Supporting Reuse by Reverse Engineering Software Architectures and Components from Object-Oriented Product Variants and APIs

by

Anas Shatnawi

UNIVERSITY OF MONTPELLIER

PHD THESIS

to obtain the title of

PhD of Computer Science

of the University of Montpellier
Specialty : SOFTWARE ENGINEERING

Defended by

Anas SHATNAWI

Supporting Reuse by Reverse Engineering Software Architectures and Components from Object-Oriented Product Variants and APIs

prepared at Laboratoire d'Informatique de Robotique et de Microctronique de Montpellier, MAREL Team

defended on June 29, 2015

Jury :

Acknowledgments

This dissertation could not have been written without Dr. Abdelhak-Djamel SERIAI who not only served as my supervisor but also encouraged and challenged me throughout my academic program. Besides Dr. SERIAI, I would like to thank my co-advisor Prof. Houari SAHRAOUI. My thanks also go to *MaREL* team. I would like to thank the members of the committee. Thank you for having accepted to review my thesis and for their time in reading and commenting on my thesis. Many thanks to *ERASMUS MUNDUS* for their generous financial support.

Contents

Supporting Reuse by Reverse Engineering Software Architecture and Component from Object-Oriented Product Variants and APIs

Abstract: It is widely recognized that software quality and productivity can be significantly improved by applying a systematic reuse approach. In this context, Component-Based Software Engineering (CBSE) and Software Product Line Engineering (SPLE) are considered as two important systematic reuse paradigms. CBSE aims at composing software systems based on pre-built software components and SPLE aims at building new products by managing commonalty and variability of a family of similar software. However, building components and SPL artifacts from scratch is a costly task. In this context, our dissertation proposes three contributions to reduce this cost.

Firstly, we propose an approach that aims at mining reusable components from a set of similar object-oriented software product variants. The idea is to analyze the commonality and the variability of product variants, in order to identify pieces of code that may form reusable components. Our motivation behind the analysis of several existing product variants is that components mined from these variants are more reusable for the development of new software products than those mined from single ones. The experimental evaluation shows that the reusability of the components mined using our approach is better than those mined from single software.

Secondly, we propose an approach that aims at restructuring object-oriented APIs as component-based ones. This approach exploits specificity of API entities by statically analyzing the source code of both APIs and their software clients to identify groups of API classes that are able to form components. Our assumption is based on the probability of classes to be reused together by API clients on the one hand, and on the structural dependencies between classes on the other hand. The experimental evaluation shows that structuring object-oriented APIs as component-based ones improves the reusability and the understandability of these APIs.

Finally, we propose an approach that automatically recovers the component-based architecture of a set of object-oriented software product variants. Our contribution is twofold: the identification of the architectural component variability and the identification of the configuration variability. The configuration variability is based on the identification of dependencies between the architectural elements using formal concept analysis. The experimental evaluation shows that our approach is able to identify the architectural variability.

Keywords: software reuse, reverse engineering, restructuring, reengineering, object oriented, software component, software product line architecture, software architecture variability, API, product variants.

Support à la réutilisation par la rétro-ingénierie des architectures et des composants logiciels à partir du code source orienté objet des variantes de produits logiciels et d'APIs

Résumé: La réutilisation est reconnue comme une démarche intéressante pour améliorer la qualité des produits et la productivité des projets logiciels. L'ingénierie des logiciels à base de composants (CBSE en anglais) et l'ingénierie des lignes de produits logiciels (SPLE en anglais) sont considérées parmi les plus importants paradigmes de réutilisation systématique. L'ingénierie à base de composants permet de développer de nouveaux systèmes logiciels par composition de briques préconstruites appelées composants. L'ingénierie des lignes de produits logiciels permet de dériver (construire) de nouveaux produits par simple sélection de leurs caractéristiques (feature en anglais). Cette dérivation est rendue possible grâce à la représentation et à la gestion de la variabilité et de la similarité des produits d'une même famille. Cependant, une des difficultés vers une large adoption de l'ingénierie des logiciels à base de composants et des lignes de produits est le coût à investir pour construire, à partir de rien, les composants et les artefacts de lignes de produits. Dans ce contexte, les travaux de cette thèse proposent de réduire ce coût par une démarche basée sur la rétro-ingénierie.

La première contribution de cette thèse consiste à proposer une approche permettant d'identifier, par l'analyse d'un ensemble de variantes d'un produit logiciel orienté objet, les éléments du code source pouvant servir à l'implémentation de composants. Au contraire des approches existantes d'identification de composants basées sur l'analyse d'un seul produit, l'originalité de notre approche consiste en l'analyse de plusieurs variantes de produits en même temps. Notre objectif est l'amélioration de la réutilisabilité des composants extraits. L'évaluation expérimentale menée dans le cadre de cette thèse a montré la pertinence de cette démarche.

La deuxième contribution de cette thèse permet de restructurer les APIs orientées objet en composants. Nous exploitons la spécificité des classes des APIs d'être conçues pour être utilisées par des applications clientes pour identifier ces composants. Le code source de ces APIs et celui de leurs applications clientes sont analysés afin d'identifier des groupes de classes qui peuvent servir à l'implémentation de composants à extraire. L'identification de ces groupes de classes est basée sur l'analyse des liens structurels entre ces classes et sur la probabilité que ces classes soient utilisées ensemble par les applications clientes. Nous montrons à travers les résultats expérimentaux que la restructuration des API orientées objet en composants facilite la réutilisation et la compréhension des éléments de ces APIs.

La troisième contribution consiste en la proposition d'une approche pour l'extraction d'une architecture à base de composants d'un ensemble de variantes d'un produit logiciel orienté objet. Il s'agit d'identifier la variabilité des composants architecturaux et la configuration architecturale. L'identification de la configuration architecturale est principalement basée sur l'utilisation de l'analyse formelle de concepts pour trouver les dépendances entre les éléments architecturaux.

L'expérimentation conduite pour l'évaluation de l'approche proposée confirme la pertinence des éléments identifiés.

Mots-clés: réutilisation logicielle, rétro-ingénierie, restructuration, ré-ingénierie, orienté objet, composants logiciels, architecture logicielle, ligne de produits logiciels, variabilité, API, variantes logicielles.

Chapter 1

Introduction

Contents

1.1 Research Context

1.1.1 Software Reuse

Software reuse refers to the process of building new software systems based on pre-existing software artifacts [Frakes 2005] [Shiva 2007] [Leach 2012]. Software reuse can be applied at different levels of abstraction(i.e., requirement, design and implementation), and can concern different software artifacts (e.g., feature, documentation, software architecture, software component, source code, etc.) [Frakes 1996] [Leach 2012]. When a systematic reuse is applied, the software quality and productivity are significantly improved [Ferreira da Silva 1996] [Frakes 1996]. On the one hand, the quality of software is enhanced because of reusing existing artifacts that are already tested, evaluated and proven in advance [Griss 1997]. On the other hand, the software productivity is improved because of reusing pre-existing software artifacts instead of developing them from scratch. This leads to reductions in both the development cost and the time to market [Frakes 2005] [Mohagheghi 2007].

The concept of systematic software reuse was firstly presented by McIlroy in *1968* [McIlroy 1968]. Since that, many software reuse approaches have been proposed, in order to reach a potential degree of software reuse [Frakes 2005] [Shiva 2007]. Examples of these approaches are Component-Based Software Engineering (CBSE) [Heineman 2001], Software Product Line Engineering (SPLE) [Clements 2002], service-oriented software engineering [Stojanović 2005] and aspect-oriented software engineering [Filman 2004].

1.1.2 Component-Based Software Engineering (CBSE)

CBSE is considered one of the most promising software reuse approaches [Cai 2000] [Gasparic 2014]. It aims at composing software systems based on pre-built software components [Heineman 2001]. This means that the development of software systems is done via components that are already built rather than the development from scratch on the one hand, and coarse-grained software entities (having explicit architecture) instead of fine-grained entities on the other hand [Lau 2007] [Heineman 2001]. In CBSE, components can be independently built, and the interaction between them is done through their required and provided interfaces that explicitly describe their dependencies and their provided services. As a result, the possibility of reusing the components in new software is opened [Land 2009].

1.1.3 Software Product Line Engineering (SPLE)

Instead of developing each software product individually, SPLE promotes a pre-planned software reuse by building and managing a family of software products that are developed in the same domain (called Software Product Line (SPL)) [Clements 2002] [Pohl 2005a]. A SPL is defined as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"[Clements 2002]. The main idea behind SPLE is to analyze a business domain in order to identify the common and the variable parts between the member products [Clements 2002] [Pohl 2005a]. This aims at building a single production line that comprises of a family of software products that can be customized based on common characteristics. This means that a software product is instantiated based on SPL core assets that consist of a set of common and reusable software artifacts [Clements 2002].

SPLE is composed of two phases; domain engineering and application engineering [Pohl 2005a]. In the former, SPL core assets are created based on the analysis of the commonalty and the variability of a SPL. The content of these core assets can be mainly composed of requirements, architectures, components, test cases, source codes [Pohl 2005a]. In the second, the created core assess are used to derive SPL products [Linden 2007] [Pohl 2005a].

One of the most important software artifact composing SPL's core assets is Software Product Line Architecture (SPLA) [DeBaud 1998] [Clements 2002] [Linden 2007] [Nakagawa 2011]. The aim of a SPLA is at highlighting the commonalty and the variability of a SPL at the architecture level [Pohl 2005a]. It does not only describe the system structure at a high level of abstraction, but also describes the variability of a SPL by capturing the variability of architecture elements [Pohl 2005a]. This is done by (i) describing how components can be configured to form a concrete architecture, (ii) describing shared components and (iii) describing individual architecture characteristics of each product.

1.2 Problem and Motivation

Applying systematic software reuse requires an initial investment to develop reusable core assets [Gasparic 2014]. Companies use three investment strategies: proactive, reactive and extractive [Frakes 2005]. In the proactive model, companies develop their reusable core assets from scratch. This requires, at an earlier stage, a large investment for the analysis and planning tasks, while the return values can be only proven when the products are developed [Gasparic 2014]. Thus the proactive strategy is considered as a costly and risky one. In the reactive, companies incrementally develop their reusable core assets during the development of new demands. This reduces the cost and the risk compared to the proactive strategy. In the extractive strategy, companies invest the already developed software systems, in order to reverse engineer the reusable core assets. By doing so, companies significantly reduce the cost compared to the other strategies [Gasparic 2014].

