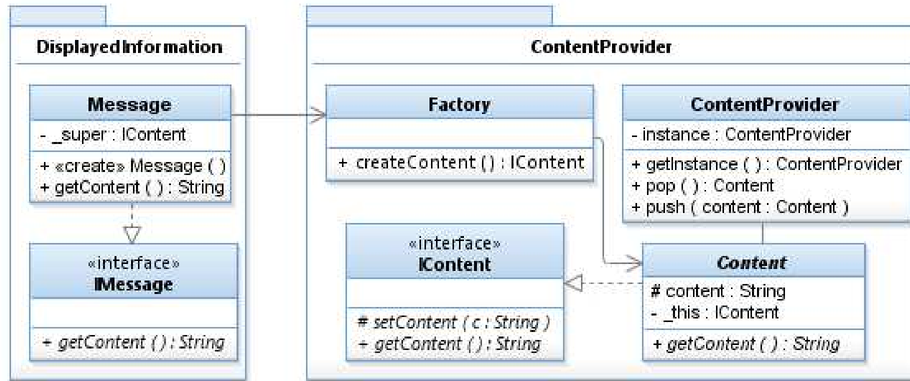


Figure 3.4: Replacing inheritance by delegation.

applied to provide *IContent* object interface. On the other side, we create a new interface *IMessage* for subclass *Message*. Then a new instance of the superclass object interface *IContent* is composed to delegate incoming method invocations.

Listing 3.4 describes the result of transforming inheritance to delegation. As we mentioned before, the transformation consists of three steps: (i) create new interfaces *IMessage* and *IContent*; (ii) *Message* is composed of an instance of type *IContent* as *super* interface; (iii) *IContent* is composed of an object interface of type *Message* as *this* interface.

Listing 3.4: Replacing inheritance by delegation.

```
public class Message implements IMessage{
  IContent _super = new Content(this);

  public void getContent(){
    _super.getContent();
  }
  ...
}

public class Content implements IContent{
  private IContent _this;

  public Content(IContent initReceiver) {
    _this = initReceiver;
  }
   public void getContent(){...}
  ...
}
```

Our solution transforms inheritance dependency but produces another dependency which is instantiation, where a subclass creates an instance of its superclass. So we apply here the solution proposed in the previous section (cf. Section 3.3).

3.4.2 Handling Subtyping

This section proposes a solution for the problem of breaking the supertype chain. In particular, a variable of superclass type can be assigned a reference to an instance of subclass type (polymorphic assignment), but the necessary assignment compatibility (subtyping) is removed by replacing inheritance with delegation. Another case occurs when a casting to superclass or a type test (*instanceof* in Java) exists in the program. For example, a variable (*content*) in class *ContentProvider* is typed with *Content*. It can be assigned an instance of *Message* or *Clock*. However, after transformation, this variable can not be assigned *Message* nor *Clock* instances.

Our solution suggests to use interface inheritance, which is the most common way to form subtypes between components [Szyperski 2002]. We introduce subtyping by adding inheritance between component interfaces providing methods of the subclass and its the component interface providing methods of the superclass. In the example of Figure 3.4, *IMessage* interface must inherit *IContent* interface. In the same way, *IClock* interface inherits from *IContent* interface. Therefore, a type of *IContent* can be assigned an instance of both types *IMessage* and *IClock*. Moreover, fields defined in *IContent* are now available in both *IMessage* and *IClock* by setter and getter methods (e.g. *setContent(c : String)* in class *Message*).

3.4.3 Dealing with Abstract Superclasses

As we explained before, a delegator is composed of an instance of a delegatee to delegate method invocations. However, what if the superclass is abstract? An abstract class cannot be instantiated, so no delegatee can be created.

Our solution is based on the proxy pattern [Vlissides 1995]. We use a third class as a proxy that breaks the inheritance between the subclass and its superclass when the latter is abstract. Thus the subclass inherits from this proxy, the proxy class inherits the abstract superclass. The proxy class defines the same methods with the same signatures of the abstract superclass. These methods are considered as proxy methods. Each of these methods delegates the received message to the abstract class when this class provides a concrete implementation of this method. In the case of an abstract method on the superclass, the corresponding method on proxy reforwards the message to subclass.

Actually, in our example (see Figure 2.6) *Content* is an abstract class and has an abstract method *getContent()*. *Factory* interface can not return an instance of this class. So we need to apply proxy pattern before applying delegation pattern. In Figure 3.5, we create a *Proxy* class that inherits from *Content* abstract class and implements *IContent* interface. Then *Factory* class provides an object interface of type *IContent* to *Message* which is placed in *DisplayedInformation*. This enabled us to decouple the inheritance dependency between the abstract superclass *Content* and its subclass *Message* that is placed in a different component.

Listing 3.5 shows the result of transforming inheritance that have abstract superclass to proxy pattern. *ProxyContent* class was created to break the inheritance

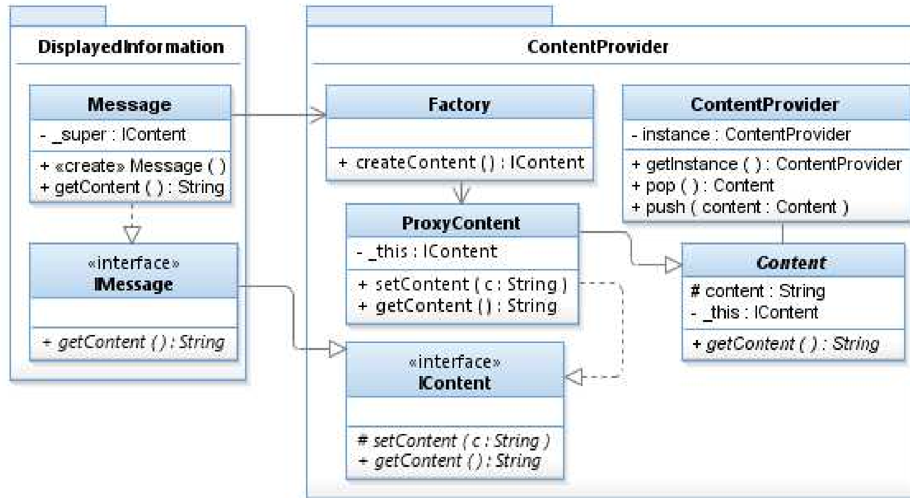


Figure 3.5: Handling abstract superclass based on proxy classes.

between *Message* and *Content*. The new class implemented all abstract methods inherited from it's superclass (*Content*). The calling of these methods are backward delegated to it's caller using our initial receiver variable *_this*. The non-abstract methods are called usually under the inheritance relationship between *ProxyContent* and *Content* and the composition between *Message* and *ProxyContent*. Consequently, the inheritance relationship that has un abstract superclass is transformed by using proxy pattern.

Listing 3.5: Handling abstract superclass based on proxy classes.

```
public class ProxyContent extends Content implements IContent{

  IContent _this = new Content(this);

  public ProxyContent(IContent initReceiver) {
    _this = initReceiver;
  }

  public void getContent(){
    _super.getContent();
  }

  ...
}

public class Message implements IMessage{
  private IContent _super;

  public Message() {...}
  public void getContent(){...}
}
```

3.5 Exception handling transformation

Considering the result of the recovery step, methods that are responsible to handle an exception raised are usually contained in different classes. These classes can be placed into different clusters. The propagation of the exception causes a violation of the principle of component encapsulation. Our approach proposes a solution to handle an exception using component interfaces by moving the responsibility of exception-handling from runtime system to components themselves through its interfaces. Our approach modifies the existent component interfaces to be able to provide normal and exceptional responds at the same time. When a client request a service from a server, the server respond either normal respond of that request, or the exception responds if an error raised during the execution of that request. The client should accept both response types (normal and exceptional) of its request.

As we described before, exception handling have two types of dependencies; throwing an exception and handling an exception. Our transformation goal is to decouple them. The following two subsections describe our solution for each type.

3.5.1 Transformation thrown exception

We need to replace direct instantiation of exception object to provide it through component interface. We create an adapter class which mediates between the two classes. The adapter is placed into the same component that is responsible to throw the exception. And it forwards the incoming method invocation to the original one (exception class) using object composition. Therefore, we replace the original dependency to a new one between adapter class and exception class. The new dependency can be transformed to be through the component interface. Indeed, adapter class have composed an object of exception class to handing a task over to him. So the solution of instantiation can be applied.

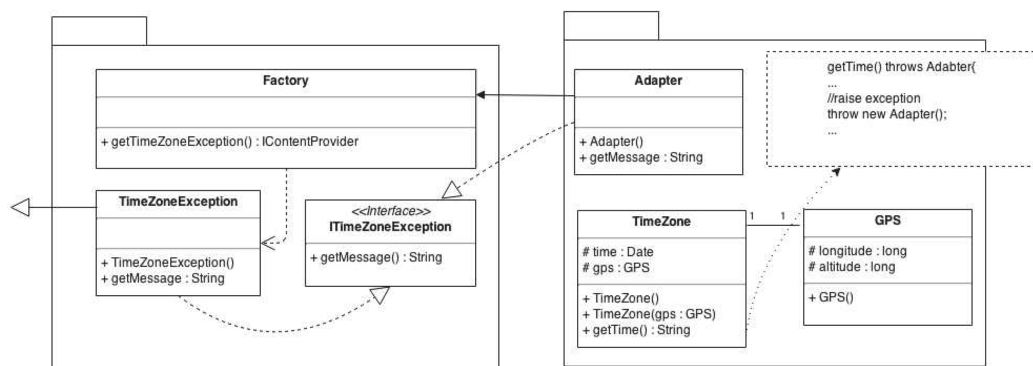


Figure 3.6: Transformation thrown exception by adapter.

Figure 3.5.1 shows transformation result of our example. On one hand, we applied Factory design pattern in *LocalTime* to provide an object interface of type *TimeZoneException*. On the other hand, we created *Adapter* in *LocalTime* that

implement the same interface (*ITimeZoneException*) of exception class (*TimeZone-Exception*). Then *TimeZone* throwing the exception object of type *Adapter* which is placed in the same component. Therefore decoupling the dependency between the two components.

3.5.2 Transforming exception handling

Right now, we decoupled the dependency between exception class and class who through exception was placed in other component. After that, an exception (exception object) is thrown to be handled by the runtime system. Indeed, runtime system passes the exception as a parameter to the special handler block. The handler block either catches the exception, or propagate the exception to calling methods iteratively until a method catch it. Therefore, the exception passed to classes implicitly under the auspices of the run time system.

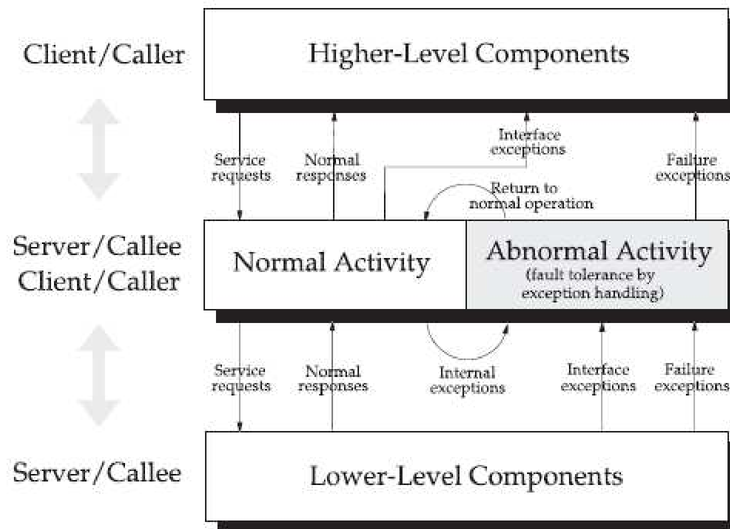


Figure 3.7: Ideal exception handling model for component [Bennett 1982].

Our solution is derived from idealized fault-tolerant component proposed by [Bennett 1982] (see Figure 3.5.2). We proposed that the response of a service which is provided by a component has three forms. the first form is normal response. Where the component provides his service normally without exception. The second form is exception response. Where an exception occurred in the provided service but the component can handle it. Therefore, the component does not depend on other components to handle his exception (internal exception [Bennett 1982]). The last form is propagate exception. Where the component can not handle the exception. So it propagates its exception to the caller of the service (see Figure 3.5.2).

Based on our exception model, we move the responsibility of exception handling to components. Where a component decides to reply its exception based on its

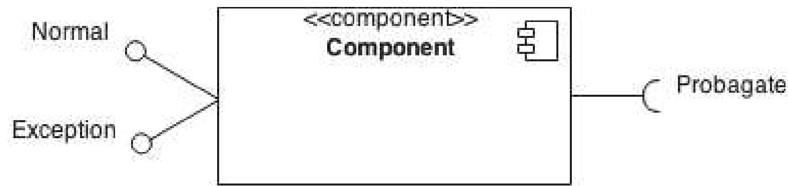


Figure 3.8: Our derived exception handling model for component.

Possibilities by dealing with the exception or propagating it. Therefore, we need to transform two code cases, handler block and propagate exception. Transforming involves separating the normal behavior of method from the exceptions. So each handler block is moved into a new method named the same as the original one plus the name of the exception. And, for a method that propagates exception, we create a new method as done previously but have an implementation propagates exception object to caller component. Where the caller is dynamic at runtime, so we used Reflection to identify it. To do so, our transformation approach consists of two steps: 1. Transform component interfaces to be able to handle normal and exceptional responses. 2. Transform exception-handling mechanism from runtime system into components.

3.5.2.1 Transform component interfaces

We transform each method that contributes to handle an exception to be able to return both normal value and exception value. The method put either a normal value (put exception value as NULL) or an exception value (put normal value as NULL). We wrap both the normal and exceptional return into a new wrapper. The wrapper is a class that has two parameters, the first one to store normal return values and the second one to store exceptional ones.

Listing3.6 describes the new wrapper class named Return. The wrapper accepts two types, the first one defined by the method to return its normal return value. The second one a subclass of Exception Class to return exceptional return value. Setter and getter method are implemented for both return types to store and retrieve it respectively.

Listing3.7 explains the transformation of method m2 in class Server. The transformation replaces the return type with a type of new wrapper (Return). When the exception E1 raised inside method m2, the new exception of type E1 is created and set as exceptional return values. In this case, the normal return value remains NULL. If there is no exception raised, the method set its normal return value while the exceptional one remains NULL.

Listing 3.6: Return wrapper class.

```
public class Return<V, E extends Exception> {

  E upNormal;
  V normal;
```

```
  public void setNormal(V normal) {
    this.normal = normal;
  }

  public void setUpNormal(E upNormal) {
    this.upNormal = upNormal;
  }

  public V getNormal() {
    return normal;
  }

  public E getUpNormal() {
    return upNormal;
  }

}
```

Listing 3.7: The transformation of method m2 in class Server.

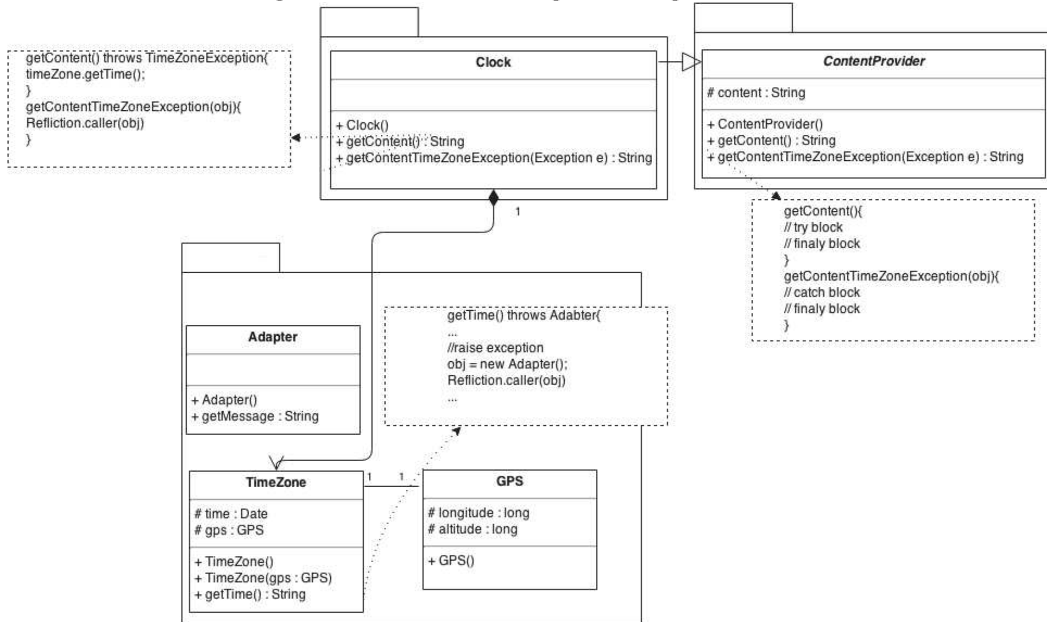
```
public class Server {

  public Return<Type, E1> m2(){
    Return<Type, CustomExceptionDelegator> ret = new Return<>();
    //...
    ret.setUpNormal(new E1());
    //...
    return ret;
  }
}
```

3.5.2.2 Transform exception-handling mechanism

In this step we handle exception explicitly by components. Where each component checks the response of its request if it has normal response value or exceptional one. If the response has the normal value, then the execution remains in normal control flow. However, if the response has exception value, then the handler block must be executed. Listing 3 shows how client check the respond value from its server. After method m1 call method m2, m1 check the return value of m2 if it has exceptional value or not using getUpNormal() method. If it does not have exceptional value then the execution continues as normal. Otherwise, if it has exceptional value, then method m1 call handler block that implemented in anonymous method in interface catch.

Figure 3.9 describes our transforming of the exception handling example illustrated in Figure 3.5.1. Where after the exception raised in *DisplayedInformation* through method *getTime*, an exception object is created and passed to caller (*Clock* in *DisplayedInformation*) rather than throwing it. The method *getContentTimeZoneException* in *ContentProvider* receives exception object as a parameter. Where *clock* can not handle the exception, then it propagates the exception object to caller (*ContentProvider* in *ContentProvider*). Also it receives exception object as a parameter on its method *getContentTimeZoneException*. This method have handled

Figure 3.9: Transforming handling exception.



the exception and return results. Finally, the execution returns to normal mode where the exception was raised.

3.6 Experimental Evaluation

This section reports on some experiments we conducted to evaluate our approach.

3.6.1 Experiment Design and Planning

Research Questions

[RQ1:] *Does the transformed code is semantically equivalent to the original one?*

Our approach transforms OO dependencies to interface-based once using well known OO design patterns. These design patterns are already theoretically proven for reserving the semantics of code. for example, the semantically equivalent for replacing inheritance with delegation has been proved [Kegel 2008a, T Genssler 1999, Kegel 2008b]. The aim of this research question is to proof that our transformation approach reserved the semantic of transformed code practically.*Does the transformation result avoid component encapsulation violation?*

Our approach transforms the OO code to avoid component encapsulation violation by making the dependencies between components explicit. The transformation aims at creating and using component interfaces to achieve component encapsulation. Thus, the aim of this research question is to measure the con-

tribution of our approach to transform OO dependencies to CB ones. *To which extent does the automatic transformation reduce the developer's effort?*
The aim of this research question is to measure the saved efforts of developers when using our automatic transformation approach instead of using manual one.

Evaluation Methods To answer **RQ1**, we have executed the predefined test cases on the source code before and after applied our transformation approach. After that we have compared the results (state testing and behavior testing) between original code and transformed one.

To answer **RQ2**, we need to evaluate how much the OO dependencies are transformed to interface-based ones. This can be measured by the ratio of the number of interface-based dependencies to the total number of dependencies between components after transformation. The *Abstractness* metric proposed by Martin [Martin 2011] for evaluating OO software fulfils this goal. This metric represents the ratio of abstract types (interfaces and abstract classes) in a package to the total number of types in that package. The range for this metric is 0.0 indicating a completely concrete package to 1.0 indicating a completely abstract package. In the context of CB software, this metric has been adapted by [Hamza 2013] to measure the quality of a component's interfaces, where the classes that represent the component's provided interfaces are grouped in a package to compute *Abstractness*. Therefore, we used this metric in the same way as [Hamza 2013] to answer **RQ2**. Based on this metric, a well designed component is supposed to provide only interfaces. Therefore, a component with high *Abstractness* means a high component encapsulation (i.e., it avoids the component encapsulation violation).

To answer **RQ3**, we compared the estimated efforts expressed by time spent by developers through manual transformation to the time made by our automatic transformation. We compute the time for each type of transformation, instance handling and both inheritance with and without an abstract superclass transformation.

Data Collection We have conducted our transformation approach on 9 Java projects in order to validate our approach. The projects have been selected from *Qualitas* Corpus [Tempero 2010]. In order to guide project selection in such a way that the coverage of a sample is maximized, we have followed the following selection criteria:

RQ3 i **Project size**: We have selected projects with different sizes.

ii **Domain**: We have selected projects from different domains to avoid the influence on experimental results of characteristics associated to a specific domain.

iii **Development team**: We have selected projects that have been developed by different teams to avoid the characteristics related to programming team habits to influence experimental results.

Table 3.1 provides some descriptive measures about these projects. It provides each project name and its version. We can observe the differences of these projects through its sizes and domains. We can infer the differences of development teams by the differences of owned company, where each project was developed by different companies.

Table 3.1: Data collection.

Application	Version	Domain	# of classes	Code size (KLOC)
Tomcat	7.0.71	middleware	1359	196
Ant	1.9.4	parsers/generators/make	1233	135
Checkstyle	6.5.0	IDE	897	63
Freecol	0.11.3	games	669	113
JFreeChart	1.0.19	tool	629	98
HyperSQL	2.3.2	database	539	168
Colt	1.2.0	SDK	288	35
Log4j	1.2.17	testing	220	21
Galleon	0.0.0-b7	3D/graphics/media	137	26

Protocol For architecture recovery, we used our method called *ROMANTIC* [Chardigny 2008a, Kebir 2012] which allows us to identify a component-based architecture from an existing Java application[1]. We applied *ROMANTIC* on our selected Java projects (Table 3.1). *ROMANTIC* clusters each project as a set of disjoint clusters (components).

For code transformation, we developed a tool (an Eclipse plugin) that automatically transforms the result of architecture recovery (clusters of java code) to OSGi components (bundles). Our tool parses the source code using the Abstract Syntactic Tree (AST) generated by Eclipse's JDT. After that, it makes transformations on this AST for the instantiation, method invocation and inheritance dependencies between components.

We conducted three experiments to answer our three research questions respectively. In the first experiment we compared the results obtained from executing test suites between the transformed source code and the original one. The transformed code are tested before packaging it into a specific component models. So, we execute test suite on OO source code that have been transformed. Therefore, we did not need to integrate the predefined test cases to a new platforms. We used JUnit testing frameworks that are most popular and available for Java.

In the second experiment we compare the *Abstractness* between the recovered components before transformation (i.e., OSGi components with direct OO dependencies) and the same components after transformation (i.e., OSGi components

[1] It is worth recalling that the experiment can be conducted using another recovery method. Only the output (in which we "trust"), which takes the form of a set of class clusters, is important for the remaining steps.

with dependencies though provided and required interfaces)[2]. We voluntarily limited the number of types to those which are provided and required by components that depend on each other. In the third experiment, we compare developers' efforts (time) between manual transformation and automatic transformation. It is obvious that the automatic transformation provides better results (small values for the transformation time), but what we would like to show here is the estimated average time to perform transformations manually on a whole Java project. The time to do it automatically is measured to estimate the multiplying factor between the two transformation processes.

In this evaluation we firstly compare the state and behavior between transformed code and the original one. We executed all test suite that are predefined for each applications.

In the second experiment we computed *Abstractness* for components (clusters) that resulted from architecture recovery step. To compute *Abstractness* for a component ***C***, we start by searching for classes of ***C*** that are used by classes of other components (provided types). Then we compute the ratio of the number of interfaces and abstract classes that belong to provided types to the total number of provided types (see Equation 3.1). After that, we used our transformation tool to transform Java clusters into OSGi components (bundles) with provided and required interfaces. Then we recompute *Abstractness* as described in equation 3.1 for OSGi components. Finally, we compare the *Abstractness* values to answer our research question **RQ2**.

$$Abstractness(C) = Na/Np \tag{3.1}$$

where Na is the number of interfaces and abstract classes that belong to provided types of component C,
and Np is the number of provided types of component C.

In the third experiment, we performed the transformation manually. To this end, we selected from our data collection three projects that have different sizes. We chose *Log4j* as a small project, *JFreeChart* as a medium project and *Tomcat* as a large project. We selected just three representative projects from the nine composing our data collection (cf. Table 3.2). We do this selection to adapt the manual experimentation to the available resources (people and time). We invited 15 developers to transform Java source code. To make sure that we obtain a relatively fair valuation, we split this group of people into three groups, five people for each. Table 3.2 provides descriptive information about these people. Before starting the experimentation, we checked that each person has understood the steps presented in our approach to applied for transforming OO to CB code. In each

[2] OSGi model allows creating components with either direct OO dependencies between classes composing these components or through interface-based connections [Platform 2015]

group, we provided each person with the source code. In addition, we gave them the information about three components with different sizes (small, medium and large) in each project (9 components in total). Then we asked them to randomly select three classes from each project and from different components. The selected classes must be transformed by satisfying our conditions: selected instantiation, inheritance and/or exception dependencies to be transformed must be related to classes belonging to other component(s). We measured the time for three types of transformation: instantiation, inheritance and inheritance with an abstract superclass, and exception handling.

Table 3.2: Information about people involved in the experiment.

Persons	# persons	Group	Experience in Java
Ph.D Students	5	1	3-6 years
Developers	5	2	4-6 years
M.S. Students	5	3	2-4 years

3.6.2 Results

3.6.2.1 Architecture Recovery Results

Table 3.3 provides some descriptive statistics about architecture recovery results (on the whole data collection, and not on the three projects selected for manual transformation). It displays the number of components recovered from each project (16-129). Moreover, it shows the nature of the components; average number of classes per component (8.5-20.7) and how strongly components are related te each other using *Afferent Couplings* (10.33-24.85). *Afferent Couplings* (also known as Outgoing Dependencies) is a metric that measures the number of types outside a component that depend on types inside the component. According to the obtained results, we observed that the number of components is almost directly proportional to the project size except in case of *Tomcat* and *Freecol* projects.

Table 3.3: Architecture recovery results.

Application	# components	Avg. number of classes per component	AVG. Efferent coupling per component
Tomcat	125	10.8	13.87
Ant	129	9.5	10.33
Checkstyle	63	14.5	13.98
Freecol	36	18.5	22.47
JFreeChart	40	15.7	15.63
HyperSQL	26	20.7	24.85
Colt	23	12.5	10.70
Log4j	23	9.5	10.74
Galleon	16	8.5	15.56

3.6.2.2 Code Transformation Results

Table 3.4 provides descriptive statistics about transformation types for our data collection. It describes the number of transformations that must be performed according to our approach for each project. The results show that instantiation is the most common transformation type with an average of 61.1% from all transformation types in all projects. Then transforming exception handling with an average of 15.9%. Then transforming inheritance that have abstract superclass with an average of 12.8% (except for HyperSQL, Clot and Galleon, where transforming inheritance is slightly bigger than transforming abstract superclass). Finally, transforming inheritance with an average of 10.2%.

Table 3.4: Statistics of transformation types.

Application	# instantiation transformations	# inheritance transformations	# abstract superclass transformations	# exception transformations
Tomcat	350	49	79	74
Ant	364	50	54	62
Checkstyle	249	37	41	49
Freecol	164	28	34	41
JFreeChart	116	22	38	40
HyperSQL	99	20	19	40
Colt	70	17	13	35
Log4j	62	16	28	32
Galleon	56	18	14	26

According to the obtained results in Table 3.4, we observed that the number of transformations is (in most cases) directly proportional to the number of components. As can be seen in Figure 3.10, the relationships between the transformation types and the number of components for our data collection. However, a small exception of that relationship occurred in the case of *Tomcat* and *Freecol* projects.

Semantics Results We selected three projects (*Tomcat*, *JFreeChart* and *Log4j*) to validate that our transformation approach produces source code semantically equivalent to original one. All test cases are executed on each selected project before and after transformation. The results of these executions were the same for each transformed code and the original one that are belonging into the same project. These results prove that the state and behavior of each selected projects are the same before and after transformation. Consequently, our approach preserved the semantic of the source code (answer to *RQ1*)..

Abstractness Results Table 3.5 shows the difference of *Abstractness* values between the components before and after transformation. Moreover, it gives the multiplying factor between the two Abstractness measures. The improvement factor ranges from 3.57 for *Tomcat*, which basically has a good design in terms of abstractness, to 8.33 for *HyperSQL*. The improvement of the level of abstractness depends thus on the analysed software system. On average in the considered data collection, our approach improved *Abstractness* by 5.57 times (answer to *RQ2*).

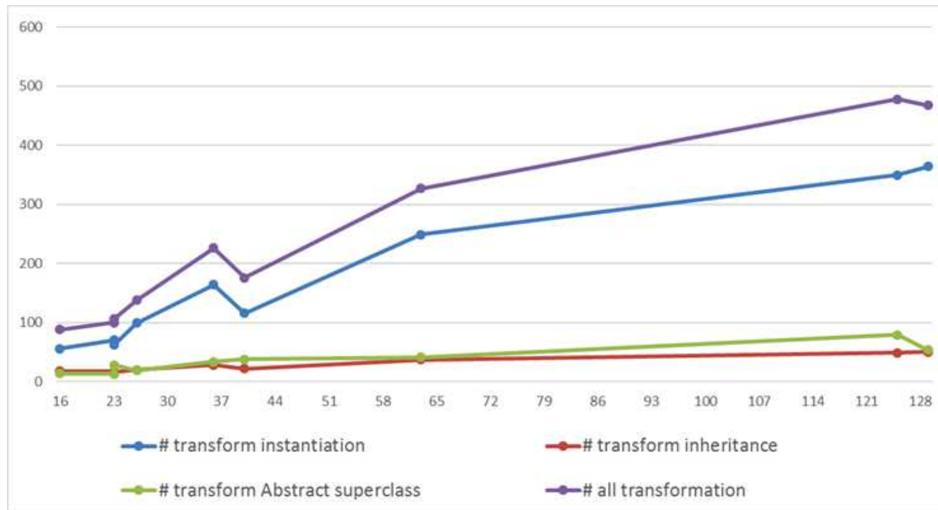


Figure 3.10: Relation between number of transformations with number of components.

Table 3.5: Improvement of abstractness after transformation.

Application	Abstractness before transformation	Abstractness after transforming instantiation	Abstractness after transforming inheritance	Abstractness aftert ransforming abstract supercalss	Abstractness after transforming exception	Improvement factor
Tomcat	0.28	0.46	0.06	0.10	0.10	3.57
Ant	0.18	0.56	0.08	0.08	0.10	5.56
Checkstyle	0.12	0.58	0.09	0.10	0.11	8.3
Freecol	0.17	0.51	0.09	0.11	0.13	5.89
JFreeChart	0.22	0.42	0.08	0.14	0.14	4.54
HyperSQL	0.12	0.49	0.10	0.09	0.20	8.33
Colt	0.26	0.38	0.09	0.07	0.19	3.84
Log4j	0.19	0.36	0.09	0.16	0.19	5.26
Galleon	0.21	0.39	0.12	0.10	0.18	4.76
AVG	**0.19**	**0.46**	**0.09**	**0.11**	**0.15**	**5.57**

We can observe that the *Abstractness* for all applications is significantly improved. This improvement lies at the core of our transformation approach, where we transform OO direct dependencies into component interface dependencies. Another observation depicted in Figure 3.11 is that the values of *Abstractness* reaches the optimal value (1.0).

Manual vs. Automatic Transformation Results The results of the second experiment are presented in Table 3.6. It shows the results of manual transformation for the three selected projects. The first two columns present the selected projects (*Tomcat*, *JFreeChart* and *Log4j*) and the transformation types. The number of needed transformations that must be achieved according to our approach are presented in the third column. Then the fourth and the fifth columns show the number of the transformations that were performed manually. The values of the fourth column shows all these transformation while the values of the fifth column present

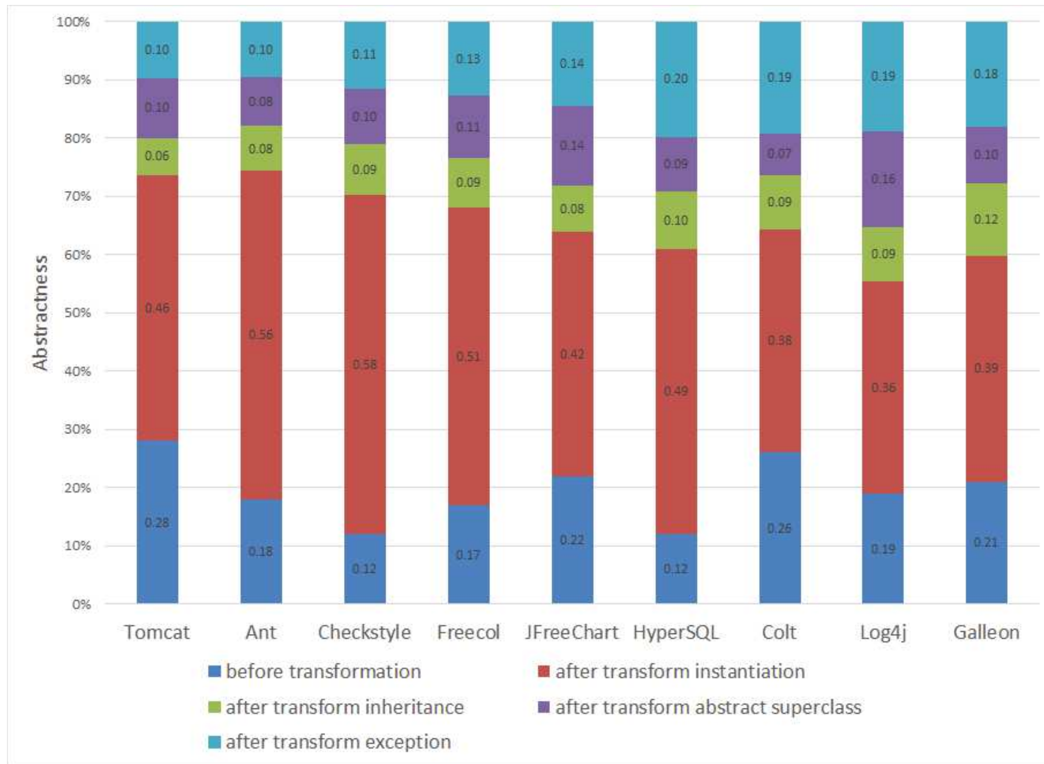


Figure 3.11: The percentage of abstractness for each transformation type.

only the number of unique transformations (i.e. transformations on dependencies done only by one developer). For instance, the number of manually transformed instantiation in *Tomcat* is 35. But the number of different manual transformation is 20. That mean 15 classes out of 35 were repeatedly transformed by different people. For example, class *WebappLoader* that belong to *Tomcat* was transformed three times. We note that the ratio of the number of realized manual different transformations to the number of all transformations automatically achieved is about 22%. This ratio constitutes a good base to compare results of manual and automatic transformations.

The sixth column shows the number of wrong manual transformations. A wrong transformation corresponds to a case where this transformation is not properly done. For example, it is the case when a person transformed OO inheritance between two classes which belong to the same component. As we have noted before, the people manually transforming source code understood very well our approach before we did the experiments. Therefore, the wrong transformations are not the result of misunderstanding our approach. The ratio of the number of wrong transformations to the total number of manual transformations is about 18%. This means that approximately one fifth of manual transformations were wrong.

Table 3.6: Estimated time for manual transformation.

Application	Transformation type	# of needed trans.	# of manual trans.	# of different manual trans.	# of wrong trans.	AVG. time (s)	Min/Max time (s)	STD time (s)	AVG. estimated time (h)
Tomcat	Instansiation	350	35	20	2	367	230/1008	126	35.68
	Inheritance	49	3	3	6	1106	928/1380	241	15.05
	Abstract superclass	79	16	9	2	1310	1019/1803	195	28.75
	Exception	74	13	8	2	1255	989/1747	173	25.80
JFreeChart	Instansiation	116	37	16	0	395	192/901	169	12.73
	Inheritance	22	16	15	2	1053	862/1301	148	6.44
	Abstract superclass	38	11	3	2	1198	1012/1405	135	12.65
	Exception	40	7	5	3	1077	1002/1359	104	12.00
Log4j	Instansiation	62	34	17	0	377	248/869	158	6.49
	Inheritance	16	9	6	11	1054	892/1401	188	4.68
	Abstract superclass	28	6	6	5	989	982/1106	64	7.69
	Exception	32	7	4	1	1033	932/1203	120	9.18

The transformation time is presented in the rest of the table (last four columns). The first three ones represent statistics about the manual transformation time in seconds for each selected project presented following the type of transformation. We can observe that the mean time for each type of transformation realized in different selected projects (*AVG. time column*) is approximatively the same. For example, the mean time taken to transform inheritance is ranged from 1053 to 1106, where the difference is just 53 seconds which is a small value compared with the transformation time (i.e. 1053 or 1106). The *Min/Max time* shows the minimum and the maximum time for the corresponding types of manual transformations which indicates to the variation of the transformations time. Moreover, a standard deviation is provided in the column *STD time* to better illustrate the amount of variation or dispersion of the manual transformation time. The little standard deviation values compared to the mean reflect a small amount of variation of the transformations time values.

The mean of the estimated time in hours to manually realize a type of transformation for each selected project is presented in the last column (*AVG. estimated time*). We compute these values by multiplying the number of the needed transformations for each project by the mean values of manual transformation time. The conclusion related to this column is that manual transformation is not an easy task. For example, to completely transform *Tomcat*, *JFreeChart* and *Log4j* manually, we need 92.21, 43.82 and 28.22 hours respectively. For example, in the case of *Tomcat*, this corresponds to more than three weeks of work according to French (employment) laws.. In addition, we did not calculate the cost caused by incorrect transformations that have an error percentage about 18%.

On the contrary, our tool transforms *Tomcat* for example in a few minutes (about 6 minutes) without any incorrect transformation. The ratio between the manual and automatic transformation times for *Tomcat* is 795. Thus, we can answer **RQ3** that our automatic approach effectively reduces the developer's efforts especially on large projects.

3.6.2.3 Threat to Validity

Internal threats: one internal threat needs to be considered when interpreting our experimentation results. This is related to the used architecture recovery approach in our experiment. For example, we observed that the number of the needed transformation depends on the number of the recovered components. The number of components depends on the used architecture recovery approach. Consequently, the improvement ratio of *Abstractness* and the saved transformation efforts obtained by our approach can be affected depending on the architecture recovery approach that are used (*ROMANTIC* approach [Kebir 2012]). For example, *Tomcat* have 350 instantiation dependencies that must be transformed (see Table 3.6). As the architecture recovery approach is responsible for identifying components (i.e., find clusters of classes), the number of dependencies between these components differs depending on the used recovery approach. Thus, the 350 instantiation depen-

dencies that must be transformed in *Tomcat* may be less or more depending on the architecture recovery approach used.

External threats: External validity refers to generalizability of the results. In this study, we have two threats to external validity to generalize our results. The first one is related to our data collection and the second one is related to the types of people who applied our approach manually.

Data Collection We performed our experiments on nine different-sizes, several-domains, well-known and open-source Java projects. Moreover, the projects are selected from different development teams. Because of the variety of our data collection, we can say that our results can be generalized to involve most Java projects.

Types of Persons We experimented our approach manually on three groups, five people in each group. The groups are PhD students, Java developers and master students (MS). By reference to Table 3.2, all these groups have an experience in Java development ranging from 2 to 6 years. Additionally, the experiments were applied on the three groups under the same conditions. In this threat we need to check that the time consumed by manual transformation was not affected by the types of people and their familiarity with Java development. In addition, we need to check that the errors caused by manual transformation was not affected by the types of people also. Figure 3.12 shows the time consumed by each group for each transformation type. We can see that the transformation times are close to each other for the three groups. For transforming instantiation, the manual transformation time ranged between 315 to 397 seconds. For transforming inheritance, the manual transformation time ranged between 1073 to 1106. For transforming inheritance that has abstract superclass, the manual transformation time ranged between 1231 to 1258. Finally, For transforming exception, the manual transformation time ranged between 1245 to 1266 seconds. Consequently, the differences of time consumed by the three groups is negligible. Thus we can emphasize that the time consumed for manual transformation is independent of person type.

Figure 3.13 shows the error percentage caused by manual transformation for each group. We can observe that the results are almost the same. The percentages of error transformation were 15%, 16% and 17% for PhD students, developers and MS respectively. Thus, we can emphasize that the error caused by manual transformation is independent of person type.

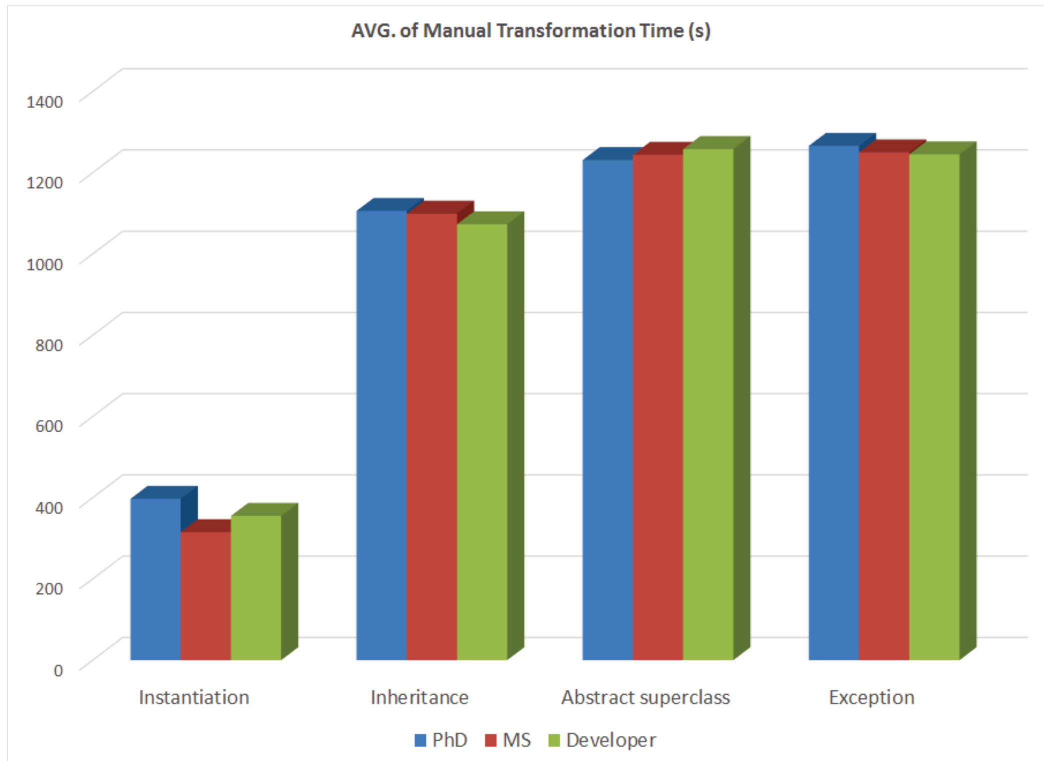


Figure 3.12: The mean of manual transformation time for each group.

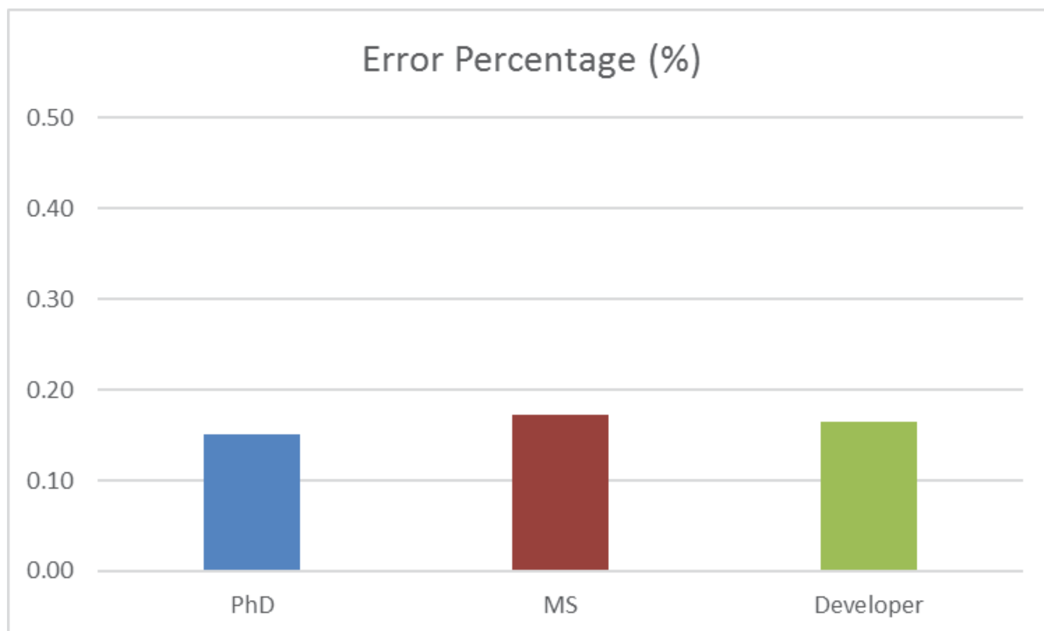


Figure 3.13: The error percentage of manual transformation for each group.

3.7 Conclusion

In this chapter, we proposed an approach to automatically transform object-oriented applications to component-based ones. We targeted the transformation of applications which are built using object-oriented languages into applications built with an object-based component model. We focus on the transformation of source code in order to produce decoupled components that are compliant with the architecture recovered in a previous step. We proposed a solution for dealing with instantiation, method invocation, inheritance dependencies and exception handling. The transformation solution based on well known design patterns like Factory, delegation, Proxy and adapter design patterns. Therefore, the transformed code is easy to understand for future maintenance. The experimentation results shows that our approach preserved the semantic of the source code. Moreover, it improves *Abstractness* and as a consequence reduce the violation of component encapsulation. Finally, it effectively reduces the developer's transformation efforts especially on large projects. The transformation was demonstrated by transforming Java applications into OSGi ones.

Chapter 4

Reveal Component Instance

Contents

4.1 Introduction 71

4.2 Problem Statement 72

4.3 Transforming OO Code to CB One 73

4.3.1 Generating Component Descriptor and Reference of its Implementation 73

4.3.2 Component Instantiation 74

4.3.3 Reveal Component-based Architecture 77

4.4 Mapping the Proposed Solution onto Component Models . 79

4.4.1 Mapping from Java to OSGi 79

4.4.2 Mapping from Java to SOFA 2.0 81

4.5 Discussion 82

4.6 Conclusion 83

4.1 Introduction

In the previous chapter, we proposed an approach to solve the problem of component encapsulation by transforming object-oriented dependencies to interface-based ones. In this chapter, we propose an approach for the identification and creation of component instances based on OO source code. This requires moving from the concept of object to a component instance. The identification and creation of component instantiation aims to solve the gap between software architecture and its running application.

The proposed approach aims at transforming OO code to CB code guided by the recovered architecture of the corresponding OO software. This approach makes it possible to reveal component descriptors, component instances and component-based architecture to materialize the recovered architecture. For that, the recovered clusters should not be considered as simple packaging and deployment units. They should be treated as real components: true structural and behavior units that are instantiable from component descriptors and connected together to materialize the architecture of the software. To validate this approach, we applied it to transform Java code to two well known component-based languages; OSGi [Platform 2015] and SOFA [Bures 2006].

The remainder of this chapter is organized as follows. Section 4.2 discusses the problem statement. Section 4.3 presents the transformation of OO code to CB one. Section 4.4 presents how the proposed solution is mapped onto OSGi and SOFA. Section 4.5 presents the discussion about our solution. Finally, Section 4.6 contains some concluding remarks and gives directions to future work.

4.2 Problem Statement

Clusters of classes identified from architecture recovery represent the primary implementation code of components. This code should be transformed to match targeted CB languages. These languages can be classified into two main categories. The first category distinguish the language used for describing components and architectures (architecture description language) from the language used to implement components (programing language) like SOFA [Bures 2006]. The second category use the same language for describing architecture descriptions and component implementations like COMPO [Spacek 2014]. In our work we focus on transforming OO code to one written using CB language of the first category. This transformation makes it possible to reuse classes of recovered clusters as the implementation of the target components. Table 4.1 summarizes the main structural elements of languages of this category. These consist of:

1. Structural elements that define component descriptions:

   (a) Component interfaces: the component descriptor needs to define provided and required interfaces. All interactions between components must be done through these interfaces.

   (b) Implementation reference: the component descriptor needs to define references of its component implementation source code.

   (c) Component instantiation: the component descriptor needs to define how its component can be instantiated.

2. Architecture description: it describes the structure of component-based systems in terms of component instances and component assembly. It ignores components implementation details and interactions.

Our approach aims at generating structural elements composing component descriptors and architecture description starting from source code of recovered clusters. In our previous work [Alshara 2015], we have proposed an approach that transforms dependencies between clusters to be interface-based ones. This approach presented component interfaces structural elements. In this chapter we complete the transformation by addressing the remaining structural elements of component descriptors; implementation references and component instantiation. This leads to revival of the CB architecture.