The ultimate goal of our dissertation is at supporting CBSE and SPLE by reverse engineering some of their reusable core assets. In this goal, we address the following research problems.

1. **Reverse engineering reusable software components:** identifying software component from existing object-oriented software is an efficient way for supplying CBSE by feeding component-based libraries. This is done through providing coarse-grained software entities that can be easily reused. Components can be identified at different levels of abstraction (requirement, design and implementation). We focus on the identification of components at the implementation level. In this context, the identification is mainly composed of analyzing the source code, in order to extract pieces of codes (i.e., sets of classes) that may form reusable components [Kebir 2012b].

   (i) Identifying reusable component by analyzing software product variants: numerous approaches have been presented to identify software components from existing object-oriented software like [Kebir 2012a] [Allier 2011] [Hasheminejad 2015]. Nevertheless these ones perform the identification by analyzing a single software product. As a result, the mined components may be useless in other software products and consequently their reusability is not guaranteed. In fact the probability of reusing a component in a new software product is proportional to the number of software products that have already used it [Sametinger 1997] [Gasparic 2014]. Thus mining software components based on the analysis of a set of software products contributes to identify components that are probably more reusable than those identified by analyzing a single software product. Otherwise, companies developed many software products in the same domain, but with functional or technical variations. Each software is developed by adding some variations to an existing software to meet the requirements of a new need. These products are called product variants [Yinxing 2010]. Nonetheless, mining reusable components from a set of product variants has not been investigated in the literature.

(ii) Identifying component by analyzing object-oriented APIs: existing component identification approaches aims at mining components by analyzing the source code of software applications. In this context, dependencies between classes are only realized via calls between methods, sharing types, etc. Nevertheless reengineering object-oriented Application Programming Interfaces (APIs) into component-based ones has not been considered in the literature. In this context, there are two kinds of dependencies. The first one is that classes are structurally dependent. The second one is that some classes need to be reused together to implement a functionality. This kind of dependencies can not be identified by analyzing the source code, but needs the analysis of how software applications use the API classes. For example, in the *android* API, *Activity* and *Context* classes are structurally and behaviorally independent, but they have to be used together to build android applications. This means that classes frequently used together are more favorable to belong to the same component.

2. **Recovering SPLA by analyzing the source code of existing software product variants:** SPLA can be recovered based on the analysis of the source code of the existing product variants. This is done through the exploitation of the commonality and the variability across the source code of product variants.

   In the literature, there are few approaches that recover SPLA from a set of product variants. However these approaches suffer from two main limitations. The first one is that the architecture variability is partially addressed since they recover only some variability aspects, no one recovers the whole SPLA. For example, [Koschke 2009, Frenzel 2007] do not identify dependencies among the architectural elements. The second one is that they are not fully-automatic since they rely on the expert domain knowledge which is not always available, such as [Pinzger 2004] and [Kang 2005].

1.3 Contribution

This dissertation presents three contributions:

1. **Mining software components by analyzing a set of similar object-oriented software product variants:**
   We propose an approach that aims at mining reusable components from a set of similar object-oriented software product variants. The idea is to analyze the commonality and the variability of product variants, in order to identify pieces of code that may form reusable components. Our motivation is that components mined based on the analysis of several existing software products will be more useful (reusable) for the development of new software products than those mined from singular ones. This is shown, thanks to the experimental evaluation.

2. **Mining software components by analyzing object-oriented APIs:**
   We propose an approach that aims at reverse engineering software components by analyzing object-oriented APIs. This approach exploits specificity of API entities by statically analyzing the source code of both APIs and their software clients to identify groups of API classes that are able to form components. This is based on the probability of classes to be reused together by API clients on the one hand, and the structural dependencies between classes on the other hand. The experimental evaluation shows that structuring object-oriented APIs as component-based ones improves the reusability and the understandability of these APIs.

3. **Recovering software architecture of a set of similar object-oriented software product variants:**
   We propose an approach that automatically recovers the architecture of a set of software product variants. Our contribution is twofold: on the one hand, we recover the architecture variability concerning both component and configuration variabilities. On the other hand, we recover the architecture dependencies between the architectural elements. The identification of architecture dependencies is done by capturing the commonality and the variability at the architectural level using formal concept analysis. The experimental evaluation shows that our approach is able to identify the architectural variability and dependencies.

1.4 Thesis Outline

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 reverse engineering software architectures and software components from object-oriented software systems.
- **Chapter 3** presents our contribution related to identify reusable software components based on the analysis of a set of object-oriented product variants.
- **Chapter 4** presents our contribution aiming at mining software components by analyzing object-oriented APIs.
- **Chapter 5** presents our approach that aims at recovering software architecture of a set of object-oriented product variants.
- **Chapter 6** reports conclusion remarks and future directions.

Chapter 2

Reverse Engineering Software Architectures and Software Components from Object-Oriented Software Systems

Contents

In this chapter, we discuss the-state-of-the-art related to reverse engineering software architectures and software components research area. We start by positioning our work compared to the related domains in Section 2.1 and Section 2.2. Then, axes used to classify related approaches are presented in Section 2.3. A detail example of a related approach is presented in Section 2.4. Next, the classification results are discussed in Section 2.5. Lastly, we conclude the chapter in Section 2.6.

2.1 Reverse Engineering

In software engineering, the traditional engineering process is forward engineering. This refers to the movement from a higher level of conceptual abstraction, e.g., software requirements, going down to a lower level of abstraction composed of details, e.g., source code [Vliet 2008]. The opposite of forward engineering is reverse engineering. It refers to the process of recovering and discovering a high level of abstraction, e.g., design and requirement, based on the analysis of a low level of abstraction, e.g., source code [Chikofsky 1990]. In other words, reverse engineering denotes to the movement from physical details going up to conceptual abstractions. Both reverse and forward engineering could be met through reengineering [Demeyer], which is defined as “the examination and alteration of a system to reconstitute it in a new form”[Chikofsky 1990]. For example, the movement of object-oriented software to component-based one. In this context, reverse engineering is required for recovering an abstract description (e.g., component-based architecture), while forward engineering follows the first by identifying implementation details of the new form based on the recovered abstract description (e.g., the component implementation) [Vliet 2008] [Chikofsky 1990]. Figure 2.1 shows these engineering processes and their relationships explained in terms of life cycle phases; requirement, design and implementation.

In the literature, researchers have used many terminologies that refer to the reverse engineering process. These are extraction [Razavizadeh 2009], identification [Mende 2009], mining [Yuan 2014], recovery [Pinzger 2004] and reconstruction [Moshkenani 2012].

The use of reverse engineering is strongly required in the case of legacy software where the source code is the only artifact that exists [Müller 2000] [Demeyer]. In this case, reverse engineering is connected with the concept of analyzing the source code of legacy software, in order to recover other software artifacts at different level of abstractions [Müller 2000]. These artifacts can be mainly classified into two types. The first one is abstract artifacts that provide a higher level of abstraction, such as architecture model [Kebir 2012a], feature model [She 2011], class diagram [Tonella 2001] and so on. The second type refers to concrete software artifacts that can be directly reused in the future development, like component [Allier 2011], feature [Ziadi 2012], service [Sneed 2006] and so on. Figure 2.2 shows this classification.

2.2 Reverse Engineering Software Architectures and Software Components

In this chapter, we focus on the state-of-the-arts related to reverse engineer software architectures, software product line architectures and software components. These are defined as follows:

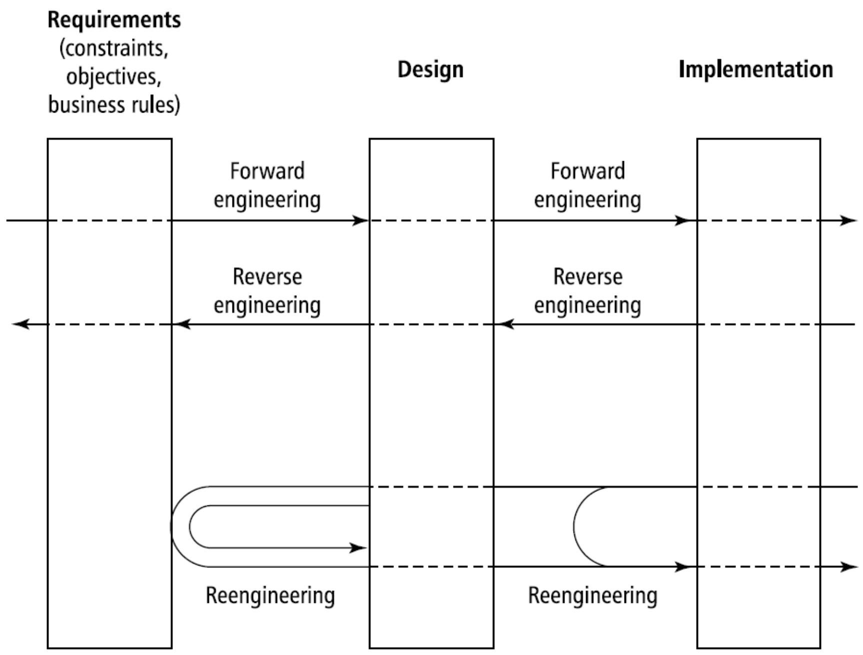


Figure 2.1: Forward engineering, reverse engineering and reengineering [Chikofsky 1990]

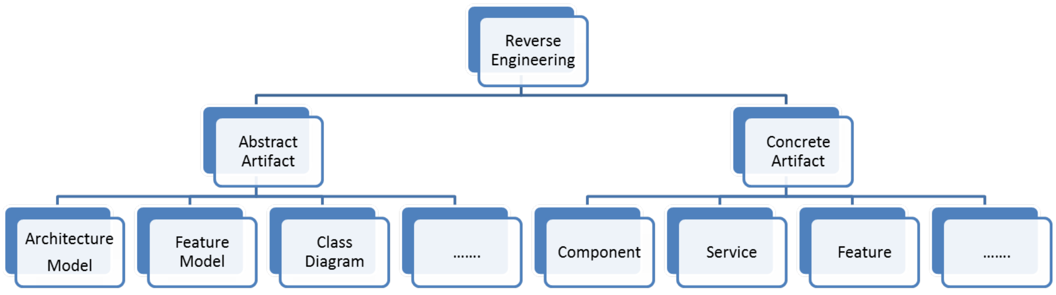


Figure 2.2: Classification of reverse engineering results

Software Architecture

The goal of Software Architecture (SA) is at playing the role of connecting business requirements and technical implementations [Perry 1992]. SA has been defined by several researchers. One of the commonly accepted definition is the one presented by Bass et al. [Bass 2012]. Based on this definition, SA describes the system structure at a high level of abstraction in terms of software elements and their relationships. However, it does not describe the nature of software elements represented in the architecture. Perry and Wolf [Perry 1992] provide a more clearly definition by distinguishing between elements composing the architecture (i.e., processing elements, data elements, and connecting elements). This classification is still used by a large number of definitions. Afterwards, each element has been given a name and its role is clearly defined. For instance, Moriconi [Moriconi 1994] defined six types of architecture elements that can be used to describe the SA. These are component, interface, connector, configuration, mapping, and architecture style or pattern. A component encapsulates a specific set of functionalities. An interface is a logical point of a component that it is used by the component to interact with its environment. A Connector defines the roles of component interactions. A configuration consists of a set of constraints defining the topology of how components are composited. A mapping model defines the correlation between a concrete architecture and the set of vocabularies and formulas of an abstract architecture. Architecture style contains a set of well-defined constraints that should be contented by an architecture written in such a style.