Table 4.1: Object-based Component Model Specifications [Crnkovic 2011a]

Component Models	Language of implementation	Interfacs type	Component Descriptor	Component instance
EJB [E.E. Group 2006]	Java	Operation-based	Yes	Single Object
Fractal citefractal	Java, C#, .Net	Operation-based	Yes	Single Object
JavaBeans [Microsystems 1997]	Java	Operation-based	Yes	Single Object
COM [Box 1997]	OO languages	Operation-based	Yes	Single Object
OpenCOM [Clarke 2001]	OO languages	Operation-based	Yes	Single Object
OSGi [Platform 2015]	Java	Operation-based	No	Many Objects
SOFA 2.0 [Bures 2006]	Java	Operation-based	Yes	Single Object
CCM [OMG 2011]	Language independent with OO implementation	Operation-based & Port-based	Yes	Single Object
COMPO [Spacek 2014]	COMPO	Operation-based & Port-based	Yes	Single Object
Palladio [Becker 2007]	Java	Operation-based	Yes	Single Object
PECOS [Winter 2002]	OO languages	Port-based	Yes	Single Object

4.3 Transforming OO Code to CB One

4.3.1 Generating Component Descriptor and Reference of its Implementation

Our approach uses the concept of class used in OO to express component descriptors. Hence, a class will represent the component descriptor. For example, the descriptor of *DisplayedInformation* component translated by creating a new class *DisplayedInformation*. Where the component descriptor describes their interfaces, the same concept of interface in OO languages is used to describe component interfaces. Then each provided interface has an OO interface that explicit its services (method signatures). The component descriptor must have the reference of its implementation of all provided interface services. For example, Listing 4.1 shows how the provided interfaces for component *DisplayedInformation* are created. But, what if two interfaces have the same method signature? The descriptor can not implement two services in the same descriptor (this is the case in Java, but in C++ and C# we can implement the same services that have the same signature by referencing the interface name before the implemented methods). For example, component *DisplayedInformation* provides two interfaces and the two interfaces have a method with the same signature(*getContent()*). Consequently, we should provide each interface with a component port.

Listing 4.1: Provided interfaces for *DisplayedInformation* component

```
public interface ITime {
  public String getContent();
  public long getCurrentTime(ITimeZone timeZone);
}
public interface IMessage {
  public String getContent();
}
```

The explicit services provided by a component interface are associated with a port. In our approach, we use the inner-class concept used in OO to represent component ports. Thus, each port is described by an inner-class associated with its interface. For example, in Listing 4.2, the *PortTime* inner-class is created to implement *ITime* interface provided by component *DisplayedInformation*, as same as *PortMessage* inner-class. Moreover, the references of each inner-class (port) are

provided by its component (e.g. *portTime* and *portMessage* class-variables) for binding components.

Listing 4.2: Descriptor and ports for *DisplayedInformation* component

```
public class DisplayedInformation{

  public static ITime portTime;
  public static IMessage portMessage;

  private class PortTime implements ITime{
      @Override
      public String getContent() {
      //TODO: add behaviore implementation
      }

        @Override
      public long getCurrentTime(ITimeZone timeZone) {
      //TODO: add behaviore implementation
      }
  }

  private class PortMessage implements IMessage{
      @Override
      public String getContent() {
      //TODO: add behaviore implementation
      }
  }

}
```

4.3.2 Component Instantiation

4.3.2.1 Mapping object instances to component instances

In OO, an instance consists of state and behavior, the state is stored in variables and exposes its behavior through methods. Object hides its internal state where methods operate in an object internal state to provide services through object-to-object communication (encapsulation). However, the recovered component is viewed as a set of one or more cooperating classes. Thus, we infer component instances from a set of class instances belonging to the same component, where the component state is the aggregated state of these instances, and the component behavior is published through the component interfaces. For example, in Fig. 4.1, we have three object call graphs for a component consisting of five classes (A, B, C, D, E). We can observe that:
(1) The component instance has three different releases (Fig. 4.1 (a), (b) and (c)).
(2) The component instance could have many class instances of the same type. For example, Fig. 4.1-(c) have two class instances from type E (e1 and e2).
(3) The client needs to have references to the class instances that provide services/methods for them. For example, the classes that implement the provided component services are A and B. Then, the client needs to reference instances of type A and B to get their required services. After that, instances of type A and B

are responsible for communicating with other instances to complete their services. And therefore, the classes that have the component provided services are considered as the only entrance to component instance.

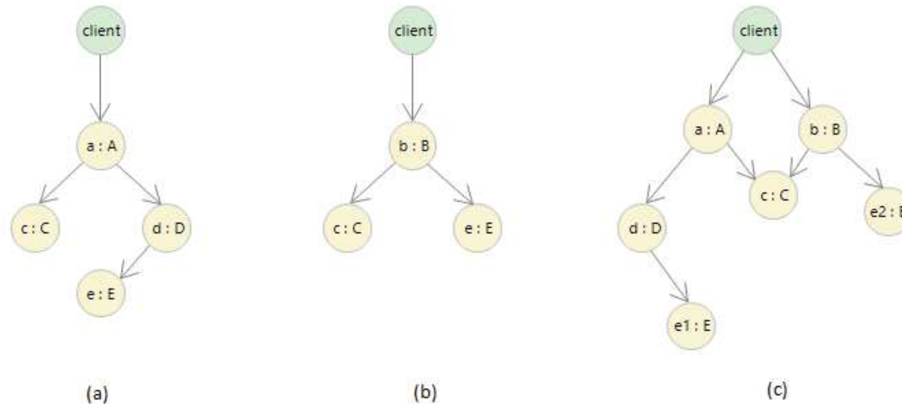


Figure 4.1: Different release of the same component instance

Based on our interpretation of the component instance, the set of class instances that constitute a component instance should behave as a single unit. Then, we need to update component descriptor to manage the links between class instances that form a component instance. We propose to delegate provided interface methods in the component descriptors to real ones. For example, Listing4.3 describes the update of the descriptor of *DisplayedInformation* component. The descriptor has references of the class types that are responsible for providing component services *Clock* and *Message*. After that, the delegations of provided services is done through component ports by using the real class instances that have these services. It is worth noting that we used the lazy instantiation of these class instances (delaying the instantiation of class instance until the first time it is needed) for performance reasons.

Listing 4.3: Component descriptor with its behaviors

```
public class DisplayedInformation{

  protected static ITime portTime;
  protected static IMessage portMessage;

  //Boundary Classes
  Clock clock = null;
  Message message = null;

  public DisplayedInformation() {
    //initializing component ports
    portTime = new PortTime();
    portMessage = new PortMessage();
  }
```

```
  private class PortTime implements ITime{

    @Override
    public String getContent() {
      if(clock == null){ //lazy instantiation
        clock = new Clock();}
      return clock.getContent();
    }

    @Override
    public long getCurrentTime(ITimeZone timeZone) {
      if(clock == null){ //lazy instantiation
        clock = new Clock();}
      return clock.getCurrentTime(timeZone);
    }
  }


  private class PortMessage implements IMessage{

    @Override
    public String getContent() { //lazy instantiation
      if(message == null){
        message = new Message();}
      return message.getContent();
    }
  }

}
```

4.3.2.2 Creating Component Instances

The services of a component can not be used directly, the component descriptor must first be instantiated. Like in OO programs, we need a constructor to create a component instance and initialize its state. The constructor of the component should be placed into the component descriptor. In addition, the descriptor implements the component services through component interfaces using associated ports. Thus, we create a default constructor (constructor without parameters) that initializes component ports. Listing 4.4 describes the default constructor of component *DisplayedInformation* and how it creates its ports (*PortTime* and *PortMessage*).

Initializing component state depends on the constructors placed into classes that have provided methods to other components (e.g. *Clock* and *Message* into *DisplayedInformation* component). For example, class *Clock* has two constructors, the first one without parameters (default constructor) and the second one with a single parameter of type *ITimeZone*. So there are two possible ways to create an instance of type *Clock*. Therefore, the component descriptor should provide all possible ways to initialize its instances. Consequently, *initialize* methods are created with different parameters to apply component configurations. For example, *DisplayedInformation* component has two classes that can be accessed from outside components (*Clock* and *Message*), and each of them has default constructor while *Clock* class has one more with *ITimeZone* parameter. Therefore, *initialize* methods are created and has *ITimeZone* parameter (see Listing 4.4).

Listing 4.4: Component constructors and initializers

```
public class DisplayedInformation{
...
  public DisplayedInformation() {
    //initializing component ports
    portTime = new PortTime();
    portMessage = new PortMessage();
  }

  public initialize(ITimeZone timeZone) {
    clock.setTimeZone(timeZone);
  }
}
```

Now, we can simply create an instance of the component using its constructor using OO instantiation and then initialize the instance using appropriate initializer. For example, an instance of *DisplayedInformation* component is created by its constructor using *new* keyword. Listing 4.5 differentiates the refactoring resulted from our approach (*ComponentClient*) and the original source code (*ClassClient*).

Listing 4.5: Component instantiation

```
public class ClassClient{

  Clock clock = new Clock(timeZone);
  clock.getCurrentTime();
}

public class ComponentClient{
  DisplayedInformation info = new DisplayedInformation();
  info.initialize(timeZone);
  info.portTime.getCurrentTime();
}
```

4.3.3 Reveal Component-based Architecture

An architecture description describes the structure of component-based systems in terms of component instances and binding. Therefore, to reveal a CB architecture, we need to identify its component instances and the binding between these instances. We can identify the component instances by analyzing the instantiation statements of its implementation. We can identify the binding between these instances based on the invocation of its services.

4.3.3.1 Identify component instances

We first statically analyze the source code to check whether to create a new component instance or to use an existing one. The analysis is based on statement scope (i.e. in the same code block) and obliterates state (i.e. the second instantiation statement obliterates the state of the instance resulted from the first one). The previous component instance can be replaced by a set of its class instances if these

set at the same scope and no one obliterates the state of another one. For example, in Listing 4.6, the IF-BLOCK into class *ClassClient* instantiates an object of type *Clock* and another of type *Message*. However, the proposed approach replaces the two instances with a component instance of type *DisplayedInformation* (*info1*) because they are in the same scope and each one does not obliterate the state of another. An example of the scope condition is obviously shown by defining *info1* and *info2*, where each of them belongs to different scopes. Defining *info2* and *info3* provides an example of obliteration state condition, where *message2* will obliterate the state of *message1* if it translated to one component instance. Listing 4.7 shows the component instances that have been identified from Listing 4.6.

Listing 4.6: Refactoring instantiation from OO code into CB one

```
public class ClassClient{

  if(condition)
  {
    Clock clock = new Clock(timeZone);
    Message message = new Message();

  }else{
    Message message1 = new Message();
    ...
    Message message2 = new Message();
  }
}


public class ComponentClient{

if(condition)
  {
    DisplayedInformation info1 = new DisplayedInformation();

  }else{
    DisplayedInformation info2 = new DisplayedInformation();
    ...
    DisplayedInformation info3 = new DisplayedInformation();
  }

}
```

Listing 4.7: Identified CB instances for architecture discriptor

```
//Darwin ADL
inst
info1 : new DisplayedInformation();
info2 : new DisplayedInformation();
info3 : new DisplayedInformation();
```

4.3.3.2 Identify component binding

Binding between component instances is used to establish interactions between these instances. An instance of component binds to another one to provided or required services through their interfaces. Therefore, we can identify the bindings between components based on service invocations between them where components must firstly bind to provided or required services. For example, in Listing 4.8, *ContentProvider* invokes a service *getCurrentTime* from *DisplayedInformation*, so the binding between these two component must be established before. Therefore, we can statically analyze these invocations between components to identify bindings (see Listing 4.9). Fig 4.2 shows the architecture recovery (c.f. Sec. 2.4.4) and a snapshot of architecture description for our running example. The architecture description describes component instances and its binding between *DisplayedInformation* and *ContentProvider* component instances.

Listing 4.8: Refactoring instantiation from OO code into CB one

```
public class ContentProvider{

  public void push(DisplayedInformation info1){
    String time = info1.portTime.getCurrentTime();
    }
}
```

Listing 4.9: Refactoring instantiation from OO code into CB one

```
//Darwin ADL
inst
content : new ContentProvider();
information : new DisplayedInformation();

bind
content.I1 -- information.ITime
}
```

4.4 Mapping the Proposed Solution onto Component Models

In this section we describe how our proposed solution is easily mapped onto existing component models. We have chosen two well known component models, OSGi and SOFA, to explain the ease of mapping.

4.4.1 Mapping from Java to OSGi

OSGi is a set of specifications that define a component model for a set of Java classes [Platform 2015]. It enables component encapsulation by hiding their implementations from other components by using services. The services are defined by

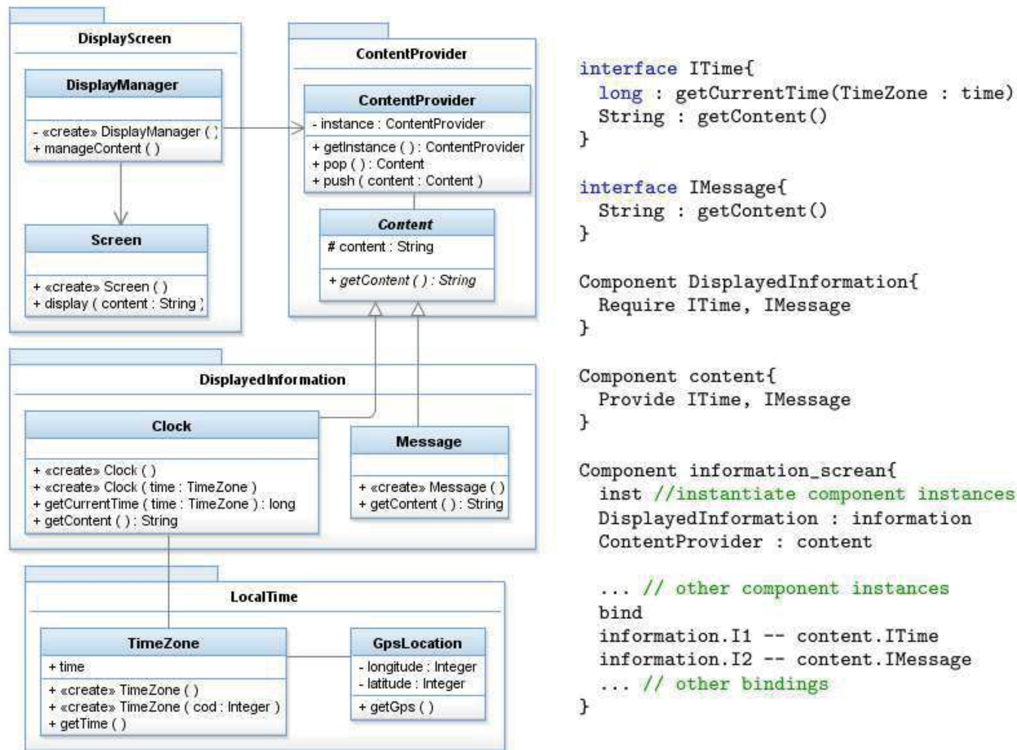


Figure 4.2: *Information-screen* architecture recovery and Darwin ADL for *DisplayedInformation* and *ContentProvider*

standard Java classes and interfaces that are registered into a *service registry*. A component (bundle) can register and use services through the *service registry*.

Listing 4.10: *DisplayedInformation* component describtor and its interface

```
public class DisplayedInformation implements IDisplayedInformation{
  /* Contents... */
}

public interface IDisplayedInformation {
  public InterTime portTime = DisplayedInformation.portTime;
  public IMessage portMessage = DisplayedInformation.portMessage;
}
```

To map our transformed code onto the OSGi framework, we firstly create an interface (Java interface) to represent the contract of provided component instance. For example, Listing 4.10 shows how we created an interface for *DisplayedInformation* component. Hence we suggest that a component binds through its port associated with a provided interface, then both ports *InterTime* and *IMessage* must be accessed by other components. After that, a metadata for both provided component *DisplayedInformation* and required component *ContentProvider* must be specified. The metadata specified through XML files uses the declarative services model. For

example, Listing 4.11 describes how *DisplayedInformation* component provides its instances as object interfaces with type *IDisplayedInformation*. And Listing 4.12 describes how *ContentProvider* component uses the provided instances. When both components are activated at runtime, the binding is established between them. Listing 4.13 describes how *ContentProvider* gets an instance of *DisplayedInformation* and calls its method *getContent()* through port *portMessage*.

Listing 4.11: DisplayedInformation.xml file to provide the instances of *DisplayedInformation*

```
<?xml version="1.0" encoding="UTF-8"?>

<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="DisplayedInformation">
   <implementation class="DisplayedInformation"/>
   <service>
      <provide interface="IDisplayedInformation"/>
   </service>
</scr:component>
```

Listing 4.12: ContentProvider.xml to bind the instances of *DisplayedInformation*

```
<?xml version="1.0" encoding="UTF-8"?>

<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="ContentProvider">
   <implementation class="ContentProvider"/>
   <reference bind="setDisplayedInformation" cardinality="1..n"
        interface="IDisplayedInformation" name="DisplayedInformation" policy="static"
        unbind="setDisplayedInformation"/>
</scr:component>
```

Listing 4.13: Binding between *DisplayedInformation* and *ContentProvider*

```
public class ContentProvider implements Inter_ContentProvider{

   public synchronized void setDisplayedInformation(IDisplayedInformation information) {
    information.portMessage.getContent();
   }
}
```

4.4.2 Mapping from Java to SOFA 2.0

SOFA is a platform for software components that uses a component model with hierarchically nested components (composite components). It describes a component by its frame (component descriptor) and component architecture. The frame is a black-box view of the component that defines its provided and required interfaces. It provides a metadata (XML files) to describe provided and required services (see Listing 4.14 and Listing 4.15). Components are interconnected via bindings among interfaces using connectors (see Listing 4.16).

Listing 4.14: DisplayedInformation.xml to provide the instances of *DisplayedInformation*

```
<?xml version="1.0"?>

<frame name="DisplayedInformation">
  <provides name="DisplayedInformation" itf-type="sofatype://IDisplayedInformation"/>
</frame>
```

Listing 4.15: ContentProvider.xml to bind the instances of *DisplayedInformation*

```
<?xml version="1.0"?>

<frame name="ContentProvider">
  <requires name="DisplayedInformation" itf-type="sofatype://IDisplayedInformation"/>
</frame>
```

Listing 4.16: binding between *DisplayedInformation* and *ContentProvider*

```
public class ContentProvider implements SOFALifecycle, Runnable, SOFAClient {

  IDisplayedInformation info = null;
  // Called during initialization of the component.

  public void setRequired(String name, Object iface) {
    if (name.equals("DisplayedInformation")) {
      if (iface instanceof IDisplayedInformation) {
      //get DisplayedInformation instance
        info = (IDisplayedInformation) iface;
        info.portMessage.getContent();
      }}}
}
```

4.5 Discussion

We can deploy a recovered cluster of classes directly onto existing component models without using our approach. Indeed, we can transform each class into a component and then assemble these components that belong to the same cluster using component composition property as a composite component. However, to compare our approach with the composite component approach, we need first to study the component composition types and component models that support these types. Table 4.2 shows the selected object-based component models and composition supported composition types. There are two types of component compositions; the first one is horizontal composition, and the second type is vertical composition. The horizontal composition means that components can be binded through their interfaces to construct component applications. The second type, vertical composition, describes the mechanism of constructing a new component from two or more other components. The new component is then called composite because they are themselves made

Table 4.2: Composition type in object-based component models

Component Models	EJB	Fractal	JavaBeans	COM	OpenCOM	OSGi	SOFA 2.0	CCM	COMPO	Palladio	PECOS
Vertical Composition	No	Yes	No	Yes	Yes	No	Yes	No	Yes	No	Yes
Aggregation		X		X	X				X		
Delegation		X		X	X		X		X		X

of more elementary components called internal components. Internal components could be accessible or visible to clients (delegation) or not (aggregation).

We can observe from Table 4.2 that there are five component models that did not support vertical composition at all (*EJB*, *JavaBeans*, *OSGi*, *CCM* and *Palladio*). Four of them provide vertical aggregation composition and six models support vertical delegation composition. However, vertical delegation composition is not appropriate because clients can access or view the internal components (violates component encapsulation). Consequently, the vertical aggregation composition could be replaced by our approach, but there are just two component models that support it.

4.6 Conclusion

In this chapter, we proposed an approach to transform recovered components from object-oriented applications to be easily mapped to component-based models. We refactored clusters of classes (recovered component) to behave as a single unit of behavior to enable component instantiation. Our approach guarantees component-based principles by resolving component encapsulation and component composition using component instances. The encapsulation of components is guaranteed by transforming the OO dependencies between recovered components which was proposed in our previous work [Alshara 2015]. Moreover, both principles applied by refactoring a recovered component source code to be instantiable, where the provided services are consumed by the component instance through its interfaces (component binding). We have shown that the source code resulting from our approach can be easily projected onto object-based component models. We illustrated the mapping onto two well known component models, OSGi and SOFA. The illustration results show that our approach facilitates the transformation process from OO applications into CB ones. Moreover, it effectively reduces the gap between recovered component architectures and its implementation source code.

Chapter 5

Model-Driven Object-Based to Component-based Software Migration

Contents

5.1 Introduction **86**

5.2 Transforming Object-oriented Applications into Component-based ones Using MDT: An Overview **87**

5.3 Transforming OOGM into CBGM: Defining the Source and the Target Metamodels and Rules of Transformation **90**

5.3.1 Metamodeling: Defining OOGMM and CBGMM 90

5.3.2 Transforming OOGM to CBGM Rules 98

5.4 Transforming CBGM into CBSMs **103**

5.4.1 Defining CBSMMs 103

5.4.2 Identifying the Variability of Transformation Rules 107

5.5 Implementation and Tools **112**

5.6 Conclusion **116**

5.1 Introduction

Model-driven engineering (MDE) has been adopted as an approach for modernizing software systems [Fleurey 2007]. It has been recognized as an efficient, flexible and reliable approach for software migration (i.e. reverse-engineering, transformation and code generation) [Schmidt 2006]. Legacy systems are represented by models to ensure a common understanding of their contents [Schmidt 2006]. These models are then used as the starting point of all the reverse engineering and transformation activities. These activities are respectively called Model Driven Reverse Engineering (MDRE) [Rugaber 2004] and Model Driven Transformation (MDT) [Mens 2006b]. Therefore, MDRE and MDT directly benefit from the genericity, extensibility, coverage, reusability, integration and automation capabilities of MDE technologies to propose efficient and profitable solutions for software migration.

In MDT, the appropriate use of models is the automated transformation to other models [Favre 2004]. Mellor et al. [Mellor 2003] define model-driven development as a MDT: *"Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing."* A prerequisite for model transformation is that the abstract syntax (e.g UML) and the semantics (e.g. metamodel) of the source and the target model are well known and understood [Schmidt 2006]. Many authors [Van Deursen 2007, Mellor 2003, Favre 2004, Ludewig 2003, Kühne 2006, Gray 2006] rightly argue that source code is also a model. Therefore, transformation from source code to another one can be considered as an MDT activity.

MDT can be implemented based either on classical programming languages like Java, or on model-based transformation languages like the domain of Extensible Stylesheet Language Transformations (QVT, XSLT[1]). These languages provide operations on models such as searching for items or patterns, as well as the creation, deletion and modification of model entities [Weis 2003]. Sendall and Kozacynski [Sendall 2003] state that abstractions are the core aspects of a transformation language. They should be intuitive and cover a large proportion of all occurring cases. Therefore, model-based transformation languages are more appropriate for MDT than classical programming languages. Model-based transformation languages are distinguishable as textual or graphical. For example, the OMG defined the textual language Query View Transformation (QVT) to describe transformations between models. Sendall and Kozacynski [Sendall 2003], Weis et al. [Weis 2003] and Moody [Moody 2009] have proposed graphical notations as Model-based transformation languages.

In this chapter, we propose an approach to transform object-oriented applications into component-based ones based on MDE. It is about model-to-model transformation using MDT. We present two contributions: First, we propose transformations from OO generic model to CB generic one and second, we propose transformations from CB generic model to CB specific models.

[1] https://www.w3.org/standards/xml/transformation

In the first contribution: Firstly, predefined generic metamodels are respectively proposed for both OO source codes (e.g. Java, C++, etc.) and CB source codes (e.g.OSGi, CCM, etc.). Then transformation rules between these generic metamodels are proposed. These rules aims to transform OO dependencies to interface-based ones that have been presented in previous chapters like method invocation, type dependency, instantiation, inheritance, exception handler, component instance and component descriptor. However, in contrast to contributions that have been proposed in previous chapters, we used MDT rather than classical programing languages to transform these dependencies.

In the second contribution: firstly, we identified and represented variabilities between metamodels of many existing component platforms (e.g. OSGi metamodel, CCM metamodel, etc.). Then we defined a feature model representing commonalities and variabilities between transformation from generic CB metamodel to all variants of specific CB metamodel.

5.2 Transforming Object-oriented Applications into Component-based ones Using MDT: An Overview

For the transforming of OO models to CB ones, the transformation could be seen as two phases as shown in Figure 5.1. The first phase operates to transform object-oriented generic model (OOGM) into component-based generic model (CBGM). CBGM defined the common elements constitute the most component-based models. It intermediates and connects the two phases of the transformation. The second phase operates to transform CBGM into component-based specific models (CBSMs). It aims to complete the transformation by considering specificities of each target component model. It is based on variable transformation rules.

Figure 5.2 describes the models handled by the transformation process. Firstly, object-oriented generic metamodel (OOGMM) and component-based generic metamodel (CBGMM) are defined. The term generic here in both OOGMM and CBGMM means that the metamodels cover respectively the majorities of OO and CB models. For example, OOGMM is a metamodel that represents many object-oriented models (e.g. Java, C++, Smalltalk, etc.), as same as, CBGMM is a metamodel that represents common principles for many component models (e.g. OSGi, SOFA, Fractal, etc.). Secondly, the transformation rules from OOGM (confirm to OOGMM) to CBGM (confirm to CBGMM) are identified. Thirdly, a transformation feature model (TFM) is identified based on the variability between component-based specific metamodels (CBSMMs). Finally, the transformation rules from CBGM to component-based specific models (CBSMs) are identified using TFM to produce component-based source code.

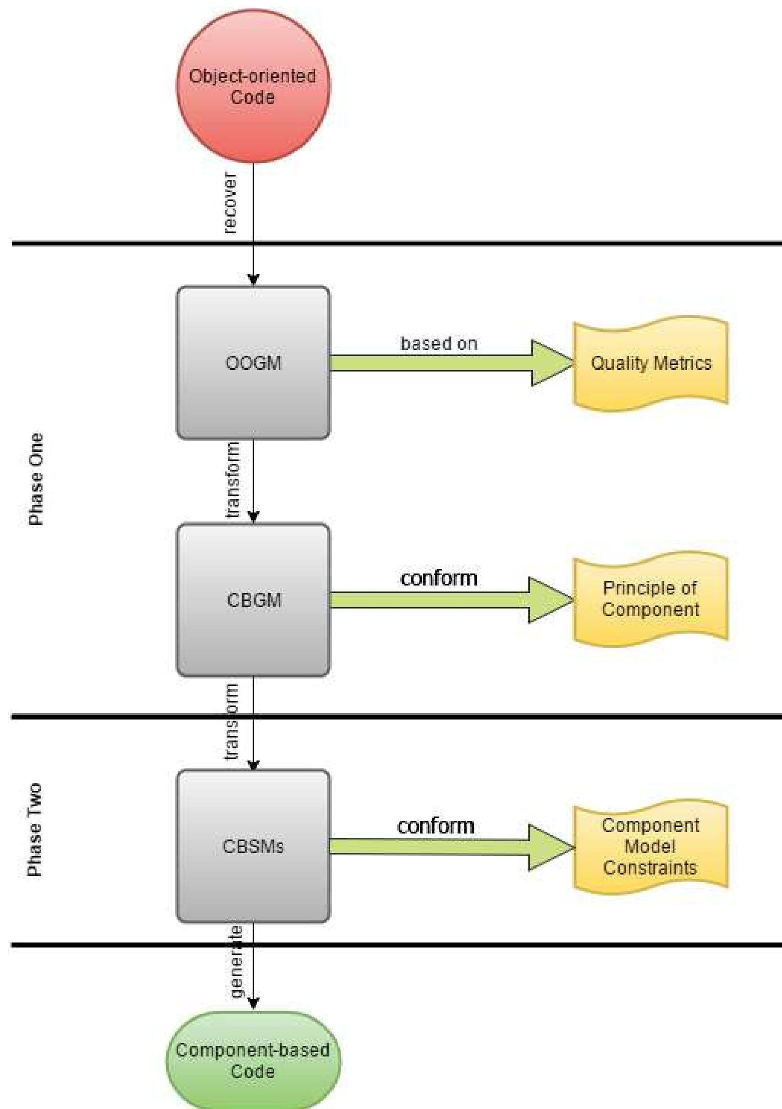


Figure 5.1: The phases of transformation process.

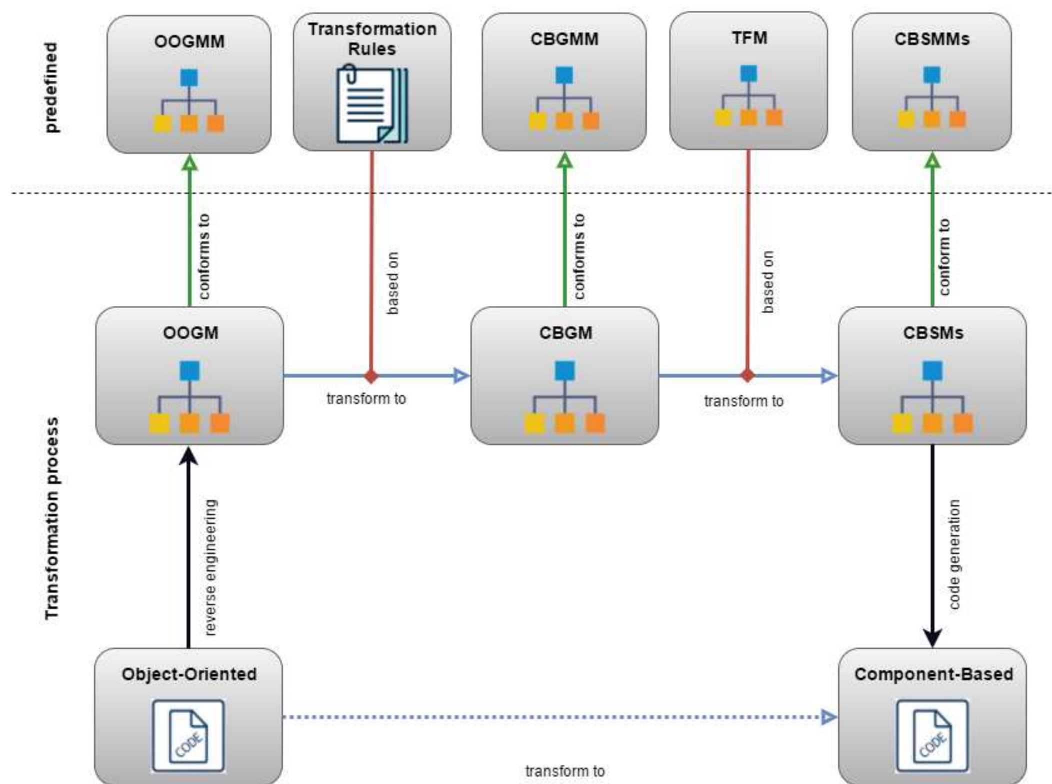


Figure 5.2: Models handled in the transformation process.

5.3 Transforming OOGM into CBGM: Defining the Source and the Target Metamodels and Rules of Transformation

This phase consists of two steps: metamodeling and MDT rules. In the first step OOGMM and CBGMM are defined, text-to-model transformations are also defined to abstract information contained in the object-oriented source code as OOGM. The OOGM includes elements related to the architecture recovered based on the reverse engineering process (an example of approach to recover software architecture is given in Sec 2.4). In the second step, the transformation rules are defined to transform OOGM into CBGM. The following subsections give details for these two steps.

5.3.1 Metamodeling: Defining OOGMM and CBGMM

To transform object-oriented code to component-based code using model-driven, we must represent the source and the target in a canonical model. For instance, to transform instance A to instance B, we must abstract A as a model Ma that conforms to MMa meta-model and B as a model Mb that conforms to MMb meta-model. After that, we transform Ma to Mb based on their meta-models. Finally, we generate B from Mb (see Figure 5.3 [2]).

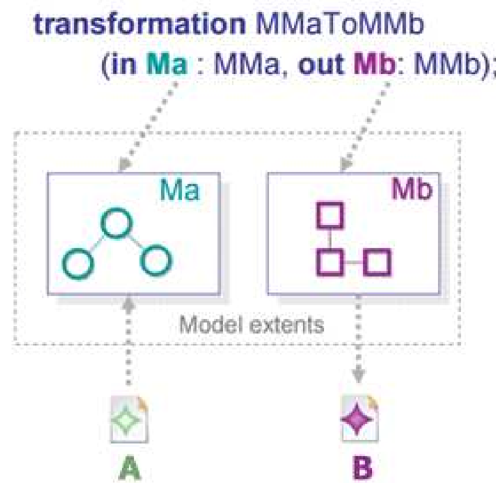


Figure 5.3: Model transformation.

In our case, we need to identify metamodels for both object-oriented and component-based source codes. As noted before, information related to the recovered architecture constitutes a part of the OOGMM. For this purpose, we proposed to use FAMIX metamodels(proposed by the MOOSE project [Demeyer 2001]).

[2]The used notation is from QVT.

FAMIX is an extensible family of metamodels which constitute a language-independent representation of object-oriented source code. Because the extensibility and independent ability of FAMIX, we argue that FAMIX is a good support for MDT [Demeyer 1999, Tichelaar 2000]. This is illustrated by many research projects where FAMIX has been used to support many model-driven analysis and reengineering tasks (e.g. [Ducasse 2000, Ducasse 2011b]). In addition, parsing technology exists to export the meta information of object-oriented languages like C++, Java, Smalltalk and Ada to the FAMIX metamodels [Ducasse 2011b, Kobel 2005].

The core of the FAMIX model is depicted in Figure 5.4. It includes the main object-oriented structural entities like Class, Method and Attribute. Also it defines other entities representing associations between structural entities like inheritance, method invocation and attribute access. The complete FAMIX model provides much more information such as functions and formal parameters, and additional relevant information for every model entity. The complete specification of the model can be found in [Ducasse 2011b].

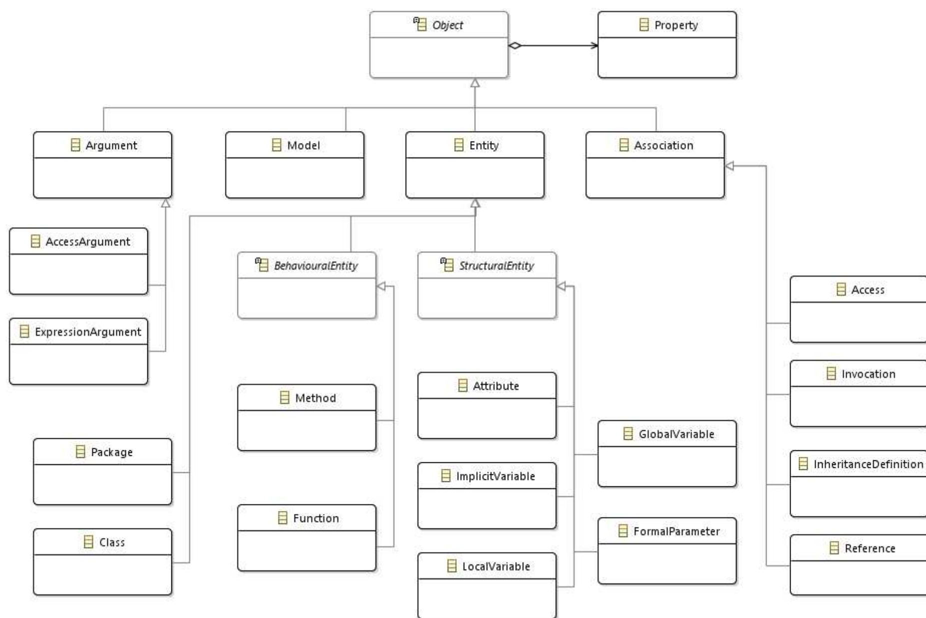


Figure 5.4: The core of FAMIX metamodel.

5.3.1.1 The Core of OOGMM

In our approach, we proposed an extension of FAMIX and we considered it as our OOGMM. As noted before, this metamodel includes entities representing both abstractions of object-oriented source code and the recovered architecture. Figure 5.5 shows the OOGMM as extension of FAMIX. The OOGMM includes an entity (class) representing all clusters of classes (instance) resulting from the architecture recovery. Each instance of this class provides information about provided and required

interfaces of the corresponding cluster.

In addition to the use of FAMIX to represent OOGMM, we also used another extension of it to represent CBGMM. The reason behind the use of FAMIX to represent component-based entities is related to the nature of component-based languages considered as target of our transformation process. We consider that these languages are extensions of object-oriented languages (e.g. OSGi as extension of Java, CCM as extension of C++). Figure 5.6 shows the metamodel for the CBGMM.

5.3.1.2 The Core of CBGMM

Three classes are added to FAMIX to be able to represent component-based entities: *Component*, *Interface* and *Descriptor*. An instance of *Component* class provides information about both the list of interfaces of the corresponding component and its descriptor. An instance of *Interface* class provides information about the methods composing the interfaces provided and required by a component. The *Descriptor* class represents component descriptors. The instances of each of these three classes are built as a result of the architecture recovery phase.

5.3.1.3 Modeling Dependencies between Object-oriented Entities

Dependencies between component-based entities resulted from OO dependencies between the corresponding clusters are represented by five types of associations: access attribute, object reference, method invocation, inheritance relationship, and exception handling. Figure 5.7 shows the first three dependencies: Attributes can be accessed, methods can be invoked, and references can be used no matter where these Attributes, methods, and references belong to. Therefore, attributes, methods, and references can be used, as in object-oriented applications, even they belong to different clusters. As each cluster should be transformed into a component, in the case where these dependencies are related to entities belonging to different clusters such use must be not allowed. For example, it will be prohibited that a method belonging to a cluster A invokes a method belonging to cluster B. To reflect this prohibition, we used OCL constraints. Figure 5.8 depicts the dependencies described above and the corresponding constraints in CBGMM.

The fourth type of dependency is related to OO inheritance. A class from a cluster A can inherit from another class from a cluster B (see Figure 5.9). This dependency must be prohibited when considering that a component is an implementation of a cluster. This dependency prohibition is represented as constraint in CBGMM (see Figure 5.10).

The fifth dependency type is the OO exception handling. In OOGMM, the exception handling is modeled by three entities: *DeclaredException*, *ThrownException*, and *CaughtException*. *DeclaredException* class models the creation of an exception object. *ThrownException* class models the method that throws an exception. *CaughtException* class models a method handler that caches an exception. Figure 5.11 shows the modeling of the exception handling in OOGMM.

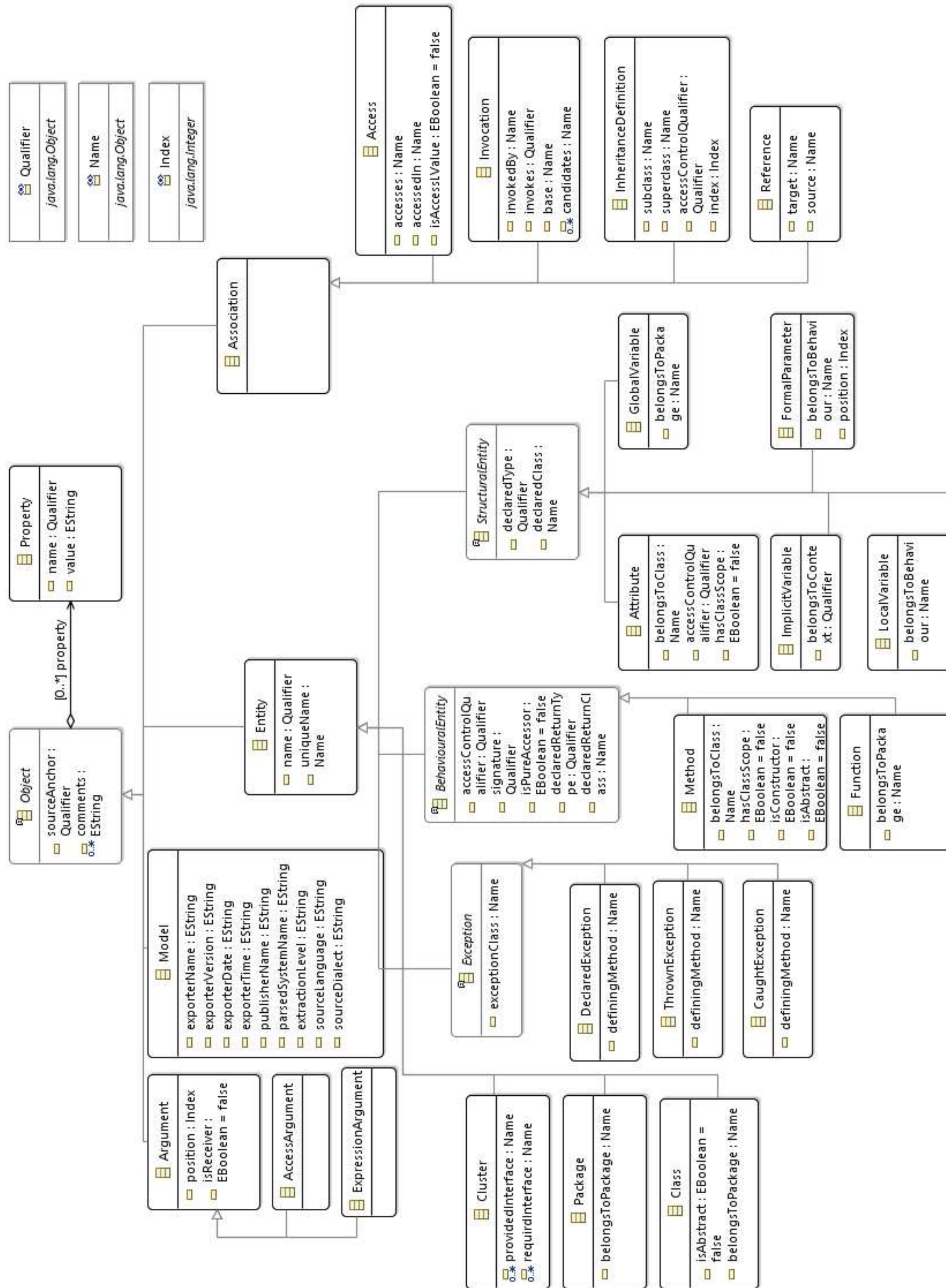


Figure 5.5: The core of the object-oriented architecture recovery metamodel.

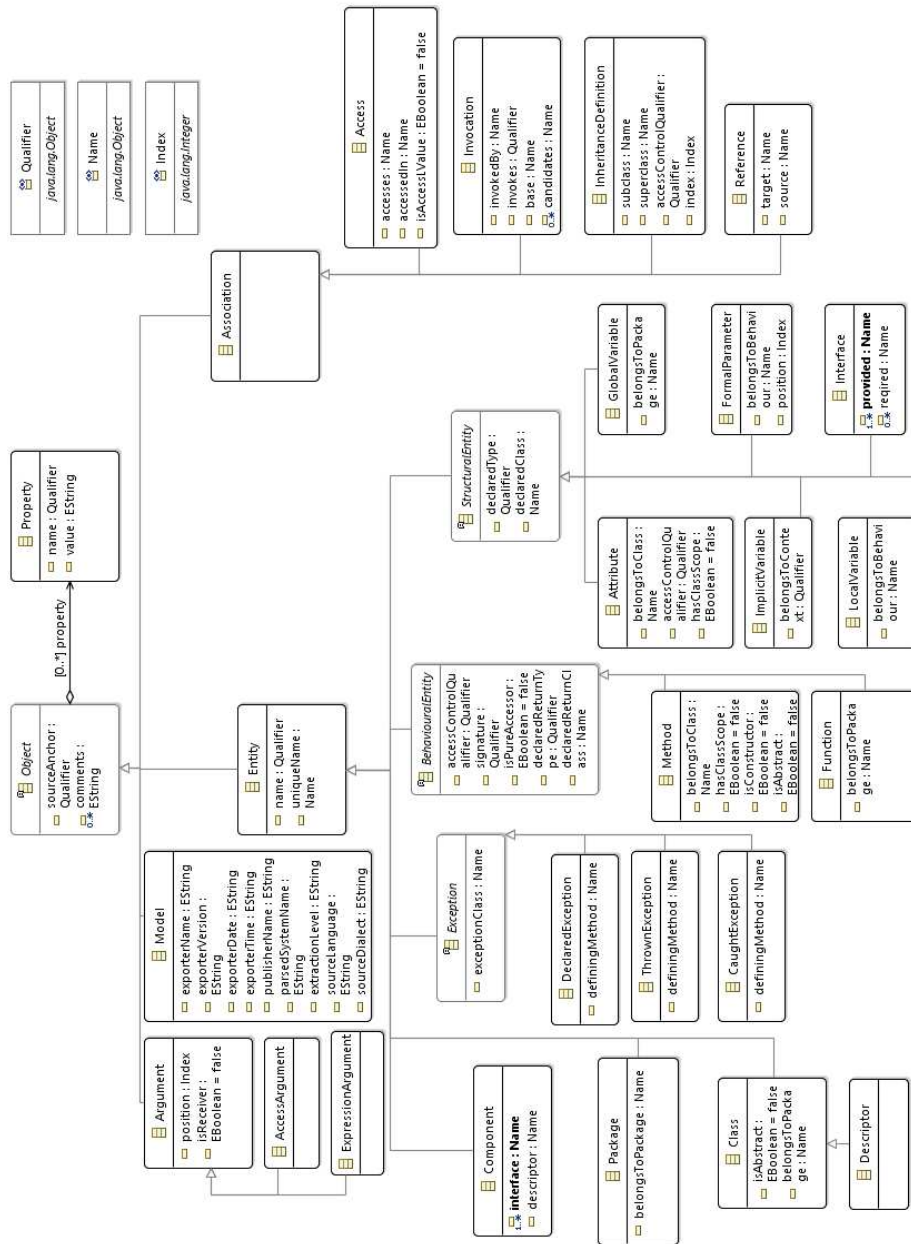


Figure 5.6: The core of the CBGMM.

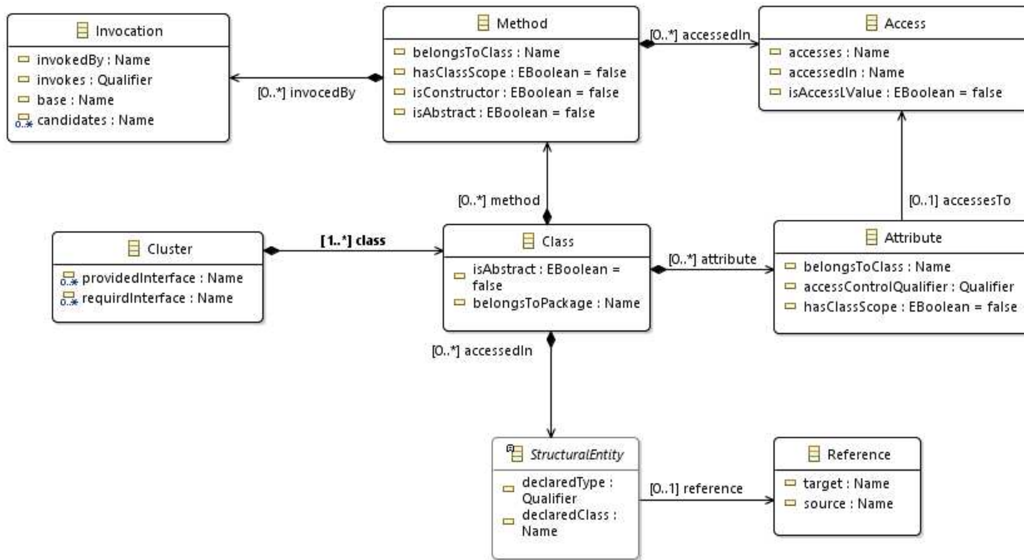


Figure 5.7: Access attribute, object reference, method invocation in object-oriented architecture recovery metamodel.