Consequently, SA is considered as a blueprint that defines the system structure at a high level of abstraction in terms of three architecture elements: (i) functional components represent the system decomposition (ii) connectors that describe the communications and connections between components (iii) a configuration that describes the topology of links between components and connectors. According to our definition, SA is mainly composed of three elements. These are software components, connectors and configuration.

Software Product Line Architecture

Software Product Line Architecture (SPLA), aka domain-specific architecture [DeBaud 1998] or reference architecture [Pohl 2005b], is a special kind of software architecture. It is designed for describing SA of a set of similar software products that are developed in the context of Software Product line (SPL) [Clements 2002]. In the literature, many definitions have been presented to define SPLA. These definitions consider SPLA as a core architecture, which captures the variability of a set of software products at the architecture level. However they differ in terms of the variability definition. For instance, a general definition is presented by DeBaud et al. [DeBaud 1998] where SPLA is considered as an architecture shared by their member products and has such a variability degree. This is a very general definition since that it does not specify the nature of the architecture variability. In contrast, Pohl et al. [Pohl 2005a] provide a more accurate definition by specifying the nature of architecture variability. For instance, SPLA includes variation points and variants that are presented in such a variability model. Gomaa [Gomaa 2005] links the architecture variability with the architectural-elements. Thus, in his definition, SPLA defines the variability in terms of mandatory, optional, and variable components, and their connections.

Software Component

Many definitions have been presented in the literature to define Software Components (SCs). Each definition describes a SC from a specific perspective. These definitions can be mainly classified into domain-oriented SCs [Baster 2001], architecture-oriented SCs [Szyperski 2002] or technical-oriented SCs [Lüer 2002].

Domain-oriented definitions focus on SCs that are associated to business concepts, such that a SC encapsulates a set of autonomous business functionalities. For example, Baster et al. [Baster 2001] define a SC 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". Architecture-oriented definitions do not focus on business concept, but focus on the logical aspects of SCs, such as the SC structure and SC interfaces. For instance, Szyperski [Szyperski 2002] defines a SC as "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". Technical-oriented definitions focus on the SC deployment and implementation point of view. For example, Lüer et al. [Lüer 2002] define a SC as "a software element that (a) encapsulates a reusable implementation of functionality, (b) can be composed without modification, and (c) adheres to a component model".

By combining these definitions, a SC is considered 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 closed functionalities, and (d) adheres to a component model"[Kebir 2012a]. Based on this definition, a SC is mainly composed of two parts; internal and external structures. Internal structure encapsulates functionalities that a component implements. External structure manages how a SC interacts with its environment. This is realized by component interfaces; required and provided interfaces. Required interfaces represent services required by the component to achieve its goal. Provided interfaces present services that the component offers.

2.3 Classification Axes of Related Works

The life cycle of an identification approach is composed of goals, inputs, processes and outputs (see Figure 2.3). According to this life cycle, approaches[1] presented in the literature can be classified mainly based on four axes; the target goal, the needed input, the applied process and the desired output. Figure 2.4 shows our classification scheme.

2.3.1 The Goal of Identification Approaches

The goal of an identification approach can be: understanding, reuse, construction, evolution, analysis or management [Garlan 2000]. These concern both software

[1]In our classification, we select approaches based on two criteria. The first one focuses on the approaches that are frequently cited since they are considered as the most known approaches presented in the state-of-the-art. The second one is related to the comprehension of the classification axes. This means that we select approaches that cover all of the classification axes presented in this chapter.

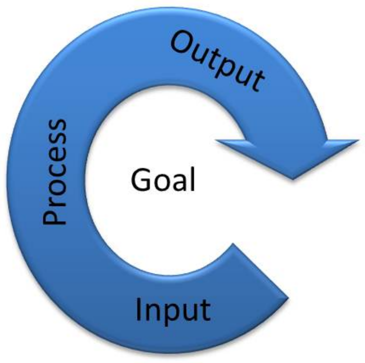


Figure 2.3: The life cycle of identification approaches

architectures and software components. Software understanding is supported by providing a high level of abstraction describing the system structure. Reuse is supported by providing a coarse-grain software entities that can be easily decoupled from the system and deployed in another one. Construction is guided by explaining how software components interact with each other through their interfaces. A better comprehension of the outcome changes is provided to software maintainers. Thus they can be more precise in estimating cost of modifications of the software evolution. Software analysis is enriched by understanding the dependencies provided by software architectures. Managing the development tasks get success, when a clear view of the system structure is provided [Garlan 2000].

2.3.2 The Required Input of Identification Approaches

The input of identification approaches can be: source codes, documentations, historical information and human expert knowledges.

2.3.2.1 Source code

Source code is a collection of computer instructions describing the system execution [Harman 2010]. It is written in a computer programming language, such as Java, C++, and C. Software developers use the source code to realize the actions that need to be performed by computers. In the case of object-oriented systems, source code is mainly composed of a set of classes that are organized in a set of packages [Rumbaugh 1991]. Methods and attributes constituting the main building units of a class. The relationship between classes is realized via method calls, access attributes inheritance, and so on. Source code is the most commonly used software artifact by the existing identification approaches, due to its availability. The identification approaches relied on relationships among classes to analyze the system structure, in order to identify high cohesive and low coupling parts. In first hand, for SAI, classes are grouped into disjoint clusters, such that each cluster represents a component of the system architecture view, like [Chardigny 2008b],

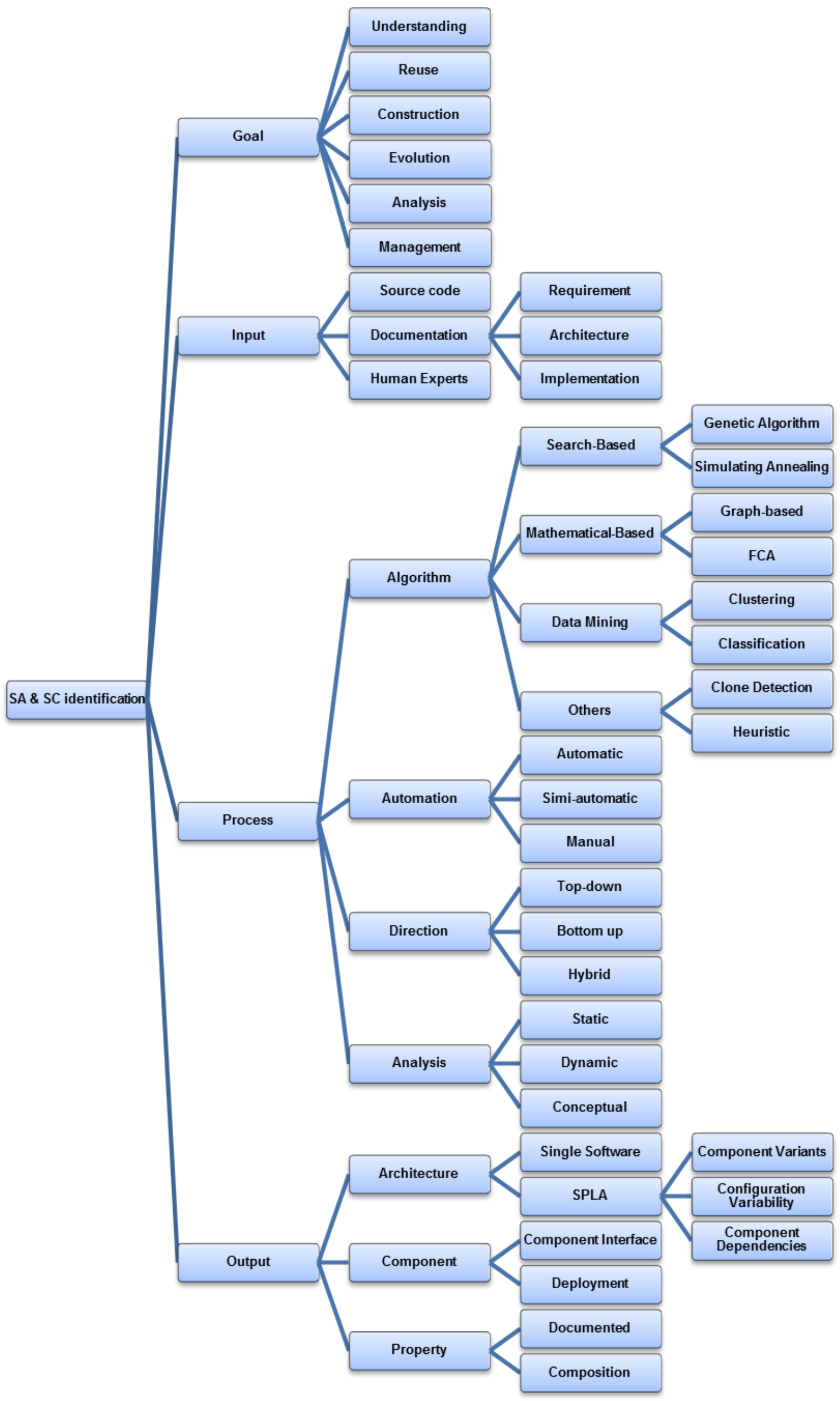


Figure 2.4: Classification axes of software architecture and component identification approaches

[Zhang 2010], [Constantinou 2011] and [Moshkenani 2012]. In the second hand, for SCI, the overlapping is allowed between the clusters since any group of classes filling the quality requirement may form a potential component, such as [Mende 2008] and [Mende 2009].

2.3.2.2 Documentation

Software documentation is a description that is used to document software artifacts at different level of abstractions [Lethbridge 2003]. It can be either text-based or model-based documents. Documentations are used to guide the identification approach by reducing the search space. We distinguish three types of documentations; requirement, design, and implementation.

- At the requirement level, documentations describe what a system should do by documenting user requirements [Pohl 2010]. For example, use cases define how actors, e.g., human and external system, could interact with a system to achieve the goals [Rumbaugh 2004]. Since use cases provide a list of system functionalities in terms of a set of steps, it is the most commonly used documentation by existing approaches. They relied on use cases to derive scenarios for executing the system for the purpose of dynamic analysis. [Allier 2011] [Mishra 2009] and [Riva 2002] are examples of these approaches. Additionally, use cases were used by [Chardigny 2008a], [Hamza 2009] and [Hasheminejad 2015] as guides to extract a set of logical functional components.

- At the design level, software systems are documented by describing their architecture [Bachmann 2000]. Examples of architecture documentations are the number of software components composing the system, the used architecture style and so on. Configuration files are used by [Weinreich 2012] to extract information about the previous architecture, which provides guidances for his identification approach. [Riva 2002] relied on architectural documentation to determine the main concepts constituting the system design. Design documents provide, in [Kolb 2006, Kolb 2005], information about how components have been designed and implemented. In [Chardigny 2010], documentations are used to drive an initial intentional architecture that represents that input of the approach.

- Implementation documentations include a demonstration of how the system should be built [Lethbridge 2003]. A class diagram is an example of an implementation model describing classes and their dependencies. For instance, [Hamza 2009] used class diagrams and [Mishra 2009] used sequence diagrams as well as class diagrams to identify the structural and behavioral dependencies among the system classes.

2.3.2.3 Historical Information

Change history is a log storing changes that have been made to a system. It is the less commonly used source of information. In [Dugerdil 2013], it is invested to identify execution trace scenarios that are used to dynamically analyze a system. In [Chardigny 2009] it is used to support driving the system partition.

2.3.2.4 Human expert Knowledge

Human experts are persons who have relevant information about the design and the implementation of a software system. In most cases, they are software architects. A few approaches utilized human experts in their approaches. For instance, they are used to select the main architecture view that needs to be the core of SPLA in [Pinzger 2004]. In [Chardigny 2010], the authors relied on them to provide an initial intentional architecture. Human experts guide the identification by, for example, classifying the architecture units and identifying the dependencies among the components in [Kang 2005]. In [Riva 2002], experts add a new architectural information that is not immediately evident from the identified one from the source code. [Erdemir 2011] depended on them to analyze and evaluate the resulted architecture, and thus modify it as needed.

2.3.2.5 Combination of Multiple Inputs

A combination of the mentioned input resources is used by some approaches. In some cases, the combination of multiple input resources is necessary, while in some others, it is an optional one. It is required for approaches that need more than one input to achieve their goal. For instance, approaches that aim at identifying software architecture based on the dynamic analysis need at least the source code and another input explaining the execution scenarios. [Allier 2009] relied on use cases and [Dugerdil 2013] depend on the log history of the system end-users. The optionality of a combination has many situations. Firstly, some approaches need to reduce the search space, such as [Chardigny 2010] which utilized the documentation and the human experts to minimize their search space in the source code. Secondly, a combination is made by investing the human experts knowledge to analyze and modify the resulted architecture, like [Erdemir 2011]. Thirdly, a previous architectural information maybe used as a guide of the identification approaches, such as [Weinreich 2012] and [Pinzger 2004].

In Table 2.1, the classification results of the identification approaches are presented in terms of software artifacts that are required as inputs.

2.3.3 The Process of Identification Approaches

The process of an identification approach refers to the way that how the approach achieves its goal based on the input resources. It is mainly composed of five aspects. These are related to algorithms used to implement the approach, the process

Table 2.1: Classification results based on the input software artifacts

Approach	Source code	Req. doc.	Design doc.	Impl. doc.	History	Human experts
[Riva 2002]	X	X	X			X
[Pinzger 2004]	X		X			X
[Kang 2005]	X					X
[Kolb 2005, Kolb 2006]	X		X			
[Chardigny 2008b]	X					
[Mende 2008, Mende 2009]	X					
[Koschke 2009, Frenzel 2007]	X					
[Hamza 2009]		X		X		
[Mishra 2009]	X	X		X		
[Allier 2009]	X	X				
[Razavizadeh 2009]	X					
[Zhang 2010]	X					
[Allier 2010]	X	X				
[Chardigny 2010]	X		X			X
[Allier 2011]	X	X				
[Constantinou 2011]	X					
[Erdemir 2011]	X					X
[Boussaidi 2012]	X					
[Weinreich 2012]	X		X			
[Moshkenani 2012]	X					
[Kebir 2012a]	X					
[Kebir 2012b]	X					
[von Detten 2012, von Detten 2013]	X					
[Dugerdil 2013]	X				X	
[Seriai 2014]	X					
[Hasheminejad 2015]		X				
[Chihada 2015]	X					

automation, the process direction and the used analysis type.

2.3.3.1 Identification Algorithm

Algorithms are used to solve computational problems [Cormen 2009]. In our context, software architecture and component identification is the problem that needs to be solved. This problem is considered as NP-hard problem since identifying the optimal solution is not reachable in a polynomial time complexity (i.e., need to evaluate all possible solutions, then take the best one). Thus, in the literature, the authors relied on several algorithms that aim at identifying a near optimal solution (i.e., good-enough solution). These algorithms can be classified mainly into five types; search-based, data mining, mathematical-based, clone detection, and heuristic algorithms.

- Search-based algorithms are meta-heuristic techniques that formulate the search problems as optimization ones [Harman 2001]. In this type of algorithms, the search space is composed of a set of problem solutions (e.g., the possible software architectures). Then, the algorithms proceed a series of iterations to find a near optimal solution, e.g., software architecture, in the search space. Existing approaches mainly used two kinds of search-based algorithms; genetic algorithm and simulating annealing. For instance, genetic algorithm is used by [Hasheminejad 2015] to partition the requirements into a set of logical components and by [Kebir 2012b] to partition the source code into groups, such that each group is a component. Simulating annealing is used in [Chardigny 2008b, Chardigny 2010] to partition the source code. Since genetic algorithm could reach a trap of a local optimum and simulating annealing could reach a trap of global optima [Elhaddad 2012], a hybrid search is applied to avoid these traps in [Allier 2010, Allier 2011].

- Data mining is a knowledge discovery technique that aims at extracting hidden interesting information from huge data [Han 2006]. In the context of software architecture and component identification, the hug data is the input software artifacts (e.g., source code), while software architecture and components are the interesting knowledges that need to be mined. There are several data mining algorithms. Existing approaches mainly used two algorithms, clustering and classification.

  Clustering is the process of grouping a set of similar objects into clusters, such that objects inside a cluster are similar to each other and dissimilar to other clusters' objects. Since the number of classes is unknown, clustering is an unsupervised learning algorithm [Han 2006]. Clustering algorithms are widely used by a bunch of approaches to partition the source code of object-oriented systems into groups, where each cluster is a component, like [Dugerdil 2013], [von Detten 2012, von Detten 2013], [Boussaidi 2012], [Erdemir 2011], [Kebir 2012a, Kebir 2012b], [Mishra 2009] and [Zhang 2010].

  Classification is a supervised learning categorization process that distributes a set of objects into a set of predefined classes [Han 2006]. It consists of two steps; model construction and model usage. In model construction, the model is learned by giving it a set of objects having known classes. Based on these objects, the model extracts such a procedure, i.e., roles, to classify new unknown objects in model usage step [Han 2006]. Two of the existing approaches utilized classification algorithms to identify software architecture. In [Chihada 2015], Support Vector Machine algorithm is used. The algorithm takes as input a set of design pattern implementations that were manually identified by the software architects. Then, Support Vector Machine is used to recognize design patterns of new source codes. Naïve Bayes classifier is used by [Moshkenani 2012] to reconstruct the architecture of legacy software.

- Mathematical-based algorithms are mathematical data analysis techniques. In the literature, there are mainly two types of used algorithms. These are Formal Concept Analysis (FCA) [Ganter 2012] and graph-based analysis [Cormen 2009].

  FCA is developed based on lattice theory [Ganter 2012]. It allows the analysis of the relationships between a set of objects described by a set of attributes. In this context, maximal groups of objects sharing the same attributes are called formal concepts. These are extracted and then hierarchically organized into a graph called a concept lattice. FCA is used as a clustering technique by [Allier 2009] to cluster object-oriented classes, [Hamza 2009] to partition software requirements into functional components and [Seriai 2014] to structure the component interfaces.

Graph based techniques refer to the formulation of the problem as a graph partitioning algorithm. Here, classes are considered as nodes and dependencies among the classes represent edges. This facilitates the understanding of the problem as well as the data processing. This type of formulation is used by [Erdemir 2011], [Constantinou 2011], [Zhang 2010], [Frenzel 2007] and [Mende 2008, Mende 2009].

- Clone detection algorithms aim at identifying duplicated source codes. In the context of migrating product variants into a SPL, clone detection algorithms are used to identify peaces of source codes that exist in many product variants. In [Koschke 2009, Frenzel 2007], clone detection is used to recover SPLA. The authors determine the architecture view of the first product as a core one. Then, using a clone detection algorithm, they map the implementation of the second product to the implementation of the first architecture model and so on. In [Kolb 2005, Kolb 2006], legacy software components are refactored to be reused in the context of SPLs. The authors used a clone detection algorithm to identify similar components that could integrate the variability. In [Mende 2008, Mende 2009], the approach depended on a clone detection algorithm to identify pairs of functions that share most of their source code, such that these functions are used to form components.