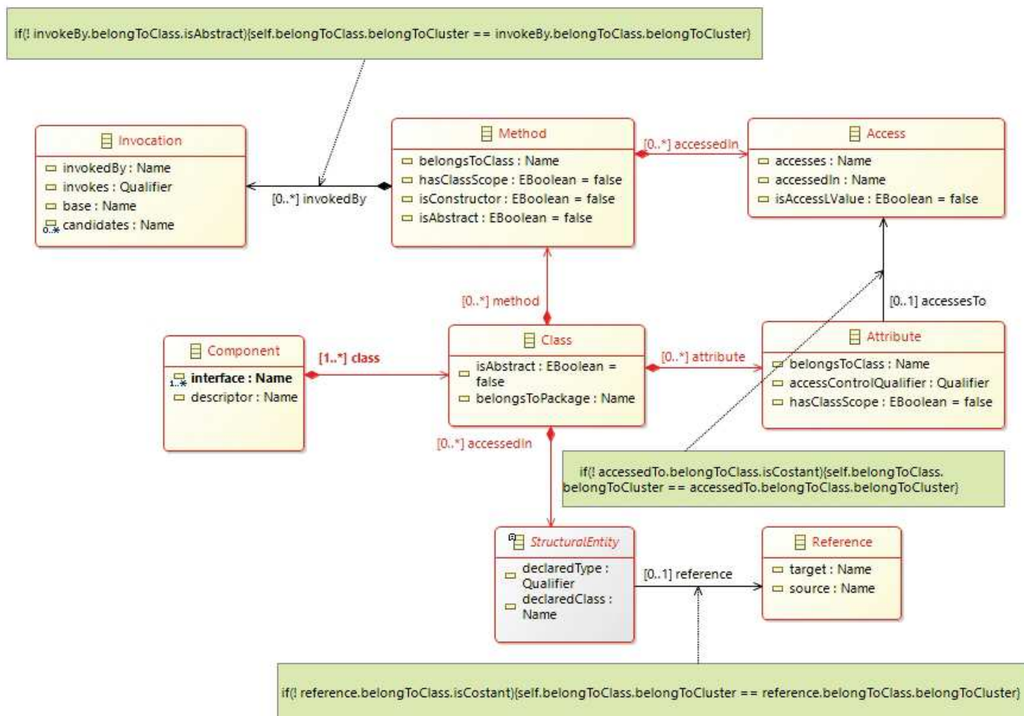


Figure 5.8: Access attribute, object reference, method invocation in CBGMM.

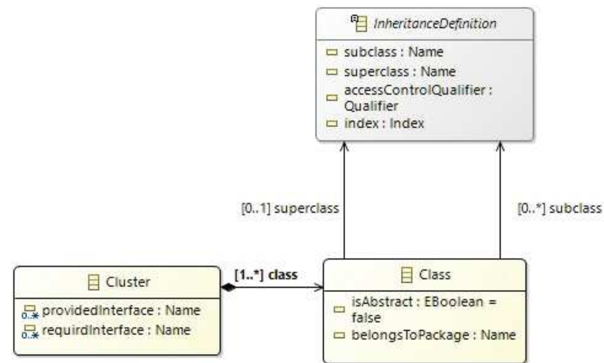


Figure 5.9: Inheritance relationship in object-oriented architecture recovery meta-model.

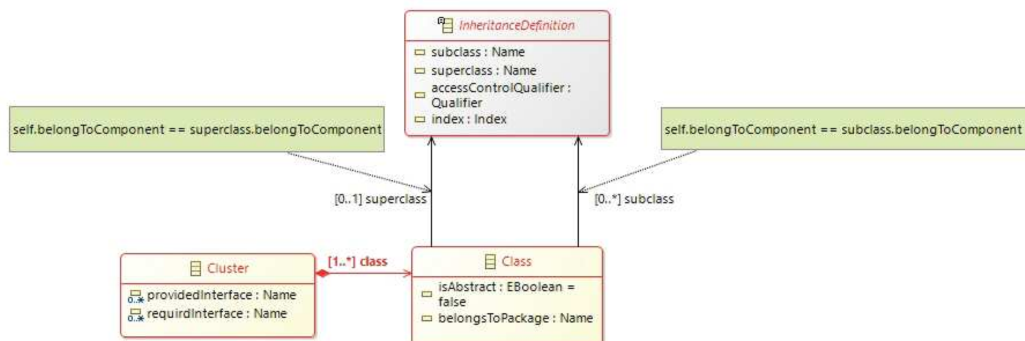


Figure 5.10: Inheritance relationship in CBGMM.

When considering that a component is an implementation of a cluster of classes, exception handling needs to be constrained to reflect component encapsulation. The first constraint is that a class belonging to a component can not instantiate an exception class that belongs to another component. The second constraint is that if a class C1 defining a method M1 considered as direct or indirect caller of a method M2 throwing an exception, then a class C2 defining the M2 method must belong to the same component of C1. The last constraint aims to prohibit a situation where the class of an exception object belongs to a component different from that including the class that defines the catcher method. Figure 5.12 shows the modeling of the exception handling in CBGMM.

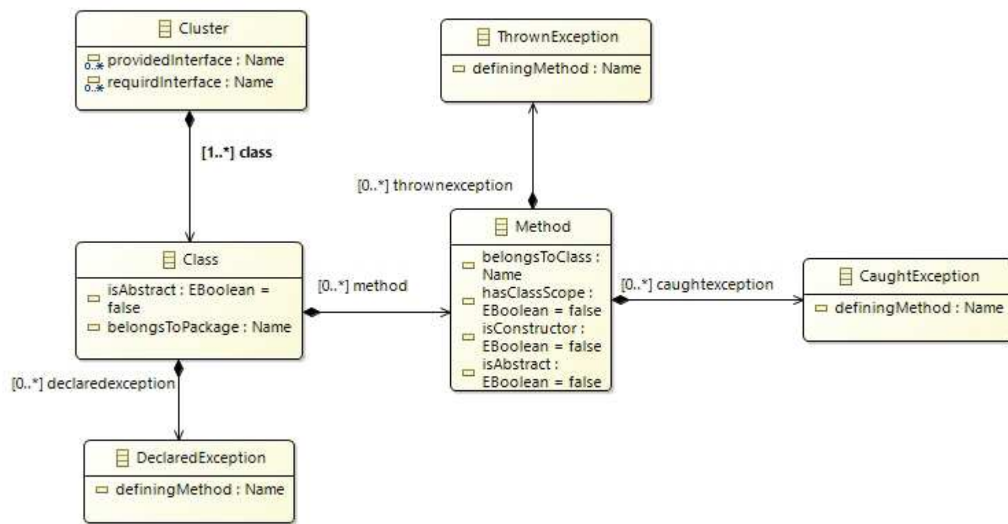


Figure 5.11: Exception handling in object-oriented architecture recovery metamodel.

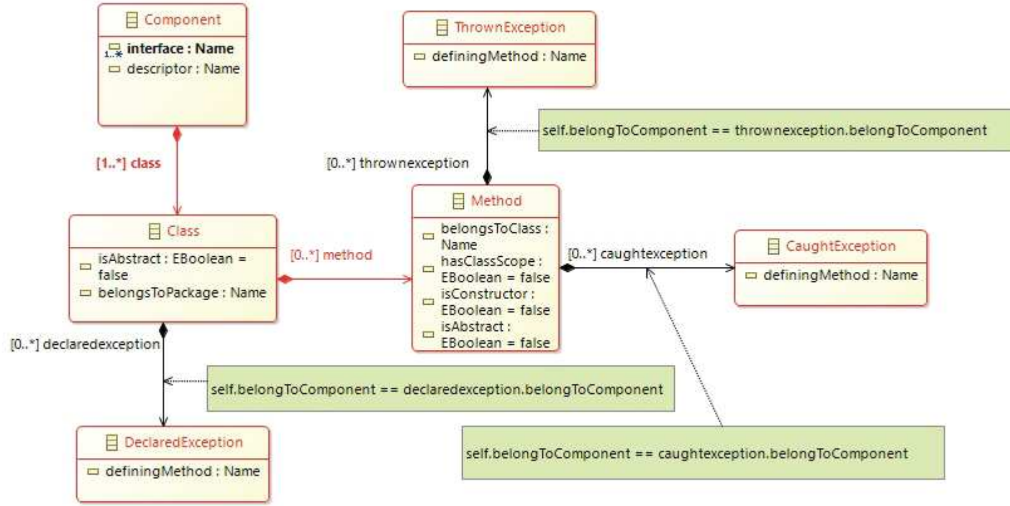


Figure 5.12: Exception handling in CBGMM.

5.3.2 Transforming OOGM to CBGM Rules

Algorithms 1, 3, 4 and 5 allow transformation from OOGM to CBGM. The syntax used in these algorithms is based on information modeled by OOGMM and CBGMM. For example, *ClassA* $\in$ *ClusterB* means the existence of composition relationship between the ClusterB and the ClassA in the OOGM instance of OOGMM. Another example is the syntax code *SubClass.isInherit(SuperClass)* where the corresponding inheritance information is available in OOGM.

Algorithm 1 Transforming instantiation and type dependency

1: **procedure** INSTANTIATION–TRANSFORMATION
2: **Pre-Conditions:**
3: $classA \in Cluster1$ & $classB \in Cluster2$
4: $classA$.isInstantiate($classB$) = true|$classA$.fasTypeOf($classB$) = true
5: **Rules:**
6: $interfaceB \leftarrow ExtractInterface(classB)$
7: $factory \leftarrow createFactory(classB, interfaceB)$
8: $Cluster2$.add($factory$)
9: **for** each Class $boundary \notin Cluster2$ **do**
10: $boundary \leftarrow replace(factory, classB)$
11: $boundary \leftarrow replace(interfaceB, classB)$
12: **end for**
13: **end procedure**

Algorithm 2 Extract interface

```
 1: function ExtractInterface(Class class)
 2:     for each Attribute attribute ∉ class & isVisible(attribute) do
 3:         class ← createSetter(attribute)
 4:         class ← createGetter(attribute)
 5:     end for
 6:     Interface interface
 7:     for each Method mothod ∉ class & isVisible(class) do
 8:         interface ← getSignature(mothod)
 9:     end for
10:     class.implements(interface)
11:     return interface
12: end function
```

Algorithm 3 Transforming inheritance relationship

```
 1: procedure Inheritance–Transformation
 2: Pre-Conditions:
 3:     subClass ∈ Cluster1 & superClass ∈ Cluster2
 4:     subClass.isInherit(superClass) = true
 5:     superClass.isAbstract() = false
 6: Rules:
 7:     subClass.removeInherit(superClass)
 8:     interfaceSub ← ExtractInterface(subClass)
 9:     interfaceSuper ← ExtractInterface(superClass)
10:     interfaceSub.inherit(interfaceSuper)
11:     applyDelegationPattern(subClass, superClass)
12:     subClass ← addAttribute(interfaceSuper, _super)
13:     superClass ← addAttribute(interfaceSuper, _this)
14: end procedure
```

Algorithm 4 Transforming abstract superclass inheritance relationship

1: **procedure** ABSTRACT INHERITANCE–TRANSFORMATION
2: **Pre-Conditions:**
3: $subClass \in Cluster1$ & $superClass \in Cluster2$
4: $subClass$.isInherit($superClass$) = true
5: $superClass$.isAbstract() = true
6: **Rules:**
7: $subClass$.removeInherit($superClass$)
8: $interfaceSub \leftarrow ExtractInterface(subClass)$
9: $interfaceSuper \leftarrow ExtractInterface(superClass)$
10: $interfaceSub$.inherit($interfaceSuper$)
11: $proxy \leftarrow createProxy(subClass, superClass)$
12: $Cluster2$.add($proxy$)
13: $proxy$.inherit($superClass$)
14: $proxy$.implements($superClass$)
15: $superClass \leftarrow addAttribute(interfaceSuper, _this)$
16: **for** each Method $abstarctMethod \in superClass$ & $abstarctMethod$.isAbstarct() = true **do**
17: $abstarctMethod$.backward(_this,subClass)
18: **end for**
19: **end procedure**

Algorithm 5 Transforming exception handling

1: **procedure** EXCEPTION–TRANSFORMATION
2: **Pre-Conditions:**
3: $exceptionInstantiation \in Cluster1$ & $(throwMethod \notin Cluster2 \mid handlerMethod \notin Cluster2)$
4: **Rules:**
5: $exceptionInstantiation.getMethod \leftarrow setReturnInstruction(exceptionObject)$
6: **for** each Method $throwMethod \notin Cluster1$ **do**
7: $throwMethod \leftarrow setReturnInstruction(eceptionObject)$
8: **end for**
9: $handlerMethod.getHandlerBlock \leftarrow replaceWithIfElse(exceptionObject)$
10: **end procedure**

Some of these transformations must be executed in order to reflect their dependencies. For example, the transformation of inheritance needs another transformation which transformation problem which is an instantiation one. For example, the transformation of inheritance caused a new transformation problem which transformation problem is an instantiation one. Therefore, these transformations must operate in an order that guarantees resolving all problems without forgetting some caused by resolving another one. Figure 5.13 depicts the order of the transformation problem and the new transformation problem resulted from each one.

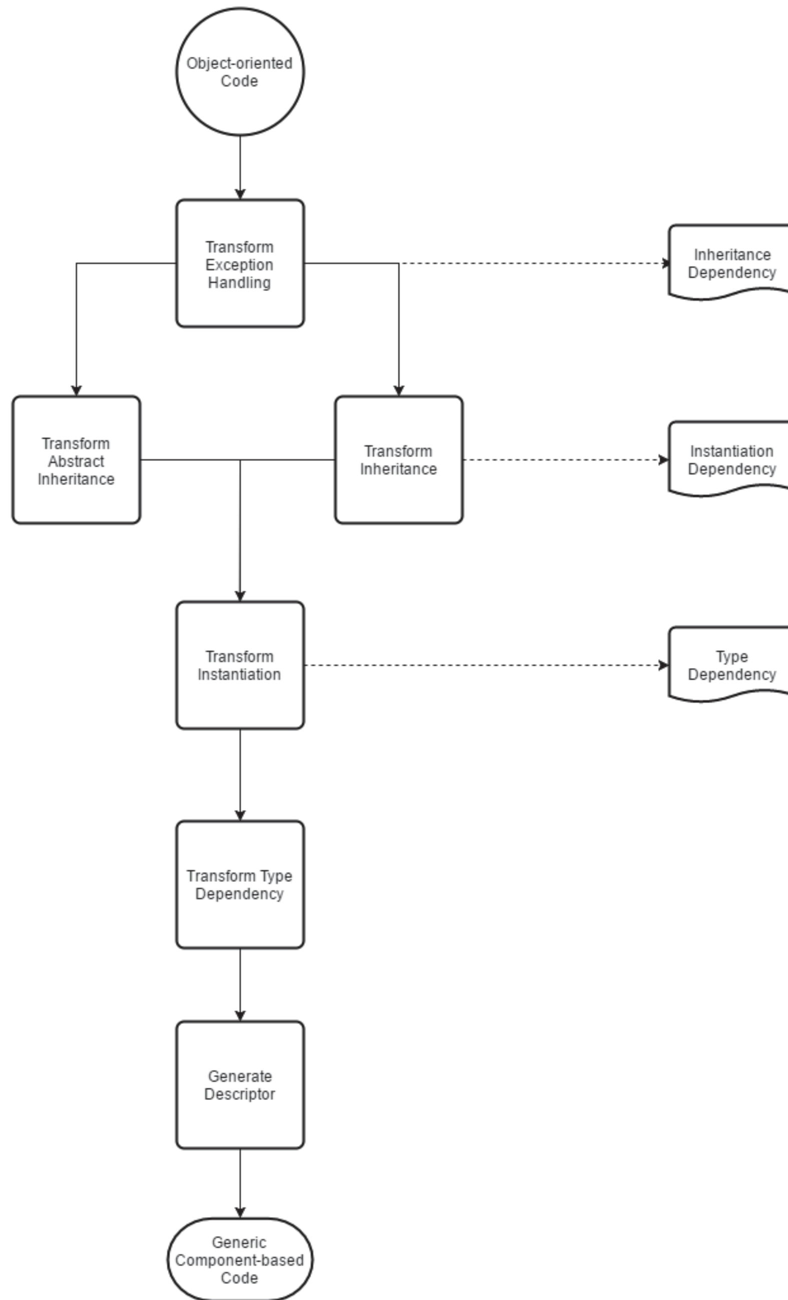


Figure 5.13: The order of the transformation problem into CBGM.

5.4 Transforming CBGM into CBSMs

This phase consists of three steps. The first step consists of the definition of a set of specific component-based metamodels respectively for a set of component-based source code (e.g. OSGi source code). The second step aims to identify commonalities and variabilities between CBSMs. The third step aims to define a feature model to represent commonalities and variabilities between transformations from a CBGM to each of the corresponding CBSMs based on commonalities and variabilities of CBSMs.

5.4.1 Defining CBSMMs

OSGi This model provides an implementation of dynamic modules (i.e. capacity to add and remove modules at runtime) for the Java platform. The OSGi specification defines a framework for managing the life-cycle of a set of components called *Bundle*. The framework also defines the concept of *internal service* whose registration is managed in a dynamic way by the central registry of the OSGi platform. A service is any object registered in the OSGi Service Registry and then can be looked up using its interface name(s). The only prerequisite is that a service must implement some Java interfaces. It can define additional meta-information in the form of a manifest file and declare dependencies to other bundles based on Java packages notion.

Figure 5.14 shows the metamodel for *Bundle* implementation. A *Bundle* is composed of a set of Java classes and Java interfaces. Moreover, it is composed of services represented as XML files. Each file contains an information about provided and required services using interfaces types.

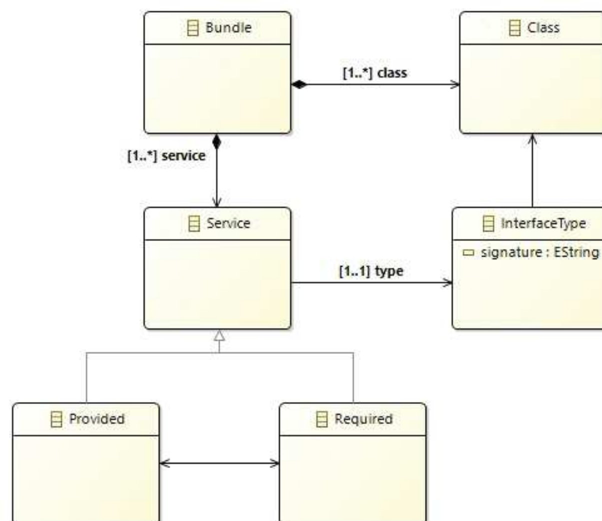


Figure 5.14: OSGi component metamodel.

SOFA The SOFA 2 framework offers a hierarchical component model where a composite component can be composed of primitive or composite ones. A primitive component is defined by its descriptor called *Frame*. A *Frame* is an XML file that describes the provided and required component's services. As with OSGi, interface types are used to identify the provided and required services. The provided and required services are bound at run time.

Figure 5.14 depicts the metamodel of SOFA component. A component is composed of a set of Java classes and Java interfaces in addition to frames. A *Frame* has two types: *Provided* and *Required*. A class implements a Java interface that could be a type of provided or required services represented into a *Frame*.

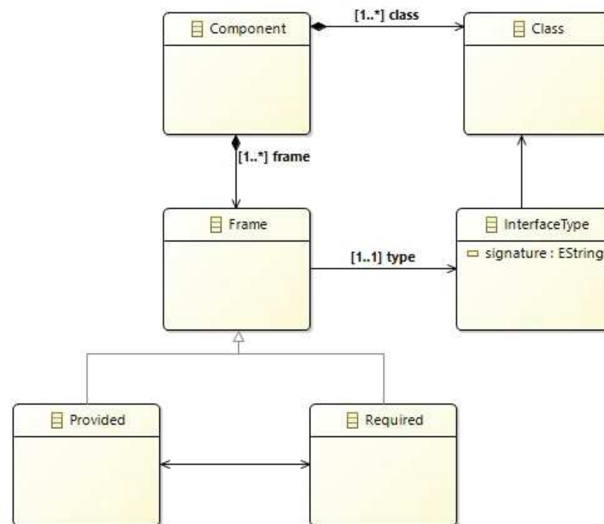


Figure 5.15: SOFA component metamodel.

CORBA Component Model (CCM) It provides a support for remote procedure calls independently of the communication protocol, the programming language, the operating system and the hardware platform. Interface Definition Language (IDL) is used to describe procedures and functions that may be remotely invoked. It provides the transformation from IDL to various programming languages like C++ or Java.

Figure 5.16 shows that a *CCMComponent* is composed of a set of classes and interfaces. The provided interface and required interfaces respectively called *Navigation* and *Receptacle*.

Fractal Fractal has three kinds of components: primitive, composite, and base component. A composite component is composed of a set of primitive or composite components. A base component does not expose any interface. Fractal components

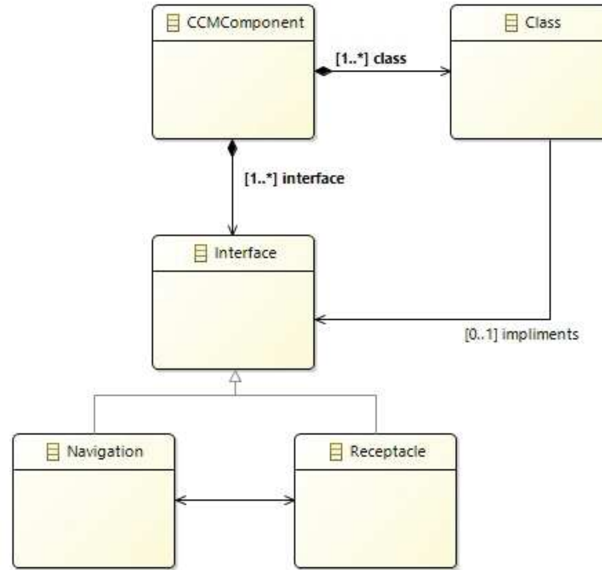


Figure 5.16: CCM metamodel.

consist of two parts: a controller and a content. The controller is a set of provided and required interfaces. The content is a class that implements component services. Fractal ADL (architecture description language) is provided to describe architectures of Fractal components. There are currently two reference implementations: Julia[3] (Java) and Cecilia[4] (C/C++) and other experimental implementations such as FractTalk [Coupaye 2006] (Smalltalk) or FractNet [Escoffier 2005] (.Net).

Figure 5.17 shows the Fractal component metamodel. A Fractal component consist of a set of classes and interfaces. the provided and required interfaces are called *Server* and *Client* respectively. *FractalADL* used as a component or architecture descriptor. It has the references of components and their interfaces to connect them.

Component Object Model (COM) COM components are not restricted to specific programming language or platform (although Microsoft Windows is the main support of this platform). The COM component model allows to specify only provisions of components in the form of provided interfaces. COM provides capabilities for introspection of provisions of components, but the required interfaces must be specified only within the source code.

Figure 5.18 depicts the metamodel of COM. A *COMComponent* is composed of a set of classes and functional interfaces. For binding between components, the interface discovery mechanism is implemented through the notion of a special interface called IUnknown that must be implemented by every COM component. The purpose of IUnknown is twofold: (i) it allows the dynamic querying of a component

[3]http://julialang.org
[4]http://fractal.ow2.org/cecilia-site/2.0/

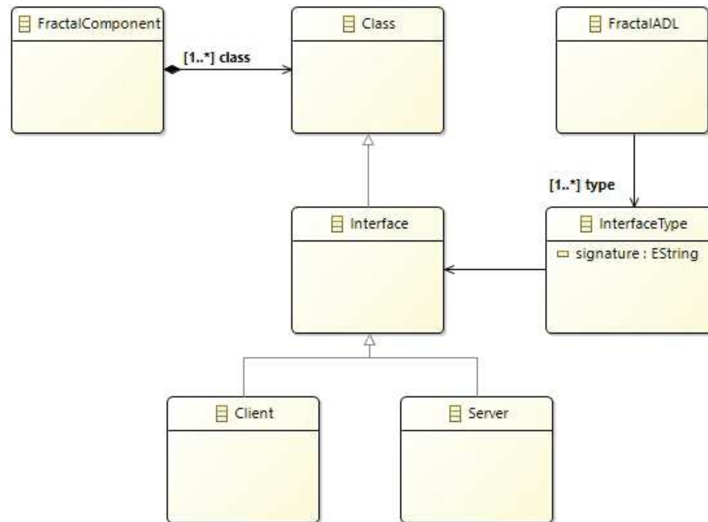


Figure 5.17: Fractal component metamodel.

(QueryInterface() operation) to find out if the component supports a given interface (in which case, a pointer to that interface is returned), and (ii) it implements reference counting in terms of the number of clients using components' interfaces. Reference counting is used to garbage collect components when they no longer have any clients.

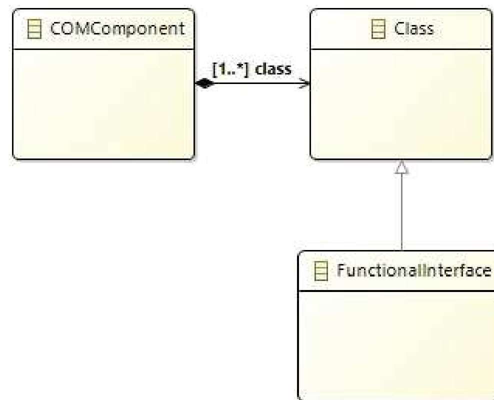


Figure 5.18: COM metamodel.

OpenCOM It is based on Microsoft COM model. The key concepts of OpenCOM are components, interfaces, receptacles. Each component implements a set of provided and required interfaces called receptacles and interfaces respectively. A receptacle describes a unit of service requirement. An interface expresses a unit of service provision. At runtime, an interface and a receptacle of the same type are

bound.

Figure 5.19 depict the metamodel of OpenCOM.

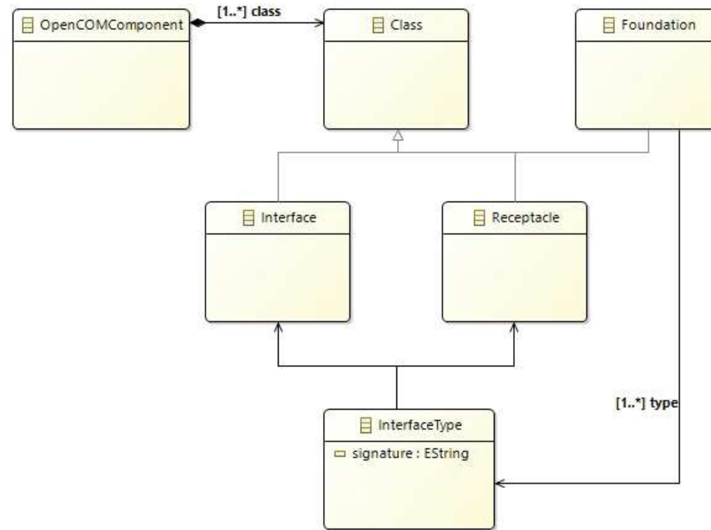


Figure 5.19: OpenCOM metamodel.

JavaBeans A Javabean is "*a reusable software component that can be manipulated in a graphical development environment*" [Hamilton 1997]. It is particularly well suited for building graphical user interfaces. A graphical component is called a widget such as buttons, menu bars, etc.. However, it should be noted that all Javabeans are not necessarily graphical components, their usage can be much wider. A JavaBean is an instance of a Java class that has attributes, methods in standard concepts in Java. Figure 5.20 illustrates the metamodel of Javabean component. A *JavaBeanComponent* composed of a set of Java classes. A Bean is a Java class that acts as a component descriptor. It has all services that the component provides.

Enterprise Java Beans (EJB) This component model primarily used for a distributed client-server architecture, where clients connect to a server in order to access services provided by the server. A Bean exports its services through a remote interface. A remote interface is a standard Java interface. Communications between beans are performed using RMI (Remote Method Invocation). Figure5.21 shows the meta model of EJB component. The component called *module* is composed of a set of Java classes and remote interfaces. The classes that implement the remote interfaces act as component descriptor.

5.4.2 Identifying the Variability of Transformation Rules

To define all elements needed for our MDT process, we need in addition to the definition of the OOGMM, the CBGMM (see Figure 5.22), the CBSMMs, and the

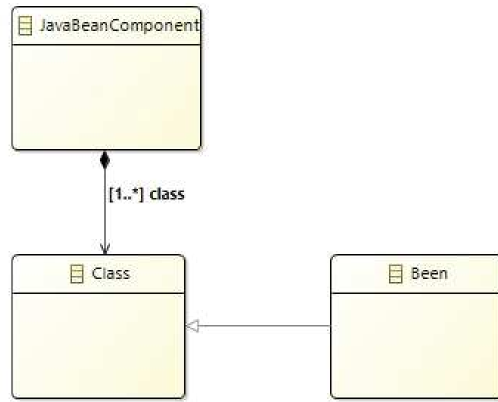


Figure 5.20: JavaBeans component metamodel.

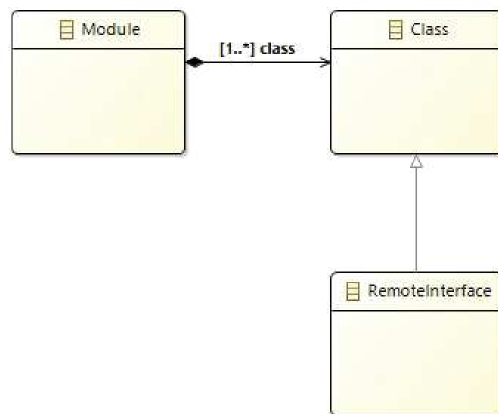


Figure 5.21: EJB component metamodel.

transformation rules from OOGM to CBGM, to define transformation rules from a CBGM to a CBSM.

As shown in the (Sec 5.4.1), these CBSMMs share common features but also have many others features that are specifics (variables). Therefore, transformation rules definition can be based on the identification of commonalities and variabilities between CBSMMs. We identified four main variable features that can impact the definition of transformation rules from an OOGM to an CBSM: component descriptor, service description, interface description, and explicit required interface.

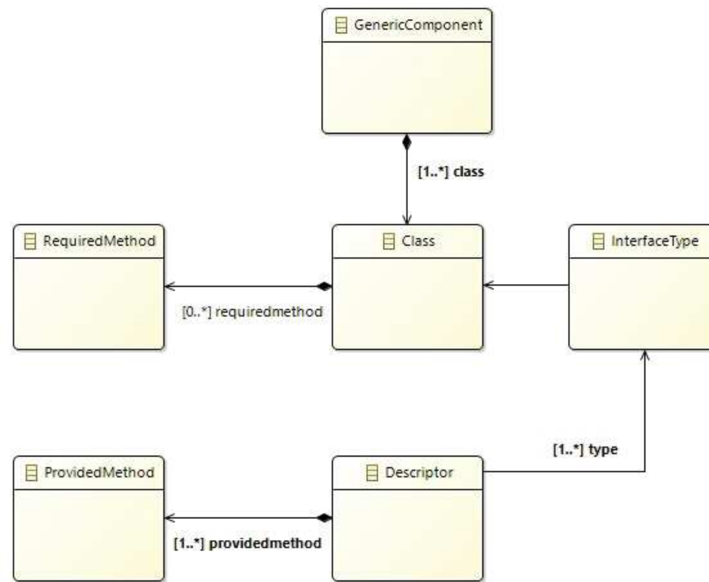


Figure 5.22: Component-Based Generic Metamodel (CBGMM).

Component descriptor. Some of the CBMM define explicit descriptors and others have implicit ones. An explicit descriptor could be given in object-oriented languages or in a specific-domain language (e.g. Component Definition Language (CDL) or Architecture Description Languages (ADL)). In the first case, it is an object-oriented class whose properties allow to specify component interfaces and binding methods. OSGi, SOFA, OpenCOM and JavaBeens are component models which include this type of descriptor. An explicit descriptor can be transformed from a generic component model to a specific one by refactoring the original one. The refactoring concludes by adding missing implementation and interfaces in addition to implement binding methods if necessary. In the second case, where a descriptor needs to be described in a specific-domain language, its generic definition in a CBGM is transformed into its specific-domain language description. An example of this kind of descriptor is Fractal component model.

In the case where descriptor is implicit (i.e. Components without a descriptor like in CCM, COM and EJB), the description of a component (i.e. interface and binding

implementations) will be embedded and distributed into component implementation (source code). Therefore, no specific transformation is needed.

Service description. Component models provide and consume services in two ways: declaratively or imperatively. When using declarative service, a component declares its provided and required references of services in an XML-like document like OSGi and SOFA. At run time, the component platform processes the XML-like document to create component instances that provide services and register them in a registry. After that, components lookup on this registry to find their required reference services for binding. This process is depicted in Figure 5.23.

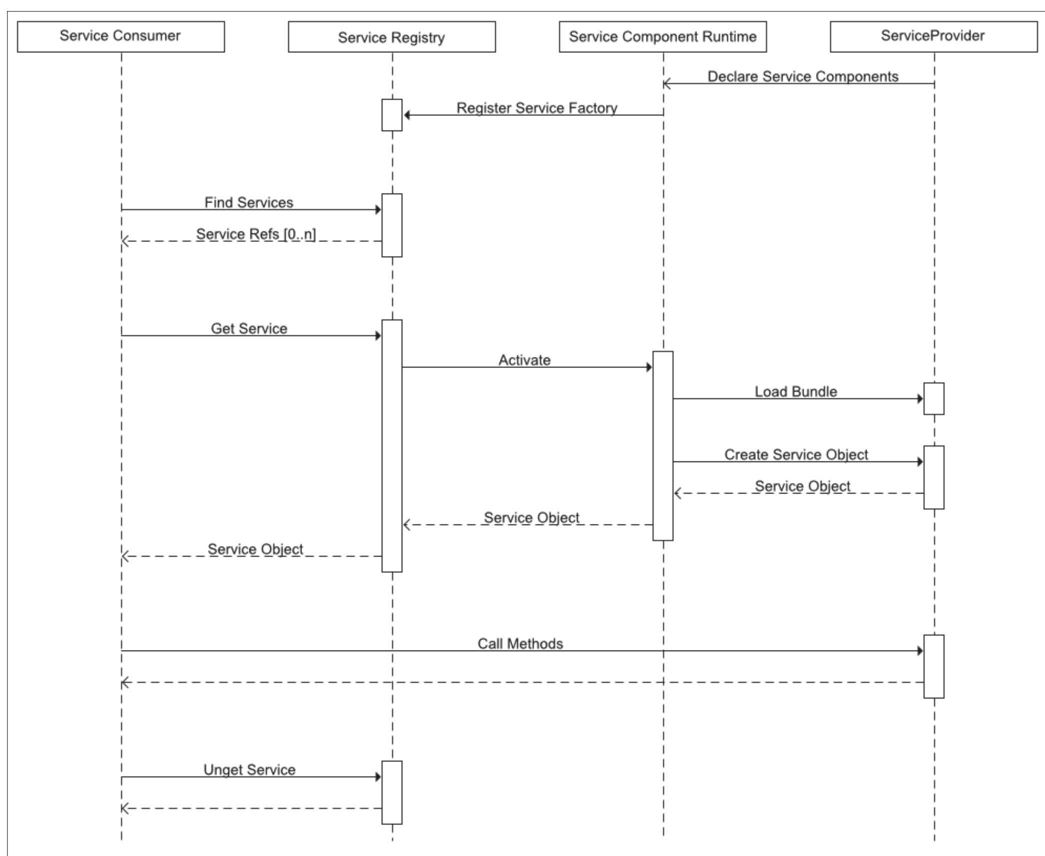


Figure 5.23: Declarative Service life cycle.

Imperative services are provided and required by components using the standard method call in object oriented. A component calls its required services (methods) using its service instance (object interface). CCM, Fractal, COM, OpenCOM, JavaBeans and EJB are component models that define these type of services.

Interface description. Some component models use an independent specification language to describe component interfaces like CCM and its Interface Description Language (IDL). This allows to describe an interface in a language-independent

way, enabling communication between software components defined by different programing languages. Other component models like OSGi, SOFA, Fractal, COM, OpenCOM, JavaBeans and EJB, use a standard object oriented interface (e.g. Java interface) to describe component interfaces.

Explicit required interface. Required interfaces of components can be either explicit or implicit. Explicit required interfaces define the required methods or services like in OSGi, SOFA, Fractal, OpenCOM and CCM. Implicit required interfaces are embedded in the component source code like in COM, JavaBeans and EJB.

5.4.2.1 Model-driven Transformation Feature Model.

A transformation feature model (TFM) is a compact representation of all needed transformations from a CBGM to a CBSM, where a transformation is considered as a feature. In addition to the common features, the TFM includes variable transformations specific to only some CBSMs. These variable features (transformations) reflect differences between elements of the CBSMs. The transformation chain from a CBGM to a specific component model is identified by a unique and legal combination of transformation features.

Figure 5.24 shows our transformation feature model. It shows four main transformation features: *Descriptor*, *Service Description*, *Interface Description* and *Required Interface*.

Descriptor. This feature is optional because not all of component models define a component descriptor. In the case where a component model includes a component descriptor, one of the following two transformations needs to be performed. If the descriptor is written in a specific-domain language, this transformation allows to derive (extract) this one from the object oriented source code. Otherwise, the descriptor object oriented code will be derived by means of some refactoring of the object oriented source code.

Service Description. This feature is mandatory. A service can be described through two variants, either imperative or declarative description. In case of imperative service, some refactoring operations are needed to be able to generate service binding code. While in the case of declarative service, provided and required services description is generated in XML-like documents. Therefore, both required and provided services must be descried explicitly.

Interface Description. This feature is mandatory. The interfaces of components described using object oriented interfaces (e.g. Java interface) do not need specific transformation, where it is already done by considering the corresponding generic component model. In contrast, interfaces that are described using a specific-domain

language (e.g. IDL) need a transformation to generate a description in the corresponding specific-domain language. For example, generating an IDL description of interfaces from available object-oriented oriented source code elements.

Required Interface. This feature is optional. Therefore, a transformation must be operated for component models that define an explicit one. The corresponding transformation extracts required interfaces that are embedded into the source code and represent it as explicit interfaces based on the targeted component model.

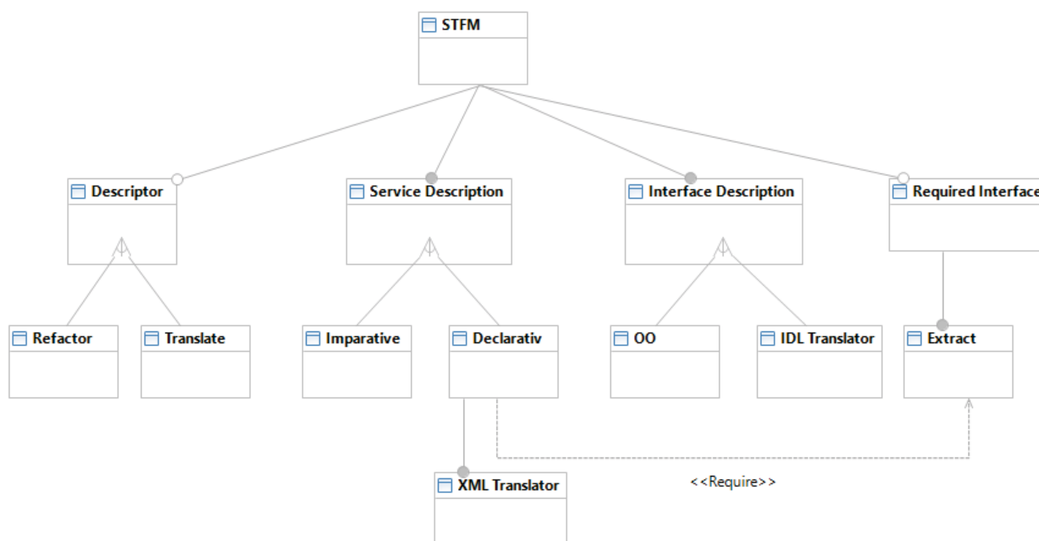


Figure 5.24: Feature model for specific transformation.

5.5 Implementation and Tools

We used many tools to implement our MDT process (see Figure 5.25). These tools are either commercial or that we have developed for the specific purpose of our work. The description of each tool is given below:

Software Architecture Recovery Many tools have been proposed to analyze and recover the software architecture from legacy object-oriented software like MoDisco [Bruneliere 2010]. However, we developed a specific tool named ROMANTIC [Kebir 2012] as an Eclipse plugin for this purpose (See sec 2.4.3). We have utilized this tool to recover architecture from legacy-object oriented code.

From source code to FAMIX model A FAMIX model can be exported in many format like XML and MSE. MSE is the generic format for FAMIX, a format similar to XML, built as part of the Fame project. Many tools have been proposed to export source code information in XML or in MSE. These tools are: VerveineJ [too 2012],

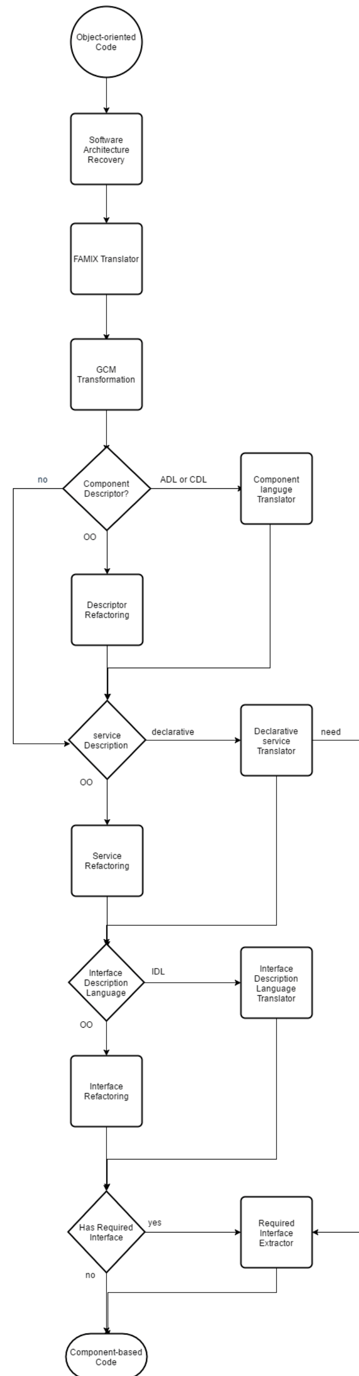


Figure 5.25: Migration process.

inFamix [too 2012] and JFamix [jfa 2004]. VerveineJ is an open-source MSE exporter built using Java and based on JDT. VerveineJ allows extracting information from Java source code. inFamix is a parser for Java and C/C++ that is based on the scalable inFusion platform. The parser is available as a free command line tool and it exports in MSE. jFamix is an Eclipse plugin which exports the meta information of a Java Project to the FAMIX meta model as an XML format.

Transformations from an OOGM to a CBGM For the implementation of transformations from OOGM to CBGM, we describe both OOGMs and CBGMs in the Ecore format. Ecore is the core metamodel at the heart of the Eclipse Modeling Framework (EMF). It allows us to take advantage of the entire EMF ecosystem and tooling. EMF is an Eclipse-based modeling framework and code generation facility for building tools.

We used QVT (Query/View/Transformation) component on EMF to transform an OOGM to a CBGM by defining QVT operations and mappings. QVT is a standard set of languages for model transformation defined by the Object Management Group. The QVT standard integrates the OCL 2.0 standard and also extends it with imperative features.

Listing 5.1 shows the importing of the two Ecore [5] metamodel files (ooar.ecore and gcm.ecore) and the main mapping functions. These functions allow to navigate among the entities of the source model that need to be mapped into a target one.

Listing 5.1: Main QVT definitions and mapping functions in our migration

```
/*
 * Two modeltypes are declared: object-oriented architecture recovery metamodel (ooar.ecore)
     and generic component metamodel (gcm.ecore).
 * The http URIs correspond to those used to register the Ecore models in the environment.
 */
modeltype OOAR uses 'http://www.eclipse.org/qvt/1.0.0/Operational/Famix/ooar';
modeltype GCM uses 'http://www.eclipse.org/qvt/1.0.0/Operational/Famix/gcm';

transformation OO_To_CB(in oo : OOAR, out GCM);

main() {
  oo.rootObjects()[OOAR::Object]->map OOModel2GCMModel();
}
...

mapping OOAR::Cluster::cluster2component() : GCM::Component {
  ...
}

mapping OOAR::InheritanceDefinition::abstractInheritance2proxy() : GCM::Component {
  ...
}

mapping OOAR::InheritanceDefinition::inheritance2delegation() : GCM::Component {
  ...
}
```

[5] http://www.eclipse.org/modeling/emf/

```
mapping OOAR::Exception::exception2ifElse() : GCM::Component {
  ...
}

mapping OOAR::Invocation::instantiation2factory() : GCM::Class {
  ...
}

mapping OOAR::Reference::reference2interface() : GCM::Class {
  ...
}

}
```

Descriptor Refactoring The refactoring is limited to adding some methods that must be implemented caused by implementing a specific interfaces of a generates component model. We use the Eclipse refactoring tools to add these methods and implement them.

Declarative service Generator We developed an Eclipse plugin that generate an XML description of provided and required services starting from source code information. Information included in this XML file are, for example, the interface type and the name of the class implementing a provided interface.

Interface Description Generator IDLJ [idl] is a compiler that reads an Object Management Group (OMG) Interface Definition Language (IDL) file and maps it, to a Java interface. Java files are generated from the IDL file according to the mapping specified in the OMG document. Table 5.1 depicts the mapping between object-oriented language and IDL. In addition, CCM tools [Teiniker 2004] can be used to generate C++ source code from IDL files.

Table 5.1: Mapping between object-oriented language and IDL

Java Type	IDL Type
package	module
boolean	boolean
char	char, wchar
byte	octet
String	string, wstring
short	short, unsigned short
int	long, unsigned long
long	long long, unsigned long long
float	float
double	double
BigDecimal	fixed
class	enum, struct, union
array	sequence, array
method	operation
accessor method	readonly attribute
accessor and modifer methods	readwrite attribute

5.6 Conclusion

In this chapter, we proposed a model-driven approach to automatically transform object-oriented applications to component-based ones.

Firstly, we defined two generic metamodels for object-oriented and component-based applications, respectively. We extended FAMIX metamodel to be able to present these metamodels. Secondly, we defined transformation rules to be able to transform object-oriented models to a generic component one. The transformation is implemented based on QVT. Thirdly, transformation rules from a generic component model to a specific one were defined based on the analysis of the features of eight component models: OSGi, SOFA, CCM, COM, OpenCOM, Fractal, JavaBeans and EJB. Finally, we define a feature model representing commonalities and variabilities of transformations from a generic component model to a specific component model. The aim of this feature model is to automate the generation of the chain of transformation following the features of each target component model.

Chapter 6

Conclusion and Future Work

Contents

6.0.1 Summary of Contributions 118
6.0.2 Future Directions 119
6.0.3 Publications 121

The ultimate goal of this dissertation is to support systematic software evolution. Towards this goal, we propose to migrate object-oriented legacy software into component-based ones. To do so, we address the following research problems:

- **Transforming source codes of object-oriented applications into component-based source codes.** Based on an available description of a component-based architecture of an object-oriented source code, our goal was to refactor (i.e. transformation without changing functionalities) this source code into another written in a component-based language. This transformation is achieved by proposing adequate solutions to the following sub-problems:
  - **Transforming object-oriented dependencies into interface-based ones.** All object-oriented dependencies like instantiation, direct type dependency and method invocation, inheritance and exception handling between object-oriented elements embedded in the implementations of the generated components need to be transformed to interface-based dependencies.
  - **Identifying Component instance.** At run-time an object-oriented application is, principally, a set of object instances of classes. Reciprocally, at run time, a component-based application is, principally, a set of components/component-instances of some component-types. To transform object-oriented applications to component-based ones, we need to define mapping between the corresponding objects and component-instances, respectively.
- **Model driven transformation.** To benefit from all advantages of model-driven engineering/transformation, operations to transform an object-oriented source code into component-based ones need to be model-based. This means that they need to be implemented as transformation rules.

6.0.1 Summary of Contributions

The main contributions of this dissertation are:

1. We proposed an approach that aims to automatically transform object-oriented applications to component-based ones. We focused on the transformation of source code in order to produce decoupled components that are compliant with a recovered component-architecture description. We proposed a solution for dealing with instantiation, type dependency and method invocation, inheritance dependency, and exception handling. We proposed to base source-code transformations on well-known design patterns like Factory, delegation, Proxy and adapter design patterns. Therefore, the transformed code is easy to understand for future maintenance operations. We demonstrated the validity of the proposed transformations on a set of applications that we automatically transformed from Java source codes into OSGi ones.

2. We proposed solutions to materialize component instances based on instances of classes. We refactored clusters of classes (recovered component) to behave as a single unit of behavior to enable component instantiation. Our approach guarantees component-based principles by resolving component encapsulation and component composition using component instances. Moreover, both principles applied to refactor the source code of a recovered component to be instantiable, where provided services are consumed by the component instances through its interfaces (component binding). We have shown that the source code resulting from our approach can be easily projected onto object-based component models. We illustrated the mapping on two well-known component models, OSGi and SOFA. The illustration results show that our approach facilitates the transformation process from OO applications into CB ones. Moreover, it effectively reduces the gap between recovered component architectures and its implementation source code.

3. We proposed a model-driven approach to automatically transform object-oriented applications to component-based ones.

   Firstly, we defined two generic metamodels for object-oriented and component-based applications, respectively. We extended Famix metamodel to be able to present these metamodels. Secondly, we defined transformation rules to be able to transform object-oriented models to a generic component one. Transformation is implemented based on QVT. Thirdly, transformation rules from a generic component model to a specific one were defined based on the analysis of the features of eight component models: OSGi, SOFA, CCM, COM, OpenCOM, Fractal, JavaBeans and EJB. Finally, we define a feature model representing commonalities and variabilities of transformations from a generic component model to a specific component model. The aim of this feature model is to automate the generation of the chain of transformation following the features of each target component model.