- Some approaches proposed their own heuristic algorithms, instead of using predefined algorithms. [Weinreich 2012], [Razavizadeh 2009], [Kang 2005], and [Pinzger 2004] are some examples of these approaches.

A combination of different types of the above algorithms has been applied by some of the existing approaches. The idea of the combination is to improve the accuracy. For instance, a combination of clustering and genetic algorithms is applied by [Kebir 2012b]. Graph-based have been combined with clustering algorithm by [Erdemir 2011] and [Zhang 2010]. Clone detection algorithms are applied on a graph-based in [Mende 2008, Mende 2009].

2.3.3.2 Process Automation

The automation of the identification process refers to the degree in which this process needs human experts interactions. In other words, it corresponds to how much an approach relies on human experts. We distinguish three types of automations; manual, semi-automatic and full automatic.

- Manual approaches fully depend on human experts. These ones only provide guides for the experts, such as visual analysis of the source code, that allow them to identify architectural elements. [Lanza 2003] and [Langelier 2005] are examples of visualization tools.

- Semi-automatic identification approaches need human expert recommendations to perform their tasks. For example, [Chardigny 2010] approach needs as an input an initial intentional architecture provided by human experts. [Pinzger 2004] relies on software architects to select the main architecture view that needs to be the core of SPLA and to manually analyze design documents. In [Erdemir 2011], software architects need to interact with the approach steps. [Kang 2005] and [Riva 2002] are also examples of semi-automatic approaches.

- Full automated approaches do not need any human interactions. There are no purely automated approaches (do not need software architects at all). We consider approaches that do not have a high impact of human interactions on their results as fully automated ones. For example, software architects need to determine some threshold values. As examples of these approaches [Kebir 2012a, Kebir 2012b], [Chardigny 2008b], [Mende 2008, Mende 2009], [Seriai 2014] and [Hasheminejad 2015].

2.3.3.3 Process Direction

The process of identification approaches can be performed in three directions. These are top-down, bottom-up and hybrid directions.

- Top-down process identifies low level software artifacts by analyzing higher level ones. For example, it identifies software architecture based on software requirements analysis. In [Ducasse 2009], the authors relied on the execution traces, identified from use cases, to recover the architecture of a corresponding system by mapping the traces classes to clusters. [Hamza 2009] and [Hasheminejad 2015] partition software requirements into functional components.

- Bottom-up process starts from low level software artifacts to identify higher level ones. For instance, starting from the source code, it extracts a higher abstraction artifact, e.g., software architecture, that can be used to understand the system structure. In [Boussaidi 2012], [Erdemir 2011], [Kebir 2012a], [Kang 2005], [Koschke 2009, Frenzel 2007] and [Zhang 2010], the authors extract the software architecture based on the source code analysis.

- Hybrid process refers to the combination of a top-down process and a bottom-up one. It starts from software requirements by applying a top-down process and from the source code by applying a bottom-up process to identify the corresponding software architecture. [Allier 2009, Allier 2010, Allier 2011], [Mishra 2009] and [Riva 2002] are examples of this type, since they analyzed use cases (the high level) and source code classes (the low level) to recover the software architecture.

2.3.3.4 Analysis type

Identification approaches may perform the software analysis either statically, dynamically or conceptually to identify the relationships between software artifacts, in order to reverse engineer software architectures/components.

- Static analysis is performed without executing the software system. In source code, dependencies between classes are potential relationships, like method calls and access attributes. These dependencies are analyzed to identify strongly connected classes, for example, to identify components. [Pinzger 2004], [Kebir 2012b, Kebir 2012a], [Weinreich 2012] and [Boussaidi 2012] are examples of approaches that used the static analysis. The main advantage of static analysis is the fact that it depends only on the source code. However, it does not address polymorphism and dynamic binding. In addition, it does not allow to distinguish between the used and unused source code.

- Dynamic analysis is performed by examining the software system at the run time. Dependencies between software elements are collected during the program executing. Thus they refer to the actual relationships. The execution is performed based on a set of cases that covers the system functionalities, called execution scenarios. Use cases, usage log and sequence diagram are respectively used to identify execution scenarios by [Allier 2009], [Dugerdil 2013] and [Mishra 2009]. In dynamic analysis, polymorphism and dynamic binding are addressed and unused source code is excluded from the analysis. However, in the case of losing use cases, the corresponding system functionalities can not be covered by dynamic analysis since the corresponding classes would not be executed.

- Conceptual analysis refers to the lexical analysis of the source code. This analysis supposes that the similarity between the classes should be taken into account during the identification process. This analysis plays the main role in approaches that used clone detection techniques, such as [Koschke 2009, Frenzel 2007] [Mende 2008, Mende 2009], and [Kolb 2005, Kolb 2006].

A combination between the above analysis types can be applied. The idea is to reduce the limitation of each analysis type when it is applied separately. [Allier 2010, Allier 2011] relied firstly on the dynamic analysis to identify classes that have frequently appeared together in execution traces. Then, the static analysis is used to investigate classes that do not occur in the execution traces. In contrast, in [Riva 2002], the static analysis is firstly realized by exploring the source code to identify significant architectural elements. Then, the dynamic analysis is performed. In [Koschke 2009, Frenzel 2007], the static analysis is used to identify architectural view from the source code and the conceptual analysis is used to detect architecture elements having similar implementations among product variants.

Table 2.2 presents the results of our classification based on the applied process. This includes the used algorithm, the process automation, the direction and the analysis type. *GA*, *SA*, *CD*, *H*, *TD*, *BU*, *HP*, *D*, *S* and *C* respectively refer to genetic algorithm, simulating annulling, clone detect, heuristic, top-down, bottom-up, hybrid, dynamic analysis, static analysis and conceptual analysis.

Table 2.2: Classification results based on the applied process

Approach	Algorithm								Automation		Direction			Analysis		
	Search-Based		Math-Based		Data Mining		Others		Semi	Full	TD	BU	HP	D	S	C
	GA	SA	Graph	FCA	Cluster	Classify	CD	H								
[Riva 2002]			X					X	X				X	X	X	
[Pinzger 2004]								X	X			X		X	X	X
[Kang 2005]								X	X			X			X	
[Kolb 2005, Kolb 2006]							X	X	X			X			X	X
[Chardigny 2008b]		X								X		X			X	
[Mende 2008, Mende 2009]			X				X			X		X			X	X
[Koschke 2009, Frenzel 2007]							X	X	X			X			X	X
[Hamza 2009]				X				X	X		X			X		
[Mishra 2009]					X			X	X				X	X	X	
[Allier 2009]				X				X		X			X	X		
[Razavizadeh 2009]								X	X			X			X	
[Zhang 2010]			X		X					X		X			X	
[Allier 2010]	X	X								X			X	X	X	
[Chardigny 2010]		X							X				X		X	
[Allier 2011]	X	X							X				X	X	X	
[Constantinou 2011]			X					X		X		X			X	
[Erdemir 2011]			X		X			X	X			X			X	
[Boussaidi 2012]					X			X	X			X			X	
[Weinreich 2012]								X		X			X		X	
[Moshkenani 2012]						X				X		X			X	
[Kebir 2012a]					X					X		X			X	
[Kebir 2012b]	X				X					X		X			X	
[von Detten 2012, von Detten 2013]					X			X	X			X			X	
[Dugerdil 2013]					X				X		X			X	X	
[Seriai 2014]				X				X		X		X			X	
[Hasheminejad 2015]	X									X	X				X	
[Chihada 2015]						X				X		X			X	

2.3.4 The Output of Identification Approaches

The output of an identification approach is related to the goal of the approach. Thus it can be SA, SPLA or SCs. In this section, we classify identification approaches based on the results that can be obtained. Most of the existing approaches identify SA of a single object-oriented system, such as [Erdemir 2011], [Moshkenani 2012], [Hamza 2009] and [Chihada 2015]. These approaches may provide different results. In the first hand, some approaches provide a documentation about the extracted SA, like [Ducasse 2009], [Kebir 2012a] and [von Detten 2012, von Detten 2013]. In the second hand, some ones support hierarchical architecture by providing a multi-layer SA, such as [Boussaidi 2012], [Constantinou 2011] and [Riva 2002]. In the third hand, some approaches support CBSE by providing SCs that can be reused. This is done by structuring component interfaces, like [Allier 2010], [Mishra 2009] and [Chihada 2015], or by providing deployable (operational) SCs, like [Allier 2011] in which the identified SCs are integrated with the OSGi component model.

The recovery of SPLA is supported by identifying component variants in [Kolb 2005, Kolb 2006] and [Koschke 2009, Frenzel 2007], component variability in [Pinzger 2004], [Kang 2005] and [Koschke 2009, Frenzel 2007], and dependencies between components in [Kang 2005]. [Mende 2008, Mende 2009] is an example of approaches aiming at identifying reusable components from the source code of a set of product variants.

The classification results concerning the output axes are presented in Table 2.3, where *Cvariant, Cvariab, Cdep, Com, Com Int, Dep* and *Hier* respectively refer to component variants, component variability, component dependencies, component, component interfaces, deployable and hierarchical composition.

Table 2.3: The results of classification based on the output of identification approaches

Approach	Architecture				Component			Property	
	Single	Multiple Software			Com	Com Int	Dep	Doc	Hier
		Cvariant	Cvariab	Cdep					
[Riva 2002]	X								X
[Pinzger 2004]			X						
[Kang 2005]			X	X					
[Kolb 2005, Kolb 2006]		X						X	
[Chardigny 2008b]	X								
[Mende 2008, Mende 2009]					X				
[Koschke 2009, Frenzel 2007]		X	X						
[Hamza 2009]	X								
[Mishra 2009]	X				X	X			
[Allier 2009]	X				X				
[Razavizadeh 2009]	X								X
[Zhang 2010]	X								
[Allier 2010]	X				X				
[Chardigny 2010]	X								
[Allier 2011]	X				X	X	X		
[Constantinou 2011]	X								X
[Erdemir 2011]	X								
[Boussaidi 2012]	X							X	X
[Weinreich 2012]	X								X
[Moshkenani 2012]	X								
[Kebir 2012a]	X				X	X		X	
[Kebir 2012b]	X								
[von Detten 2012, von Detten 2013]	X				X	X		X	
[Dugerdil 2013]	X							X	
[Seriai 2014]						X			
[Hasheminejad 2015]	X								
[Chihada 2015]	X								

2.4 Example of Identification Approach: ROMANTIC

In this section, we detail one component identification approach that aims to analyze a single software application. We select this approach to present how the process is done in detail. In addition, we rely on its quality model as a black box model that is used to measure the quality of a group of classes to form a component.

In [Chardigny 2008b] and [Kebir 2012a], 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.

2.4.1 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 components' classes. Classes that form the external structure of a component define provided and required interfaces. Figure 2.5 shows the object-to-component mapping model.

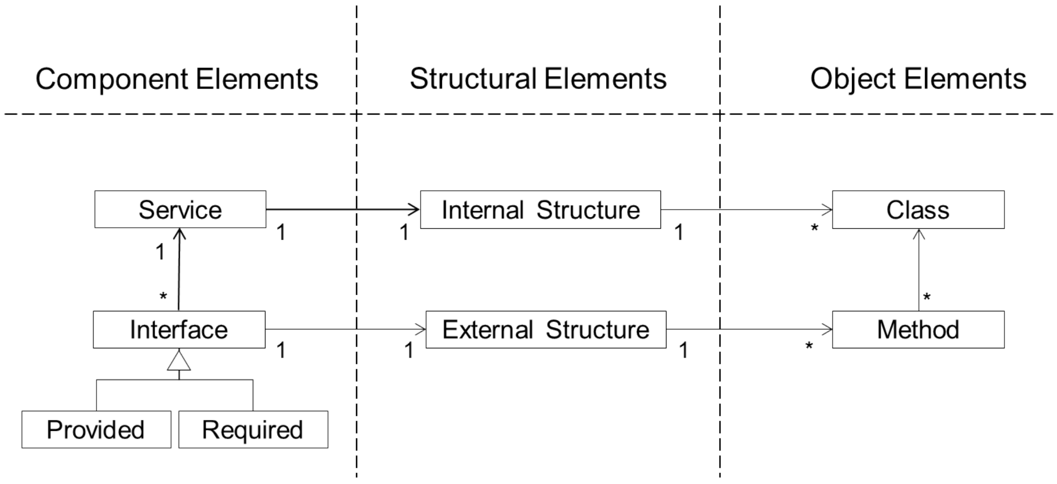


Figure 2.5: Object-to-component mapping model

2.4.2 Quality Measurement Model

According to [Szyperski 2002] [Lüer 2002] and [Heineman 2001], 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 2012a]. Based on this definition, ROMANTIC identifies three quality characteristics of a component: composability, autonomy and specificity [Chardigny 2008b]. 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 2.6 shows how the component characteristics are refined following the proposed measurement model.

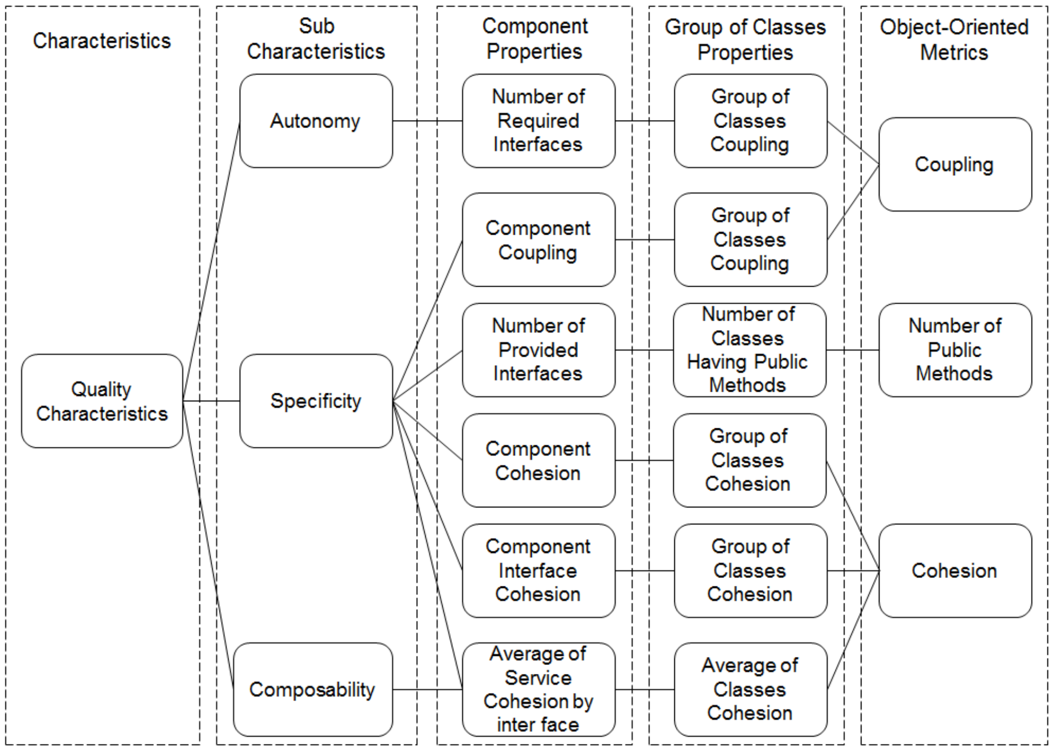

Characteristics
Sub Characteristics
Component Properties
Group of Classes Properties
Object-Oriented Metrics
Quality Characteristics
Autonomy
Specificity
Composability
Number of Required Interfaces
Component Coupling
Number of Provided Interfaces
Component Cohesion
Component Interface Cohesion
Average of Service Cohesion by inter face
Group of Classes Coupling
Group of Classes Coupling
Number of Classes Having Public Methods
Group of Classes Cohesion
Group of Classes Cohesion
Average of Classes Cohesion
Coupling
Number of Public Methods
Cohesion

Figure 2.6: 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:

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

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 2008b] and [Kebir 2012a].

This component quality function is applied in a hierarchical clustering algorithm [Kebir 2012a, Chardigny 2008b] as well as in search-based algorithms [Chardigny 2008a] 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.

2.5 Discussion

In this section, we discuss findings resulted from our classification. The findings are organized based on the input, the process and the output of identification approaches.

2.5.1 The Required Input of Identification Approaches

Almost all existing approaches relied on the source code as the main source of information to be analyzed. *92.5%* of the approaches used the source code, such that *48%* of the approaches depended only on the source code, while *44.5%* of the approaches took other inputs. In the case of a combination, it is used by *22%* of the approaches with the documentations, *7.5%* of the approaches with the human experts' knowledge, *4%* of the approaches with the historical data, and *11%* of the approaches with both the documentation and the human experts' knowledge. Only few approaches relied only on the documentations (i.e., *7.5%* of the approaches). These are approaches that aim at identifying SA based on a forward engineering technique. Figure 2.7 shows the distribution of the approaches based on the required input.

However, there is a trade off between depending on the source code only or taking other information from documentations and human experts. In the first hand, the source code is always available. Thus approaches used only the source code can be applied to any software system. In the other hand, although documentations and human experts improve the accuracy of identification approaches and reduce the search-space, but they are not always available.

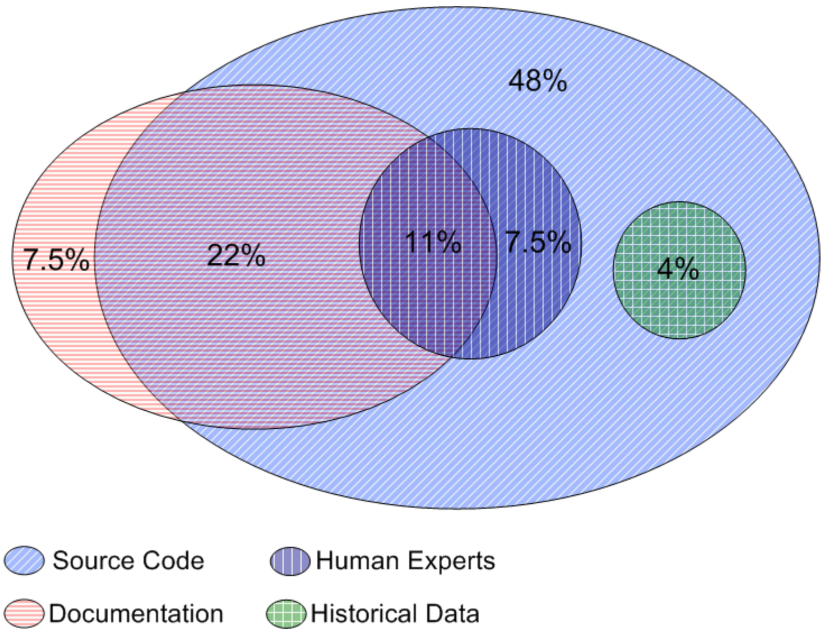


Figure 2.7: The analysis of the approaches based on their inputs

2.5.2 The Process of Identification Approaches

The process applied by identification approaches has many dimensions; the algorithm, the automation, the direction and the analysis type. Several algorithms have been applied. All of them identifies a near optimal system partition. Each one has a degree of accuracy. Search-based algorithms are deployed by *22%* of the approaches, such that half of them (*11%*) for each of genetic algorithm and simulating annulling. Data mining techniques are utilized more than search-based ones where it is used by *37%* of the approaches. These are distributed into *30%* for clustering algorithms and *7%* for classification ones. Mathematical-based algorithms are used by *30%* of the approaches such that graph-based is used by *19%* of the approaches and FCA is used by *11%* of the approaches. Clone detection algorithms are used by *11%* of the approaches. The authors have proposed their heuristic algorithms in *56%* of the approaches. However, in computer science, there are several optimization algorithms that have not been deployed to improve the accuracy.

Certainly, fully-automated approaches are relatively preferred than semi-automated or manual ones. *48%* of the approaches are fully-automatic, while *52%* are semi-automated. The selection of a process direction is based on the input artifacts. If the input is the source code, then a bottom-up process is applied. This is the situation in *63%* of the approaches. If requirement documents are used, then a top-down process need to be applied, *11%* of the approaches. *26%* of the approaches applied a hybrid process direction. Most of the approaches relied on static (*93%*) or dynamic analyses (*30%*), while few ones depend on conceptual analysis (*15%*).