6.0.2 Future Directions

Based on the research work presented in this dissertation, many future directions are identified. These include:

1. **Migration from specific object-oriented models to specific component-based ones.** In this dissertation we have addressed problems related to transformation of generic OO source codes to specific component-based ones (e.g. OSGi, SOFA, MCC, etc.). By generic OO source code, we mean source codes conform to the common specification of object-oriented (i.e. Java-like). We have not addressed transformations of source code elements related to specificities (variabilities) of some OO programming languages. For example, we have not addressed transformation related to the multiple inheritance dependencies related to certain programing languages like C++. Therefore, we plan to extend our approach to address the transformation of OO source code elements related to specificities of certain OO languages.

2. **Refactoring OO source codes for better decoupling.** In this dissertation we have addressed problems related to transformation of OO source codes to component-based ones. The transformations can be considered as a phase in a migration process from OO applications to component-based ones. The migration can be motivated by all the advantages of component concept compared to object one, mainly the explicit description of dependencies of components through provided and required interfaces. This property allows components to benefit from advantages of decoupling. Decoupling favors reusability, maintainability, understandability, etc. Another interesting property of component-based applications compared to OO ones is the late binding of components. A component-based application is defined via a concrete architecture that describes the involved components and their bindings/connections. This property allows other advantages like the capacity of reconfiguring the architecture. We claim that object-oriented applications can benefit of (part of) these properties/advantages without migrating to components. The idea is to consider a class as a type of component and thus consider its objects as components. Thus, transformation related to dependencies proposed in this dissertation can be adapted to be applied to decouple classes. Design patterns like dependency injection can be applied to play a similar role as required interfaces of components. Therefore, we plan to adapt solutions proposed in this dissertation to explore this research direction.

3. **Migration from Object-oriented application into service-oriented architecture.** We plan to adapt the proposed approach to tackle problems related to migration of object-oriented applications into service-oriented architecture (SOA). Two types of service-based architecture can be considered: Service Component Architecture (SCA) and web-service based architecture. Some specific issues related to this goal are to be addressed: recovering the

dynamic behavior of the application to be able to define an orchestrator with an explicit behavior, handling the stateless property of web services proposed in some service-based models, refactoring object-oriented source codes to be adapted to the implementations of services as proposed in some service-based languages (e.g. single class implementation), etc..

6.0.3 Publications

This PhD thesis started in December 2013. During this period, we have published the following research papers:

- Mining Software Components from Object-Oriented APIs - Anas Shatnawi, Abdelhak Seriai, Houari A. Sahraou1i, **Zakarea AlShara**. 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. Lecture Notes in Computer Science 8919, Springer 2014, ISBN 978-3-319- 14129-9: 330-347.

- Reverse Engineering Reusable Software Components from Object-Oriented APIs. Anas Shatnawi, Abdelhak Seriai, Houari A. Sahraou1i, **Zakarea Al-Shara**. Journal of Systems and Software (JSS).

- Migrating large object-oriented Applications into component-based ones: instantiation and inheritance transformation. **Zakarea AlShara**, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde Lilia Bouziane, Christophe Dony and Anas Shatnawi. (2015, October). In Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE'15, (pp. 55-64). ACM.ââ

- Materializing Architecture Recovered from OO Source Code in Component-Based Languages. **Zakarea AlShara**, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde Lilia Bouziane, Christophe Dony and Anas Shatnawi. Software architecture: 10th european conference, ecsa 2016, Istanbul, Turky, September 5- 9, 2016. Springer International Publishing, 2016.

Bibliography

[Abadi 1996] Martín Abadi, Luca Cardelli and Ramesh Viswanathan. *An interpretation of objects and object types.* In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 396–409. ACM, 1996. (Cited on page 45.)

[Abadi 2012] Martin Abadi and Luca Cardelli. A theory of objects. Springer Science & Business Media, 2012. (Cited on page 45.)

[Adjoyan 2014] Seza Adjoyan, Abdelhak-Djamel Seriai and Anas Shatnawi. *Service Identification Based on Quality Metrics - Object-Oriented Legacy System Migration Towards SOA.* In The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013., pages 1–6, 2014. (Cited on page 36.)

[Ahmad 2014] Aakash Ahmad and Muhammad Ali Babar. *A Framework for Architecture-driven Migration of Legacy Systems to Cloud-enabled Software.* In Proceedings of the WICSA 2014 Companion Volume, WICSA '14 Companion, pages 7:1–7:8, New York, NY, USA, 2014. ACM. (Cited on pages 17, 19, 23, 28 and 30.)

[Akers 2004] Robert L. Akers, Ira D. Baxter and Michael Mehlich. *Program Transformations for Re-engineering C++ Components [OOPSLA/GPCE].* In Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '04, pages 25–26, New York, NY, USA, 2004. ACM. (Cited on pages 16, 17, 19, 20, 23, 28 and 30.)

[Allier 2010] Simon Allier, HouariA. Sahraoui, Salah Sadou and StÃ©phane Vaucher. *Restructuring Object-Oriented Applications into Component-Oriented Applications by Using Consistency with Execution Traces.* In Lars Grunske, Ralf Reussner and Frantisek Plasil, editeurs, Component-Based Software Engineering, volume 6092 of *Lecture Notes in Computer Science*, pages 216–231. Springer Berlin Heidelberg, 2010. (Cited on page 7.)

[Allier 2011] S. Allier, S. Sadou, H. Sahraoui and R. Fleurquin. *From Object-Oriented Applications to Component-Oriented Applications via Component-Oriented Architecture.* In Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on, pages 214–223, June 2011. (Cited on pages 16, 17, 19, 23, 28 and 30.)

[Alshara 2015] Zakarea Alshara, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde Lilia Bouziane, Christophe Dony and Anas Shatnawi. *Migrating large object-oriented Applications into component-based ones: instantiation and*

inheritance transformation. In Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pages 55–64. ACM, 2015. (Cited on pages 72 and 83.)

[Axelsen 2012] Eyvind W. Axelsen and Stein Krogdahl. *Package Templates: A Definition by Semantics-preserving Source-to-source Transformations to Efficient Java Code*. SIGPLAN Not., vol. 48, no. 3, pages 50–59, September 2012. (Cited on pages 17, 19, 23, 28 and 30.)

[Bass 2012] Len Bass, Paul Clements and Rick Kazman. Software architecture in practice. Addison-Wesley Professional, 3rd édition, 2012. (Cited on page 31.)

[Baster 2001] Greg Baster, Prabhudev Konana and Judy E Scott. *Business components: a case study of bankers trust Australia limited*. Communications of the ACM, vol. 44, no. 5, pages 92–98, 2001. (Cited on page 31.)

[Becker 2007] Steffen Becker, Heiko Koziolek and Ralf Reussner. *Model-based performance prediction with the palladio component model*. In Proceedings of the 6th international workshop on Software and performance, pages 54–65. ACM, 2007. (Cited on page 73.)

[Beisiegel 2005] Michael Beisiegel, Henning Blohm, Dave Booz, J Dubray, Adrian Colyer, Mike Edwards, Don Ferguson, Bill Flood, Mike Greenberg, Dan Kearns*et al.* *Service Component Architecture: Building Systems Using a Service Oriented Architecture*. Whitepaper [online], vol. 1, page 31, 2005. (Cited on page 6.)

[Beisiegel 2007] Michael Beisiegel, Henning Blohm, Dave Booz, Mike Edwards, Oisin Hurley, S Ielceanu, A Miller, A Karmarkar, A Malhotra, J Marino*et al.* *SCA service component architecture, assembly model specification*. Open Service Oriented Architecture www. osoa. org/download/attachments/35/SCA_Assembly_Model_V100. pdf, 2007. (Cited on page 6.)

[Bennett 1982] PA Bennett. *Fault Tolerance: Principles and Practice*. The Computer Journal, vol. 25, no. 3, pages 400–d, 1982. (Cited on pages xi and 54.)

[Bertolino 2005a] Antonia Bertolino, Antonio Bucchiarone, Stefania Gnesi and Henry Muccini. *An architecture-centric approach for producing quality systems*. In Quality of Software Architectures and Software Quality, pages 21–37. Springer, 2005. (Cited on page 5.)

[Bertolino 2005b] Antonia Bertolino, Antonio Bucchiarone, Stefania Gnesi and Henry Muccini. *An architecture-centric approach for producing quality systems*. In Quality of Software Architectures and Software Quality, pages 21–37. Springer, 2005. (Cited on page 5.)

[Bieman 1995] James M Bieman and Byung-Kyoo Kang. *Cohesion and reuse in an object-oriented system.* In ACM SIGSOFT Software Engineering Notes, volume 20, pages 259–262. ACM, 1995. (Cited on page 36.)

[Biggerstaff 1989] Ted J. Biggerstaff. *Design Recovery for Maintenance and Reuse.* Computer, vol. 22, no. 7, pages 36–49, July 1989. (Cited on page 12.)

[Binkley 2006] D. Binkley, M. Ceccato, M. Harman, F. Ricca and P. Tonella. *Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects.* IEEE Transactions on Software Engineering, vol. 32, no. 9, pages 698–717, Sept 2006. (Cited on pages 17, 19, 23, 28 and 30.)

[Birkmeier 2009] Dominik Birkmeier and Sven Overhage. *On Component Identification Approaches â Classification, State of the Art, and Comparison.* In GraceA. Lewis, Iman Poernomo and Christine Hofmeister, editeurs, Component-Based Software Engineering, volume 5582 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2009. (Cited on pages 7, 31 and 33.)

[Bisbal 1999] Jesús Bisbal, Deirdre Lawless, Bing Wu and Jane Grimson. *Legacy information systems: Issues and directions.* IEEE software, vol. 16, no. 5, page 103, 1999. (Cited on pages 2, 3 and 10.)

[Boshernitsan 2006] Marat Boshernitsan and Susan L. Graham. *Interactive Transformation of Java Programs in Eclipse.* In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 791–794, New York, NY, USA, 2006. ACM. (Cited on pages 17, 19, 23, 28 and 30.)

[Box 1997] D. Box. *Essential COM. Object Technology Series*, 1997. (Cited on page 73.)

[Brodie] ML Brodie and M Stonebraker. *Migrating legacy systems: gateways, interfaces & the incremental approach. 1995.* (Cited on page 4.)

[Brooks Jr 1995] Frederick P Brooks Jr. The mythical man-month: Essays on software engineering, anniversary edition, 2/e. Pearson Education India, 1995. (Cited on page 4.)

[Bruneliere 2010] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault and Frédéric Madiot. *MoDisco: A Generic and Extensible Framework for* ***Model Driven*** *Reverse Engineering.* In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM. (Cited on pages 17, 19, 20, 23, 28, 30 and 112.)

[Buhr 2000] Peter A Buhr and WY Russell Mok. *Advanced exception handling mechanisms.* Software Engineering, IEEE Transactions on, vol. 26, no. 9, pages 820–836, 2000. (Cited on page 45.)

[Bures 2006] T. Bures. *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*. In Software Engineering Research, Management and Applications., 2006. (Cited on pages 71, 72 and 73.)

[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. Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006). (Cited on pages 17, 19, 20, 23, 28 and 30.)

[CanforaHarman 2007] Gerardo CanforaHarman and Massimiliano Di Penta. *New frontiers of reverse engineering*. In 2007 Future of Software Engineering, pages 326–341. IEEE Computer Society, 2007. (Cited on pages 11, 20 and 21.)

[Chardigny 2008a] S. Chardigny, A. Seriai, M. Oussalah and D. Tamzalit. *Extraction of Component-Based Architecture from Object-Oriented Systems*. In Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on, pages 285–288, Feb 2008. (Cited on pages 7, 33 and 59.)

[Chardigny 2008b] Sylvain Chardigny, Abdelhak Seriai, Mourad Oussalah and Dalila Tamzalit. *Search-based extraction of component-based architecture from object-oriented systems*. In Software Architecture, pages 322–325. Springer, 2008. (Cited on page 36.)

[Chardigny 2008c] Sylvain Chardigny, Abdelhak Seriai, Dalila Tamzalit and Mourad Oussalah. *Quality-driven extraction of a component-based architecture from an object-oriented system*. In 12th European Conference on Software Maintenance and Reengineering (CSMR), pages 269–273. IEEE, 2008. (Cited on pages 33, 34 and 36.)

[Charnes 1978] Abraham Charnes, William W Cooper and Edwardo Rhodes. *Measuring the efficiency of decision making units*. European journal of operational research, vol. 2, no. 6, pages 429–444, 1978. (Cited on page 16.)

[Chen 2002] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox and Eric Brewer. *Pinpoint: Problem determination in large, dynamic internet services*. In Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, pages 595–604. IEEE, 2002. (Cited on page 21.)

[Chikofsky 1990] Elliot J Chikofsky, James H Cross*et al.* *Reverse engineering and design recovery: A taxonomy*. Software, IEEE, vol. 7, no. 1, pages 13–17, 1990. (Cited on pages 11 and 16.)

[Cimitile 1999] Aniello Cimitile, Andrea De Lucia, Giuseppe Antonio Di Lucca and Anna Rita Fasolino. *Identifying objects in legacy systems using design metrics*. Journal of Systems and Software, vol. 44, no. 3, pages 199–211, 1999. (Cited on page 12.)

[Clark 2008] Tony Clark, Paul Sammut and James Willans. *Applied metamodelling: a foundation for language driven development.* 2008. (Cited on page 18.)

[Clark 2015] Tony Clark, Paul Sammut and James Willans. *Applied Metamodelling: A Foundation for Language Driven Development.* arXiv preprint arXiv:1505.00149, 2015. (Cited on page 18.)

[Clarke 2001] Michael Clarke, Gordon S Blair, Geoff Coulson and Nikos Parlavantzas. *An efficient component model for the construction of adaptive middleware.* In IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, pages 160–178. Springer, 2001. (Cited on page 73.)

[Clavreul 2010] Mickael Clavreul, Olivier Barais and Jean-Marc Jézéquel. *Integrating Legacy Systems with MDE.* In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, pages 69–78, New York, NY, USA, 2010. ACM. (Cited on pages 17, 19, 23, 28 and 30.)

[Clements 2002] Paul Clements and Linda Northrop. *Software product lines: practices and patterns.* 2002. (Cited on page 15.)

[Constantinou 2015] Eleni Constantinou, Athanasios Naskos, George Kakarontzas and Ioannis Stamelos. *Extracting reusable components: A semi-automated approach for complex structures.* Information Processing Letters, vol. 115, no. 3, pages 414 – 417, 2015. (Cited on page 5.)

[Coupaye 2006] Thierry Coupaye and Jean-Bernard Stefani. *Fractal component-based software engineering.* In European Conference on Object-Oriented Programming, pages 117–129. Springer, 2006. (Cited on page 105.)

[Crnkovic 2011a] Ivica Crnkovic, Severine Sentilles, Aneta Vulgarakis and Michel RV Chaudron. *A classification framework for software component models.* IEEE Transactions on Software Engineering, vol. 37, no. 5, pages 593–615, 2011. (Cited on pages xiii and 73.)

[Crnkovic 2011b] Ivica Crnkovic, Severine Sentilles, Aneta Vulgarakis and Michel RV Chaudron. *A classification framework for software component models.* IEEE Transactions on Software Engineering, vol. 37, no. 5, pages 593–615, 2011. (Cited on page 33.)

[De Lucia 2008] Andrea De Lucia, Rita Francese, Giuseppe Scanniello and Genoveffa Tortora. *Developing legacy system migration methods and tools for technology transfer.* Software: practice & experience, vol. 38, no. 13, page 1333, 2008. (Cited on pages 17, 19, 23, 28 and 30.)

[Demeyer 1999] Serge Demeyer, Stéphane Ducasse and Sander Tichelaar. *Why FAMIX and not UML*. In Proceedings of UMLâ99, volume 1723, 1999. (Cited on page 91.)

[Demeyer 2001] Serge Demeyer, Sander Tichelaar and Stéphane Ducasse. *FAMIX 2.1âthe FAMOOS information exchange model*, 2001. (Cited on page 90.)

[Ding 2011] Zuohua Ding, Mingyue Jiang and Jens Palsberg. *From Textual Use Cases to Service Component Models*. In Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems, PESOS '11, pages 8–14, New York, NY, USA, 2011. ACM. (Cited on pages 17, 19, 20, 23, 28 and 30.)

[Dony 1990] Christophe Dony. *Exception handling and object-oriented programming: towards a synthesis*. ACM Sigplan Notices, vol. 25, no. 10, pages 322–330, 1990. (Cited on page 45.)

[Ducasse 2000] Stéphane Ducasse, Michele Lanza and Sander Tichelaar. *Moose: an extensible language-independent environment for reengineering object-oriented systems*. In Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000), volume 4, 2000. (Cited on page 91.)

[Ducasse 2009a] S. Ducasse and D. Pollet. *Software Architecture Reconstruction: A Process-Oriented Taxonomy*. Software Engineering, IEEE Transactions on, vol. 35, no. 4, pages 573–591, July 2009. (Cited on pages 7, 32 and 33.)

[Ducasse 2009b] Stéphane Ducasse and Damien Pollet. *Software architecture reconstruction: A process-oriented taxonomy*. Software Engineering, IEEE Transactions on, vol. 35, no. 4, pages 573–591, 2009. (Cited on page 29.)

[Ducasse 2011a] Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval and Tudor Girba. *MSE and FAMIX 3.0: an interexchange format and source code model family*. 2011. (Cited on page 18.)

[Ducasse 2011b] Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval and Tudor Girba. *MSE and FAMIX 3.0: an interexchange format and source code model family*. 2011. (Cited on page 91.)

[Dunsmore 2000] Alastair Dunsmore, Marc Roper and Murray Wood. *Object-oriented inspection in the face of delocalisation*. In Proceedings of the 22nd international conference on Software engineering, pages 467–476. ACM, 2000. (Cited on page 32.)

[E.E. Group 2006] Oracle E.E. Group. *JSR 220: Enterprise JavaBeansTM,Version 3.0 EJB Core Contracts and Requirements Version 3.0, Final Release*, May 2006. (Cited on page 73.)

[Einarsson 2012] Hafsteinn Þór Einarsson and Helmut Neukirchen. *An Approach and Tool for Synchronous Refactoring of UML Diagrams and Models Using Model-to-model Transformations*. In Proceedings of the Fifth Workshop on Refactoring Tools, WRT '12, pages 16–23, New York, NY, USA, 2012. ACM. (Cited on pages 17, 19, 23, 28 and 30.)

[Erlikh 2000] Len Erlikh. *Leveraging legacy system dollars for e-business*. IT professional, vol. 2, no. 3, pages 17–23, 2000. (Cited on page 5.)

[Ernst 2003] Michael D Ernst. *Static and dynamic analysis: Synergy and duality*. In WODA 2003: ICSE Workshop on Dynamic Analysis, pages 24–27. Citeseer, 2003. (Cited on pages 21 and 39.)

[Escoffier 2005] C Escoffier and D Donsez. *FractNet: A Fractal implementation for .NET*. Session poster de la 2ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2005), 2005. (Cited on page 105.)

[Eysholdt 2010] Moritz Eysholdt and Johannes Rupprecht. *Migrating a Large Modeling Environment from XML/UML to Xtext/GMF*. In Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA '10, pages 97–104, New York, NY, USA, 2010. ACM. (Cited on pages 17, 19, 20, 23, 28 and 30.)

[Favre 2004] Jean-Marie Favre. *Towards a basic theory to model model driven engineering*. In 3rd Workshop in Software Model Engineering, WiSME, pages 262–271. Citeseer, 2004. (Cited on page 86.)

[Favre 2005] Jean-Marie Favre. *Foundations of model (driven)(reverse) engineering: Models–episode I: stories of the fidus papyrus and of the solarus*. In Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2005. (Cited on page 26.)

[Fenton 2014] Norman Fenton and James Bieman. Software metrics: a rigorous and practical approach. CRC Press, 2014. (Cited on pages 15 and 16.)

[Fleurey 2007] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas and Jean-Marc Jézéquel. *Model-driven engineering for software migration in a large industrial context*. In International Conference on Model Driven Engineering Languages and Systems, pages 482–497. Springer, 2007. (Cited on page 86.)

[Forward 2002] Andrew Forward and Timothy C. Lethbridge. *The Relevance of Software Documentation, Tools and Technologies: A Survey*. In Proceedings

of the 2002 ACM Symposium on Document Engineering, DocEng '02, pages 26–33, New York, NY, USA, 2002. ACM. (Cited on page 18.)

[Frakes 1996] William Frakes and Carol Terry. *Software reuse: metrics and models*. ACM Computing Surveys (CSUR), vol. 28, no. 2, pages 415–435, 1996. (Cited on page 15.)

[Frakes 2005] William B. Frakes and Kyo Kang. *Software Reuse Research: Status and Future*. IEEE Trans. Softw. Eng., vol. 31, no. 7, pages 529–536, July 2005. (Cited on pages 13 and 15.)

[Freeman 2004] Eric Freeman, Elisabeth Robson, Bert Bates and Kathy Sierra. Head first design patterns. " O'Reilly Media, Inc.", 2004. (Cited on page 24.)

[Fuhr 2011] Andreas Fuhr, Tassilo Horn, Volker Riediger and Andreas Winter. *Model-driven software migration into service-oriented architectures*. Computer Science - Research and Development, vol. 28, no. 1, pages 65–84, 2011. (Cited on pages 17, 19, 23, 28 and 30.)

[Garlan 2000] David Garlan. *Software architecture: a roadmap*. In Proceedings of the Conference on the Future of Software Engineering, pages 91–101. ACM, 2000. (Cited on pages 31 and 32.)

[Gligoric 2014] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya and Benjamin Livshits. *Automated Migration of Build Scripts Using Dynamic Analysis and Search-based Refactoring*. SIGPLAN Not., vol. 49, no. 10, pages 599–616, October 2014. (Cited on pages 17, 19, 23, 28 and 30.)

[Gray 2006] Jeff Gray, Yuehua Lin and Jing Zhang. *Automating change evolution in model-driven engineering*. Computer, vol. 39, no. 2, pages 51–58, 2006. (Cited on page 86.)

[Grundy 2000] John Grundy and John Hosking. *High-level static and dynamic visualisation of software architectures*. In Visual Languages, 2000. Proceedings. 2000 IEEE International Symposium on, pages 5–12. IEEE, 2000. (Cited on page 32.)

[Haas 2004] Hugo Haas and Allen Brown. *Web Services Glossary*. W3C note, W3C, February 2004. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/. (Cited on page 22.)

[Hamilton 1997] Graham Hamilton. *JavaBeans*. API Specification, Sun Microsystems, 1997. (Cited on page 107.)

[Hamza 2013] S. Hamza, S. Sadou and R. Fleurquin. *Measuring Qualities for OSGi Component-Based Applications*. In Quality Software (QSIC), 2013 13th International Conference on, pages 25–34, July 2013. (Cited on page 58.)

[Harman 2010] Mark Harman. *Why Source Code Analysis and Manipulation Will Always Be Important.* In Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM '10, pages 7–19, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on page 16.)

[Heineman 2001a] George T Heineman and William T Councill. *Component-based software engineering.* Putting the pieces together, addison-westley, page 5, 2001. (Cited on page 15.)

[Heineman 2001b] George T Heineman and William T Councill. *Component-based software engineering.* Putting the pieces together, addison-westley, page 5, 2001. (Cited on page 34.)

[Hunold 2008] S. Hunold, M. Korch, B. Krellner, T. Rauber, T. Reichel and G. RÃ¼nger. *Transformation of Legacy Software into Client/Server Applications through Pattern-Based Rearchitecturing.* In 2008 32nd Annual IEEE International Computer Software and Applications Conference, pages 303–310, July 2008. (Cited on pages 17, 19, 23, 28 and 30.)

[Hunold 2009] Sascha Hunold, Björn Krellner, Thomas Rauber, Thomas Reichel and Gudula Rünger. Enterprise information systems: 11th international conference, iceis 2009, milan, italy, may 6-10, 2009. proceedings, chapitre Pattern-Based Refactoring of Legacy Software Systems, pages 78–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. (Cited on pages 17, 19, 23, 28 and 30.)

[Hunt 2002] Andy Hunt and Dave Thomas. *Software archaeology.* IEEE Software, vol. 19, no. 2, pages 20–22, 2002. (Cited on page 2.)

[idl] *OMG IDL to Java Language Mapping Specification, ptc, 00-01-08.* (Cited on page 115.)

[IEE 1998] *IEEE Standard for Software Maintenance.* IEEE Std 1219-1998, pages i–, 1998. (Cited on page 11.)

[Iso 2001] ISO Iso. *IEC 9126-1: Software Engineering-Product Quality-Part 1: Quality Model.* Geneva, Switzerland: International Organization for Standardization, 2001. (Cited on page 34.)

[Jacobson 1997] Ivar Jacobson, Martin Griss and Patrik Jonsson. Software reuse: architecture, process and organization for business success. ACM Press/Addison-Wesley Publishing Co., 1997. (Cited on page 15.)