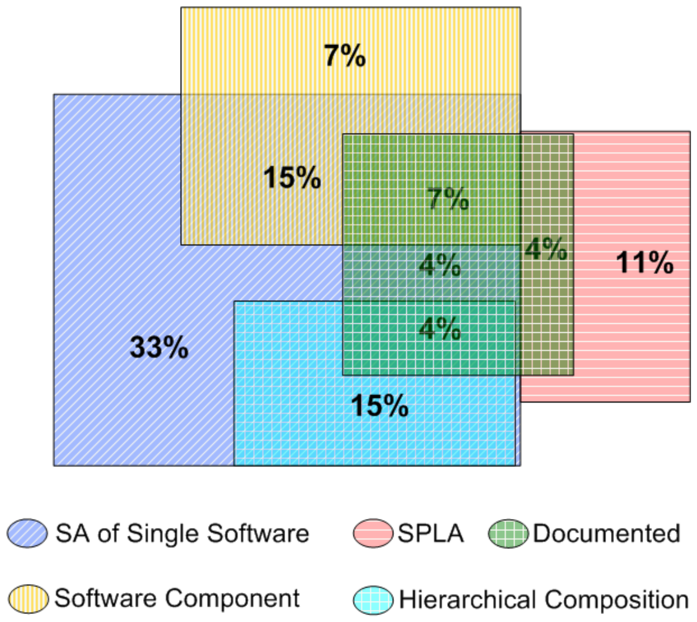


Figure 2.8: The analysis of the approaches based on their outputs

2.5.3 The Output of Identification Approaches

Most of the existing approaches aim at recovering the software architecture of a single software, *78%* of the approaches. The results of these approaches are groups of classes represent a system partition corresponding to component-based architecture. *33%* of these approaches provided only a system partition, while *15%* of them converted the groups into operational components by identifying component interfaces or/and adhering them into such a component model. In addition, the resulted software architecture is documented in few approaches (*19%* of them). A software architecture that supports the hierarchical composition is provided by *19%* of the approaches, such that *4%* of them also provided a documentation. However, a perfect software architecture identification approach can be presented by providing operational component that can be used to constitute component-based libraries. Also, providing documentation will help the software architects to understand and reuse the identified components. Identifying software components, that do not represent a system partition, is presented by *7%* of the approaches.

SPLA has been investigated in few approaches (*15%*, such that *4%* of them provided documentations). However these approaches suffer from two main limitations. One the one hand, the architecture variability is partially addressed. For instance, [Koschke 2009, Frenzel 2007] do not identify dependencies among the architectural elements. However SPLA should be recovered in terms of component variants, variable components, links variability and interfaces variability. On the other hand, they are not fully-automatic. For example, [Pinzger 2004] and [Kang 2005] required domain expert knowledges which is not always available. Otherwise mining software components from object-oriented APIs has not been investigated. Existing approaches only focus on mining software components from object-oriented software applications. Figure 2.8 presents the analysis of the approaches based on their outputs.

2.6 Conclusion

In this chapter, we present the-state-of-the-art related to software architecture and software component identification. This include positioning our dissertation compared to the domain concepts and the related works. Related works are classified based on four axes. These are the goals, the required inputs, the applied processes and the obtained outputs. The chapter is concluded with the following remarks:

First Numerous approaches have been presented to identify software architecture from a single software system.

Second Few approaches recover SPLA from a set of product variants. However, existing SPLA recovery approaches only identify some variability aspects, no one recovers the whole SPLA.

Third Identifying reusable SC based on the analysis of the source code of a set of product variants has not been addressed.

Fourth Mining software components from object-oriented APIs has not been investigated. Existing approaches only focus on mining software components from object-oriented software applications.

Chapter 3

Mining Software Components from a Set of Similar Object-Oriented Product Variants

Contents

3.1 Introduction

Identifying software components from existing object-oriented software is an efficient way that supports software reuse by providing coarse-grained software entities that can be easily reused through their required and provided interfaces [Birkmeier 2009].

To address this issue, numerous approaches have been presented, like [Kebir 2012a] [Allier 2011] [Hasheminejad 2015]. Nevertheless these ones perform the identification by analyzing a single software product. As a result, the mined components may be useless in other software products and consequently their reusability is not guaranteed. In fact the probability of reusing a component in a new software product is proportional to the number of software products that have already used it [Sametinger 1997] [Gasparic 2014]. Thus mining software components based on the analysis of a set of software products contributes to identify reusable components. These components will be more useful (reusable) for the development of new software products than those mined from singular ones. Nonetheless, this has not been investigated in the literature (see Chapter 2).

In this chapter, we propose an approach that aims at mining reusable software components from a set of similar object-oriented software product variants. The main idea is to analyze the commonality and the variability of product variants, in order to identify pieces of code that may form reusable components.

The rest of this chapter is organized as follows: in Section 3.2, we put in context the problem of component identification from product variants. It presents the goal of the proposed approach and the problem analysis. Section 3.3.1 discusses how potential components are extracted. In Section 3.3.2, Similar components are identified. Reusable components are recovered from the similar ones in Section 3.3.3. Section 3.4 presents how to structure the component interfaces. Section 3.5 shows the process of documenting the mined components. Our experimentation is discussed in Section 3.6. Threats to validity are discussed in Section 3.7. A conclusion of this chapter is presented in Section 3.8.

3.2 The Proposed Approach Foundations

3.2.1 The Goal: Object to Component

Our goal is to mine software components based on the analysis of object-oriented software product variants. Based on [Szyperski 2002], [Lüer 2002] and [Heineman 2001], a software component is considered 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 closed functionalities, and (d) adheres to a component model"[Kebir 2012a]. According to this definition, three quality characteristics should be satisfied by a component; *Composability*, *Autonomy* and *Specificity*.

In the context of our approach, the identification[1] of a component means identifying a cluster of object-oriented classes that can be considered as the implementation of this component. Thus we consider that a component can be identified

[1]Component identification is the first step of the migration process of object-to-component

from a collection of classes that may belong to different packages. Classes that have direct links (e.g., method call, attribute access) with classes implementing other components compose the interfaces of the component. Provided Interfaces of a component are defined as a group of methods implemented by classes composing these interfaces. Required interfaces of a component are defined as a group of methods invoked by the component and provided by other components. Figure 3.1 shows our object to component mapping model that is adhered from ROMANTIC approach [Chardigny 2009] [Kebir 2012a] (see Section 2.4).

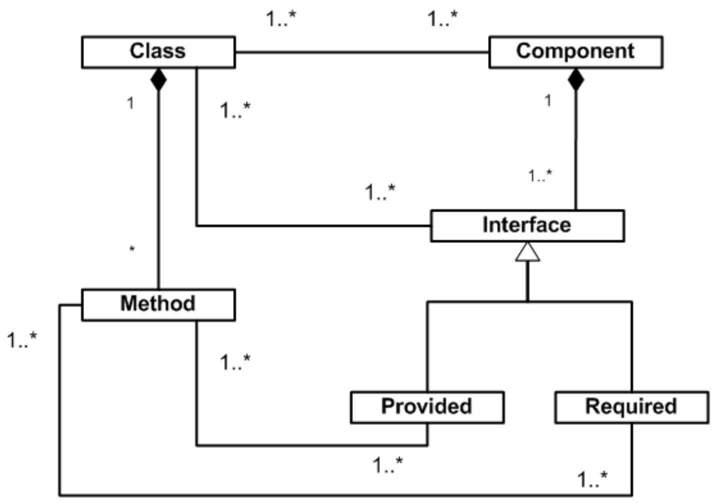


Figure 3.1: Object-to-component mapping model

3.2.2 Approach Principles and Process

The proposed approach aims at mining reusable software components from a collection of similar object-oriented software product variants. This is done by statically analyzing the commonality and the variability between components composing the products. Thus we identify components composing each product variant. Since that an object-oriented class may contribute to implement different functionalities by participating with different sets of object-oriented classes, we consider that any set of classes could form a potential component if and only if it has an accepted quality function value, following such a component quality model. Due to the similarity between the product variants, their potential components may provide similar functionalities. Thus we identify similar components among all potential ones. Similar components are those providing mostly the same functionalities and differ compared to few others. A group of similar components is considered as variants of one component since that they provide mostly the same functionalities. Thus we extract a common component from each group of similar components. This component is considered as the most reusable one compared to the members of the analyzed group. Only classes constituting the internal structures, i.e., the implementation, of the reusable components are identified. However a component is used based on its provided and required interfaces. Thus we structure component interfaces; required and provided. The selection of a component to be reused is based on its documentation. This means that software architects go through the component documentation to decide if the component serve their needs. Thus we document the mined components by describing the services that components provide.

Based on that, we propose the following process to identify reusable components from a set of product variants (see Figure 3.2):

Identifying potential components: each software product is independently analyzed to identify all potential components. These are identified based on the evaluation of their quality characteristics.

Identifying similar components: we identify similar components among all potential ones. To this end, we cluster the components into groups based on the lexical similarity among classes composing the components.

Reusable component mining from similar potential ones: we rely on the similarity of each group of components to build a single component, which will be representative of this group; this will be considered as a reusable component. To this end, we rely on how the composing classes are distributed between the components.

Identifying component interfaces: we structure component interfaces, provided and required ones, based on the analysis of the dependencies between components in order to identify how they interact with each other.

Documentation of components: we document the minded components by providing a description of the component functionalities.

3.3 Identifying Classes Composing Reusable Components

3.3.1 Identifying Potential Components

We view a potential component as a set of object-oriented classes, where the corresponding value of the quality fitness function is satisfactory (i.e., its quality value is higher than a predefined quality threshold). Thus our analysis consists of extracting any set of object-oriented classes that can be formed as a potential component. Such that the overlapping between the components is allowed.