[Jacobson 2004] Ivar Jacobson and Pan-Wei Ng. Aspect-oriented software development with use cases (addison-wesley object technology series). Addison-Wesley Professional, 2004. (Cited on page 15.)

[jfa 2004] *JFamix,*, 2004. (Cited on page 114.)

[Jones 2008] Capers Jones. Applied software measurement: global analysis of productivity and quality. McGraw-Hill Education Group, 2008. (Cited on page 16.)

[Jouault 2008] Frédéric Jouault, Freddy Allilaire, Jean Bézivin and Ivan Kurtev. *ATL: A model transformation tool.* Science of computer programming, vol. 72, no. 1, pages 31–39, 2008. (Cited on pages 24 and 25.)

[Kapur 2010] Puneet Kapur, Brad Cossette and Robert J Walker. Refactoring references for library migration, volume 45. ACM, 2010. (Cited on pages 17, 19, 23, 25, 28 and 30.)

[Karim 2014] Rezwana Karim, Mohan Dhawan and Vinod Ganapathy. Ecoop 2014 – object-oriented programming: 28th european conference, uppsala, sweden, july 28 – august 1, 2014. proceedings, chapitre Retargetting Legacy Browser Extensions to Modern Extension Frameworks, pages 463–488. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. (Cited on pages 17, 19, 23, 26, 28 and 30.)

[Kazman 1998] R. Kazman, S. G. Woods and S. J. Carriere. *Requirements for integrating software architecture and reengineering models: CORUM II.* In Reverse Engineering, 1998. Proceedings. Fifth Working Conference on, pages 154–163, Oct 1998. (Cited on pages 10 and 11.)

[Kebir 2012] Selim Kebir, A-D Seriai, Sylvain Chardigny and Allaoua Chaoui. *Quality-centric approach for software component identification from object-oriented code.* In 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), pages 181–190. IEEE, 2012. (Cited on pages 7, 12, 22, 33, 34, 36, 59, 66 and 112.)

[Kegel 2008a] Hannes Kegel and Friedrich Steimann. *Systematically Refactoring Inheritance to Delegation in Java.* In Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pages 431–440, New York, NY, USA, 2008. ACM. (Cited on pages 17, 19, 23, 28, 30 and 57.)

[Kegel 2008b] Hannes Kegel and Friedrich Steimann. *Systematically Refactoring Inheritance to Delegation in Java.* In Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pages 431–440, New York, NY, USA, 2008. ACM. (Cited on pages 49 and 57.)

[Kjolstad 2011] Fredrik Kjolstad, Danny Dig, Gabriel Acevedo and Marc Snir. *Transformation for Class Immutability.* In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 61–70, New York, NY, USA, 2011. ACM. (Cited on pages 17, 19, 23, 28 and 30.)

[Kleppe 2003] Anneke G Kleppe, Jos B Warmer and Wim Bast. Mda explained: the model driven architecture: practice and promise. Addison-Wesley Professional, 2003. (Cited on page 12.)

[Kobel 2005] Markus Kobel, Oscar Nierstrasz, Horst Bunke, Tudor Gırba and Michele Lanza. *Parsing by example*. Institut fur Informatik und angewandte Mathematik, 2005. (Cited on page 91.)

[Kühne 2006] Thomas Kühne. *Matters of (meta-) modeling*. Software & Systems Modeling, vol. 5, no. 4, pages 369–385, 2006. (Cited on page 86.)

[Lau 2005] Kung-Kiu Lau and Zheng Wang. *A taxonomy of software component models*. In Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on, pages 88–95, Aug 2005. (Cited on pages 43 and 44.)

[Lau 2007] Kung-Kiu Lau and Zheng Wang. *Software Component Models*. Software Engineering, IEEE Transactions on, vol. 33, no. 10, pages 709–724, Oct 2007. (Cited on page 5.)

[Leach 1997] Ronald J Leach. Software reuse: methods, models, and costs. McGraw-Hill New York, 1997. (Cited on pages 13 and 15.)

[Lehman 1980] Meir M Lehman. *Programs, life cycles, and laws of software evolution*. Proceedings of the IEEE, vol. 68, no. 9, pages 1060–1076, 1980. (Cited on page 16.)

[Lehman 1985] Manny M Lehman and Laszlo A Belady. Program evolution: processes of software change. Academic Press Professional, Inc., 1985. (Cited on pages 4 and 5.)

[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. IEEE, 1997. (Cited on page 16.)

[Li 1993] Wei Li and Sallie Henry. *Object-oriented metrics that predict maintainability*. Journal of systems and software, vol. 23, no. 2, pages 111–122, 1993. (Cited on page 16.)

[Ludewig 2003] Jochen Ludewig. *Models in software engineering–an introduction*. Software and Systems Modeling, vol. 2, no. 1, pages 5–14, 2003. (Cited on page 86.)

[Lüer 2002] Chris Lüer and André Van Der Hoek. Composition environments for deployable software components. Citeseer, 2002. (Cited on pages 31 and 34.)

[M. D. McIlroy 1968] P. Naur M. D. McIlroy J. Buxton and B. Randell. *Mass-produced software components*. In null, pages 88â–98. 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany, 1968. (Cited on page 15.)

[Maqbool 2007] Onaiza Maqbool and Haroon A Babri. *Hierarchical clustering for software architecture recovery*. Software Engineering, IEEE Transactions on, vol. 33, no. 11, pages 759–780, 2007. (Cited on page 29.)

[Martin 2011] Robert C. Martin. Agile software development: Principles, patterns, and practices: International edition. Pearson, London, UK, 2011. (Cited on page 58.)

[Matos 2011] Carlos Matos and Reiko Heckel. Rigorous software engineering for service-oriented systems: Results of the sensoria project on software engineering for service-oriented computing, chapitre Legacy Transformations for Extracting Service Components, pages 604–621. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. (Cited on pages 17, 19, 23, 28 and 30.)

[Medvidovic 2003] Egyed Medvidovic. *Gruenbacher. Stemming architectural erosion by architectural discovery and recovery*. In International Workshop from Software Requirements to Architectures (STRAW), 2003. (Cited on page 32.)

[Medvidovic 2006] Nenad Medvidovic and Vladimir Jakobac. *Using software evolution to focus architectural recovery*. Automated Software Engineering, vol. 13, no. 2, pages 225–256, 2006. (Cited on page 33.)

[Mellor 2003] Stephen J Mellor, Tony Clark and Takao Futagami. *Model-driven development: guest editors' introduction*. IEEE software, vol. 20, no. 5, pages 14–18, 2003. (Cited on page 86.)

[Mellor 2004] Stephen J Mellor. Mda distilled: principles of model-driven architecture. Addison-Wesley Professional, 2004. (Cited on page 18.)

[Mendonça 1996] Nabor C Mendonça and Jeff Kramer. *Requirements for an effective architecture recovery framework*. In Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops, pages 101–105. ACM, 1996. (Cited on page 33.)

[Mens 2006a] Tom Mens and Pieter Van Gorp. *A Taxonomy of Model Transformation*. Electronic Notes in Theoretical Computer Science, vol. 152, pages 125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)Graph and Model Transformation 2005. (Cited on pages 26 and 40.)

[Mens 2006b] Tom Mens and Pieter Van Gorp. *A taxonomy of model transformation*. Electronic Notes in Theoretical Computer Science, vol. 152, pages 125–142, 2006. (Cited on page 86.)

[Mens 2008] Tom Mens. Introduction and roadmap: History and challenges of software evolution. Springer, 2008. (Cited on page 16.)

[Microsystems 1997] Sun Microsystems. *Javabeans Specification*, 1997. (Cited on page 73.)

[Miller 1997] Robert Miller and Anand Tripathi. *Issues with exception handling in object-oriented systems*. In ECOOP'97âObject-Oriented Programming, pages 85–103. Springer, 1997. (Cited on page 45.)

[Mitchell 2006] Brian S Mitchell and Spiros Mancoridis. *On the automatic modularization of software systems using the bunch tool*. Software Engineering, IEEE Transactions on, vol. 32, no. 3, pages 193–208, 2006. (Cited on page 12.)

[Mohagheghi 2007] Parastoo Mohagheghi and Reidar Conradi. *Quality, productivity and economic benefits of software reuse: a review of industrial studies*. Empirical Software Engineering, vol. 12, no. 5, pages 471–516, 2007. (Cited on page 15.)

[Moody 2009] Daniel Moody. *The âphysicsâ of notations: toward a scientific basis for constructing visual notations in software engineering*. IEEE Transactions on Software Engineering, vol. 35, no. 6, pages 756–779, 2009. (Cited on page 86.)

[Moriconi 1994] Mark Moriconi and Xiaolei Qian. *Correctness and Composition of Software Architectures*. In Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT '94, pages 164–174, New York, NY, USA, 1994. ACM. (Cited on page 31.)

[Müller 2013] Bernd Müller. Reengineering: Eine einführung. Springer-Verlag, 2013. (Cited on pages 3 and 5.)

[Nguyen 2014] Anh Tuan Nguyen, Tung Thanh Nguyen and Tien N. Nguyen. *Migrating Code with Statistical Machine Translation*. In Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, pages 544–547, New York, NY, USA, 2014. ACM. (Cited on pages 17, 19, 23, 28 and 30.)

[Nummenmaa 2011] Timo Nummenmaa, Aleksi Tiensuu, Eleni Berki, Tommi Mikkonen, Jussi Kuittinen and Annakaisa Kultima. *Supporting agile development by facilitating natural user interaction with executable formal specifications*. ACM SIGSOFT Software Engineering Notes, vol. 36, no. 4, pages 1–10, 2011. (Cited on page 25.)

[O'Brien 2002] Liam O'Brien, Christoph Stoermer and Chris Verhoef. *Software architecture reconstruction: Practice needs and current approaches*. Rapport technique, DTIC Document, 2002. (Cited on page 33.)

[OMG 2011] OMG. *OMG CORBA Component Model v4.0*, 2011. (Cited on page 73.)

[Perry 1992] Dewayne E Perry and Alexander L Wolf. *Foundations for the study of software architecture*. ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4, pages 40–52, 1992. (Cited on page 31.)

[Pfenning 1988] Frank Pfenning and Conal Elliot. *Higher-order abstract syntax*. In ACM SIGPLAN Notices, volume 23, pages 199–208. ACM, 1988. (Cited on pages 24 and 25.)

[Platform 2015] Osgi Service Platform. *The OSGi Alliance, Release 6*, 2015. (Cited on pages 44, 60, 71, 73 and 79.)

[Poch 2009] Tomáš Poch and František Plášil. Component-based software engineering: 12th international symposium, cbse 2009 east stroudsburg, pa, usa, june 24-26, 2009 proceedings, chapitre Extracting Behavior Specification of Components in Legacy Applications, pages 87–103. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. (Cited on pages 16, 17, 19, 23, 28 and 30.)

[Pohl 2010] Klaus Pohl. Requirements engineering: Fundamentals, principles, and techniques. Springer Publishing Company, Incorporated, 1st édition, 2010. (Cited on page 18.)

[Porres 2003] Ivan Porres. Model refactorings as rule-based update transformations. Springer, 2003. (Cited on page 25.)

[Pressman 1986] R S Pressman. Software engineering: A practitioner's approach (2nd ed.). McGraw-Hill, Inc., New York, NY, USA, 1986. (Cited on page 39.)

[Rugaber 2004] Spencer Rugaber and Kurt Stirewalt. *Model-driven reverse engineering*. IEEE software, vol. 21, no. 4, pages 45–53, 2004. (Cited on page 86.)

[Rumbaugh 1991a] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William E. Lorensen*et al.* Object-oriented modeling and design, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991. (Cited on page 16.)

[Rumbaugh 1991b] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William E. Lorensen*et al.* Object-oriented modeling and design, volume 199. Prentice-hall Englewood Cliffs, NJ, 1991. (Cited on page 21.)

[Santos 2015] G. Santos, N. Anquetil, A. Etien, S. Ducasse and M. T. Valente. *System specific, source code transformations*. In Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, pages 221–230, Sept 2015. (Cited on pages 17, 19, 23, 28 and 30.)

[Scalise 2010] Eugenio GP Scalise, Jean-Marie Favre and Nancy Zambrano. *MODEL-DRIVEN REVERSE ENGINEERING AND PROGRAM COMPREHENSION: AN EXAMPLE/INGENIERÍA REVERSA Y COMPRENSIÓN DE PROGRAMAS DIRIGIDA POR MODELOS: UN EJEMPLO*. Ingeniare: Revista Chilena de Ingenieria, vol. 18, no. 1, page 76, 2010. (Cited on page 1.)

[Schmidt 2006] Douglas C Schmidt. *Model-driven engineering*. COMPUTER-IEEE COMPUTER SOCIETY-, vol. 39, no. 2, page 25, 2006. (Cited on page 86.)

[Seacord 2003a] Robert C Seacord, Daniel Plakosh and Grace A Lewis. Modernizing legacy systems: software technologies, engineering processes, and business practices. Addison-Wesley Professional, 2003. (Cited on pages 3 and 5.)

[Seacord 2003b] Robert C Seacord, Daniel Plakosh and Grace A Lewis. Modernizing legacy systems: software technologies, engineering processes, and business practices. Addison-Wesley Professional, 2003. (Cited on page 24.)

[Selim 2013] Gehan M. K. Selim, Shige Wang, James R. Cordy and Juergen Dingel. *Model transformations for migrating legacy deployment models in the automotive industry*. Software & Systems Modeling, vol. 14, no. 1, pages 365–381, 2013. (Cited on pages 17, 19, 23, 26, 28 and 30.)

[Sendall 2003] Shane Sendall and Wojtek Kozaczynski. *Model transformation the heart and soul of model-driven software development*. Rapport technique, 2003. (Cited on page 86.)

[Seriai 2014] Abderrahmane Seriai, Salah Sadou and Houari A. Sahraoui. Software architecture: 8th european conference, ecsa 2014, vienna, austria, august 25-29, 2014. proceedings, chapitre Enactment of Components Extracted from an Object-Oriented Application, pages 234–249. Springer International Publishing, Cham, 2014. (Cited on pages 17, 19, 23, 28 and 30.)

[Shiva 2007] Sajjan G Shiva and Lubna Abou Shala. *Software reuse: Research and practice*. In null, pages 603–609. IEEE, 2007. (Cited on pages 13 and 15.)

[Siff 1999] Michael Siff and Thomas Reps. *Identifying modules via concept analysis*. Software Engineering, IEEE Transactions on, vol. 25, no. 6, pages 749–768, 1999. (Cited on page 12.)

[Smith 2011] David J Smith. Reliability, maintainability and risk 8e: Practical methods for engineers including reliability centred maintenance and safety-related systems. Elsevier, 2011. (Cited on page 16.)

[Sneed 1984] Harry M Sneed. *Software renewal: A case study*. IEEE Software, vol. 1, no. 3, page 56, 1984. (Cited on page 1.)

[Sneed 2000] Harry M Sneed. *Encapsulation of legacy software: A technique for reusing legacy software components.* Annals of Software Engineering, vol. 9, no. 1-2, pages 293–313, 2000. (Cited on page 24.)

[Sommerville 2010] Ian Sommerville. Software engineering. Addison-Wesley Publishing Company, USA, 9th édition, 2010. (Cited on page 1.)

[Spacek 2012] Petr Spacek, Christophe Dony, Chouki Tibermacine and Luc Fabresse. *An Inheritance System for Structural & Behavioral Reuse in Component-based Software Programming.* In Proceedings of the 11th International Conference on Generative Programming and Component Engineering, GPCE '12, pages 60–69, New York, NY, USA, 2012. ACM. (Cited on page 44.)

[Spacek 2014] Petr Spacek, Christophe Dony and Chouki Tibermacine. *A Component-based Meta-level Architecture and Prototypical Implementation of a Reflective Component-based Programming and Modeling Language.* In Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering, CBSE '14, pages 13–22, New York, NY, USA, 2014. ACM. (Cited on pages 72 and 73.)

[Stevanetic 2015] Srdjan Stevanetic, Muhammad Atif Javed and Uwe Zdun. *The Impact of Hierarchies on the Architecture-Level Software Understandability-A Controlled Experiment.* In Software Engineering Conference (ASWEC), 2015 24th Australasian, pages 98–107. IEEE, 2015. (Cited on page 15.)

[Stojanović 2005] Zoran Stojanović and Ajantha Dahanayake. Service-oriented software system engineering: challenges and practices. IGI Global, 2005. (Cited on page 15.)

[Stroustrup 1991] Bjarne Stroustrup. *What is ââObject-Oriented Programmingââ?(1991 revised version).* In Proc. 1st European Software Festival, 1991. (Cited on page 45.)

[Szyperski 2002] Clemens Szyperski. Component software: Beyond object-oriented programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd édition, 2002. (Cited on pages 18, 19, 31, 34, 44 and 51.)

[T Genssler 1999] B Schulz T Genssler. Transforming inheritance into composition â a reengineering pattern. 1999. (Cited on page 57.)

[Teiniker 2004] Egon Teiniker and Leif Johnson. *CCM Tools.* 2004. (Cited on page 115.)

[Tempero 2010] E. Tempero, C. Anslow, J. Dietrich, T. Han, Jing Li, M. Lumpe, H. Melton and J. Noble. *The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies.* In Software Engineering Conference (APSEC), 2010 17th Asia Pacific, pages 336–345, Nov 2010. (Cited on page 58.)

[Thomas 2001] Bill Thomas and Scott Tilley. *Documentation for software engineers: what is needed to aid system understanding?* In Proceedings of the 19th annual international conference on Computer documentation, pages 235–236. ACM, 2001. (Cited on page 15.)

[Tichelaar 2000] Sander Tichelaar, Stéphane Ducasse and Serge Demeyer. *Famix and xmi*. In Reverse Engineering, 2000. Proceedings. Seventh Working Conference on, pages 296–298. IEEE, 2000. (Cited on page 91.)

[Tilevich 2009] Eli Tilevich and Yannis Smaragdakis. *J-Orchestra: Enhancing Java Programs with Distribution Capabilities*. ACM Trans. Softw. Eng. Methodol., vol. 19, no. 1, pages 1:1–1:40, August 2009. (Cited on pages 17, 19, 23, 28 and 30.)

[Tomer 2004] Amir Tomer, Leah Goldin, Tsvi Kuflik, Esther Kimchi and Stephen R Schach. *Evaluating software reuse alternatives: a model and its application to an industrial case study*. Software Engineering, IEEE Transactions on, vol. 30, no. 9, pages 601–612, 2004. (Cited on page 15.)

[too 2012] *Moose technology: Home*,, 2012. (Cited on pages 112 and 114.)

[Van Deursen 2007] Arie Van Deursen, Eelco Visser and Jos Warmer. *Model-driven software evolution: A research agenda*. Rapport technique, Delft University of Technology, Software Engineering Research Group, 2007. (Cited on page 86.)

[Visser 2001] Eelco Visser. *A survey of rewriting strategies in program transformation systems*. Electronic Notes in Theoretical Computer Science, vol. 57, pages 109–143, 2001. (Cited on page 26.)

[Visser 2005] Eelco Visser. *A survey of strategies in rule-based program transformation systems*. Journal of Symbolic Computation, vol. 40, no. 1, pages 831–873, 2005. (Cited on page 25.)

[Vliet 2008] Hans van Vliet. Software engineering: Principles and practice. Wiley Publishing, 3rd édition, 2008. (Cited on page 11.)

[Vlissides 1995] John Vlissides, Richard Helm, Ralph Johnson and Erich Gamma. *Design patterns: Elements of reusable object-oriented software*. Reading: Addison-Wesley, vol. 49, no. 120, page 11, 1995. (Cited on pages 48 and 51.)

[Wagner 2014] Christian Wagner. Model-driven software migration: A methodology: Reengineering, recovery and modernization of legacy systems. Springer Science & Business Media, 2014. (Cited on pages 3 and 16.)

[Wang 2006] Xinyu Wang, Jianling Sun, Xiaohu Yang, Chao Huang, Zhijun He and Srinivasa R. Maddineni. *Reengineering Standalone C++ Legacy Systems into the J2EE Partition Distributed Environment*. In Proceedings of the 28th

International Conference on Software Engineering, ICSE '06, pages 525–533, New York, NY, USA, 2006. ACM. (Cited on pages 17, 19, 20, 23, 28 and 30.)

[Washizaki 2005] Hironori Washizaki and Yoshiaki Fukazawa. *A technique for automatic component extraction from object-oriented programs by refactoring*. Science of Computer programming, vol. 56, no. 1, pages 99–116, 2005. (Cited on pages 5 and 43.)

[Waters 1994] Richard C Waters and Elliot Chikofsky. *Reverse engineering*. Communications of the ACM, vol. 37, no. 5, pages 22–26, 1994. (Cited on page 19.)

[Weck 1996] Wolfgang Weck and Clemens Szyperski. *Do we need inheritance*. In Proc. of the CIOO Workshop at ECOOP. Citeseer, 1996. (Cited on page 49.)

[Weiderman 1997] Nelson H Weiderman, John K Bergey, Dennis B Smith and Scott R Tilley. *Approaches to Legacy System Evolution*. Rapport technique, DTIC Document, 1997. (Cited on page 3.)

[Weis 2003] Torben Weis, Andreas Ulbrich and Kurt Geihs. *Model metamorphosis*. IEEE software, vol. 20, no. 5, page 46, 2003. (Cited on pages 25 and 86.)

[Wilde 1991] Norman Wilde and Ross Huitt. *Maintenance support for object oriented programs*. In Software Maintenance, 1991., Proceedings. Conference on, pages 162–170. IEEE, 1991. (Cited on page 32.)

[Winter 2002] M. Winter. *The PECOS software process*. In Workshop on Components-based Software Development Processes, ICSR, 2002. (Cited on page 73.)

[Winter 2007] Victor L. Winter and Azamat Mametjanov. *Generative Programming Techniques for Java Library Migration*. In Proceedings of the 6th International Conference on Generative Programming and Component Engineering, GPCE '07, pages 185–196, New York, NY, USA, 2007. ACM. (Cited on pages 17, 19, 23, 28 and 30.)

[Xue 2011] Yinxing Xue. *Reengineering Legacy Software Products into Software Product Line Based on Automatic Variability Analysis*. In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 1114–1117, New York, NY, USA, 2011. ACM. (Cited on pages 17, 19, 23, 28 and 30.)

[Yan 2004] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich and Rick Kazman. *Discotect: A system for discovering architectures from running systems*. In Proceedings of the 26th International Conference on Software Engineering, pages 470–479. IEEE Computer Society, 2004. (Cited on page 33.)

[Yemini 1985] Shaula Yemini and Daniel M Berry. *A modular verifiable exception handling mechanism.* ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 7, no. 2, pages 214–243, 1985. (Cited on page 45.)

[Ying 2013] Ming Ying and James Miller. *Refactoring legacy {AJAX} applications to improve the efficiency of the data exchange component.* Journal of Systems and Software, vol. 86, no. 1, pages 72 – 88, 2013. (Cited on pages 17, 19, 23, 28 and 30.)

[Zou 2002] Ying Zou and Kostas Kontogiannis. *Migration to object oriented platforms: A state transformation approach.* In Software Maintenance, 2002. Proceedings. International Conference on, pages 530–539. IEEE, 2002. (Cited on page 16.)

La Migration des Applications Orientées-Objet vers Celles à Base de Composants

Abstract: Les applications orientées objet de tailles significatives ont des dépendances complexes et nombreuses, et généralement ne disposent pas d'architectures logicielles explicites. Par conséquent, elles sont difficiles à maintenir, et certaines parties de ces applications sont difficiles à réutiliser. Le paradigme de développement à base de composants est né pour améliorer ces aspects et pour soutenir la maintenabilité et la réutilisation efficaces. Il offre une meilleure compréhensibilité à travers une vue d'architecture de haut niveau. Ainsi, la migration des applications orientées objet à celles à base de composants contribuera à améliorer ces caractéristiques, et de soutenir l'évolution des logiciels et la future maintenance.

Dans cette thèse, nous proposons une approche pour transformer automatiquement les applications orientées objet à celles à base de composants. Plus particulièrement, l'entrée de l'approche est le résultat fourni par la récupération de l'architecture logicielle: une description de l'architecture à base de composants. Ainsi, notre approche transforme le code source orienté objet afin de produire des composants déployables. Nous nous concentrons sur la transformation des dépendances orientées objet en celles basées sur les interfaces. De plus, nous passons du concept d'objet au concept d'instance d'un composant. En outre, nous fournissons une approche de transformation déclarative en utilisant des langages dédiés. Nous démontrons notre approche sur de nombreux modèles de composants bien connus.
Keywords: Component-based, Object-oriented, Software Migration, Reengineering, Reverse engineering, Transformation, Model-driven, Software evolution, Software maintenance, Software reuse, Design pattern.