3.3.1.1 Potential Component Identification Method

Identifying all potential components needs to investigate all subsets of classes that can be formulated from the source code (i.e., brute force technique). Then, the

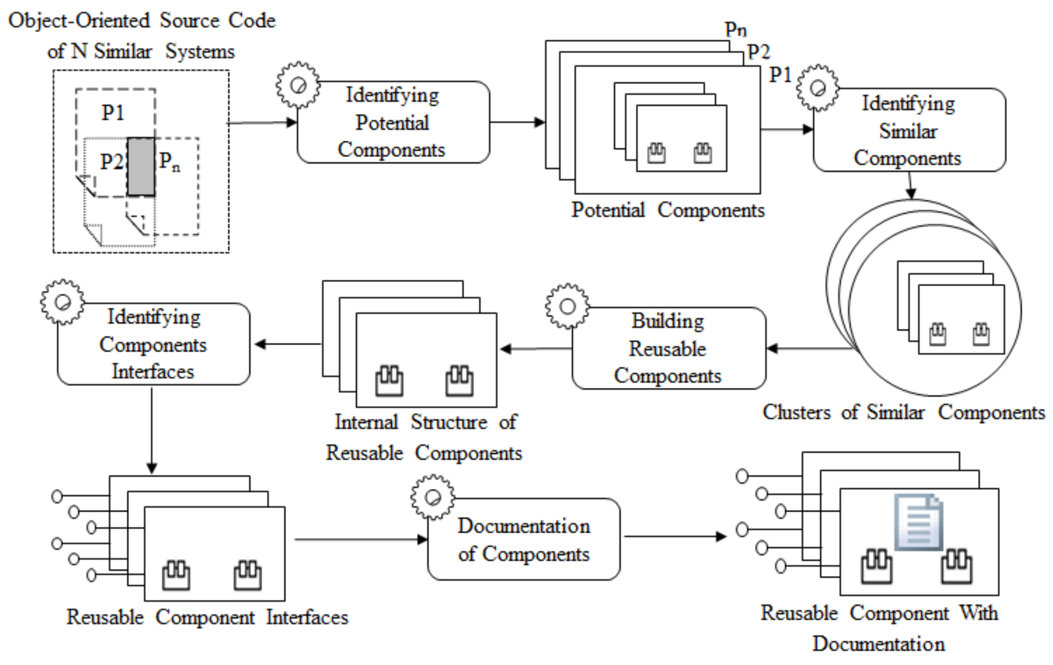

Object-Oriented Source Code of N Similar Systems
P1
P2
Pn
Identifying Potential Components
Pn
P2
P1
Potential Components
Identifying Similar Components
Clusters of Similar Components
Building Reusable Components
Internal Structure of Reusable Components
Identifying Components Interfaces
Reusable Component Interfaces
Documentation of Components
Reusable Component With Documentation

Figure 3.2: The process of reusable components mining

ones that maximize the quality fitness function are selected[2]. Nevertheless, this is considered as NP-hard problem since that the computation of all subsets requires an exponential time complexity ($O(2^n)$). Figure 3.3 presents the curve of the time complexity. To this end, we propose a heuristic-based technique that aims at mining a set of groups, such that these groups are good enough ones of the corresponding optimal groups. We consider that classes composing a potential component are gradually identified starting from a core class that participates with other classes to contribute functionalities. Thus each class of the analyzed software product can be selected to be a core one. Classes having either direct or indirect link with it are candidates to be added to the corresponding component.

3.3.1.2 Potential Component Identification Algorithm

The selection of a class to be added at each step is decided based on the quality function value obtained from the formed component. In other words, classes are ranked based on the obtained value of the quality function when it is gathered to the current group composing the component. The class obtaining the highest quality value is selected to extend the current group. We do this until all candidate classes are grouped into the component. The quality of the formed groups is evaluated at each step, i.e., each time when a new class is added. We select the peak quality value to decide which classes form the component. This means that we exclude

[2]We rely on the component quality model presented by ROMANTIC [Chardigny 2009] [Kebir 2012a] (see Section 2.4.2).

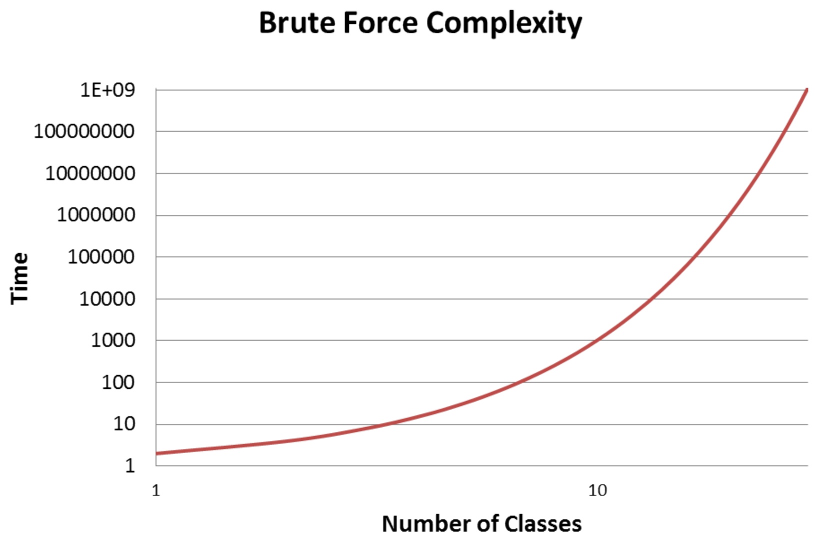


Figure 3.3: The time complexity of brute force technique

classes added after the quality function reaches the peak value since they minimize the quality of the mined component.

For example, in Figure 3.4, *Class 7* and *Class 8* are put aside from the group of classes related to *Component 2* because when they have been added the quality of the component is decreased compared to the peak value. Thus classes retained in the group are those maximizing the quality of the formed component. After identifying all potential components of such a software product, the only ones retained are components that their quality values are higher than a quality threshold that is defined by software architects. For example, in Figure 3.4, suppose that the predefined quality threshold value is *70%*. Thus *Component 1* does not reach the required threshold. Therefore it should not be retained as a potential component. This means that the starting core class is not suitable to form a component.

Algorithm 1 illustrates the process of potential component mining. In this algorithm, *Q* refers to the quality fitness function and *Q-threshold* is a predefined quality threshold.

3.3.2 Identifying Similar Components

We define similar components as a set of components providing mostly the same functionalities and differing in few ones. These can be considered as variants of the same component.

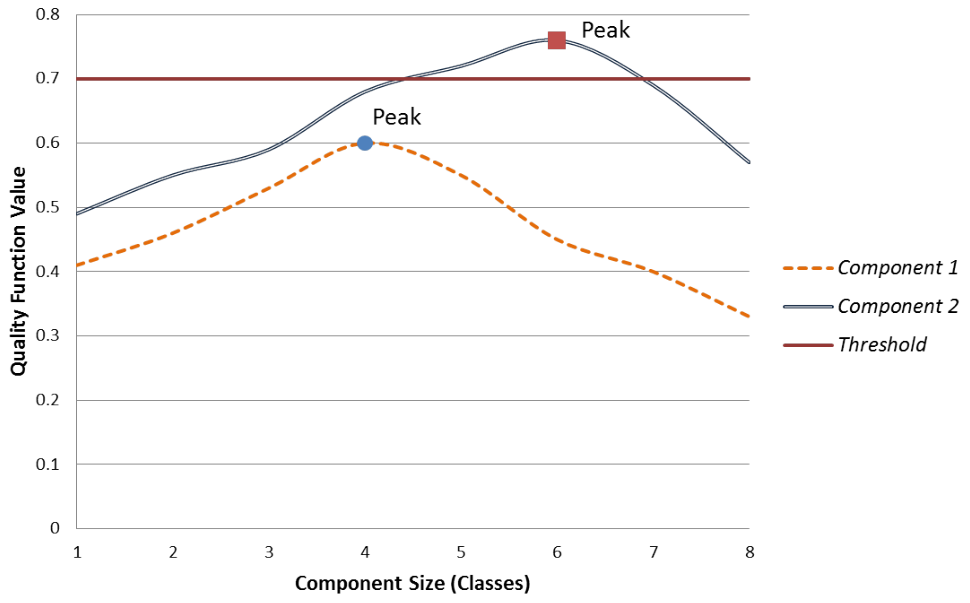


Figure 3.4: Forming potential components by incremental selection of classes

3.3.2.1 Similar Components Identification Method

Product variants are usually developed using copy-paste-modify technique. Thus, we consider that classes having similar names implement almost the same functionalities. Even if some of the composed methods are overridden, added or deleted, the main functionalities are still the same ones. Therefore the similarity, as well as the difference, between components appears compared to their internal structures composed of object-oriented classes. Thus similar components are those sharing the majority of their classes and differing considering the other ones.

Groups of similar components are built based on a lexical similarity metric. Thus components are identified as similar compared to the strength of similarity links between classes composing them. We use cosine similarity metric [Han 2006]. Following this metric each component is considered as a text document, which consists of a list of component classes' names. The similarity between a set of components is calculated based on the ration between the number of shared classes to the total number of distinguished classes.

3.3.2.2 Similar Components Identification Algorithm

We use a hierarchical clustering algorithm to gather similar components into groups. The algorithm consists of two steps. The first one aims at building a binary tree, called dendrogram. This dendrogram provides a set of candidate clusters by presenting a hierarchical representation of component similarity. Figure 3.5 shows an example of a dendrogram tree, where C_i refers to $Component_i$. The second step

Algorithm 1: Identifying Potential Components

```
Input: Object-Oriented Source Code($OO$)
Output: A Set of Potential Components($PC$)
$classes$ = extractInformation($OO$);
for each c in classes do
    $component = c$;
    $candidateClasses = classes$.getConnectedClasses($c$);
    $bestComponent = component$;
    while (|candidateClasses| >= 1) do
        $c1$ = getNearestClass($component$, $candidateClasses$);
        $component = component + c1$;
        $candidateClasses = candidateClasses$ - $c1$;
        if Q(component)) > Q(bestComponent) then
            $bestComponent = component$;
        end
    end
    if Q(bestComponent) > $Q - threshold$ then
        $PC = PC + bestComponent$;
    end
end
return $PC$
```

Algorithm 2: Building Dendrogram

```
Input: Potential Components($PC$)
Output: Dendrogram Tree ($dendrogram$)
BinaryTree $dendrogram = PC$;
while (|dendrogram| > 1) do
    $c1, c2$ = mostLexicallySimilarNodes($dendrogram$);
    $c$ = newNode($c1, c2$);
    remove($c1, dendrogram$);
    remove($c2, dendrogram$);
    add($c, dendrogram$);
end
return $dendrogram$
```

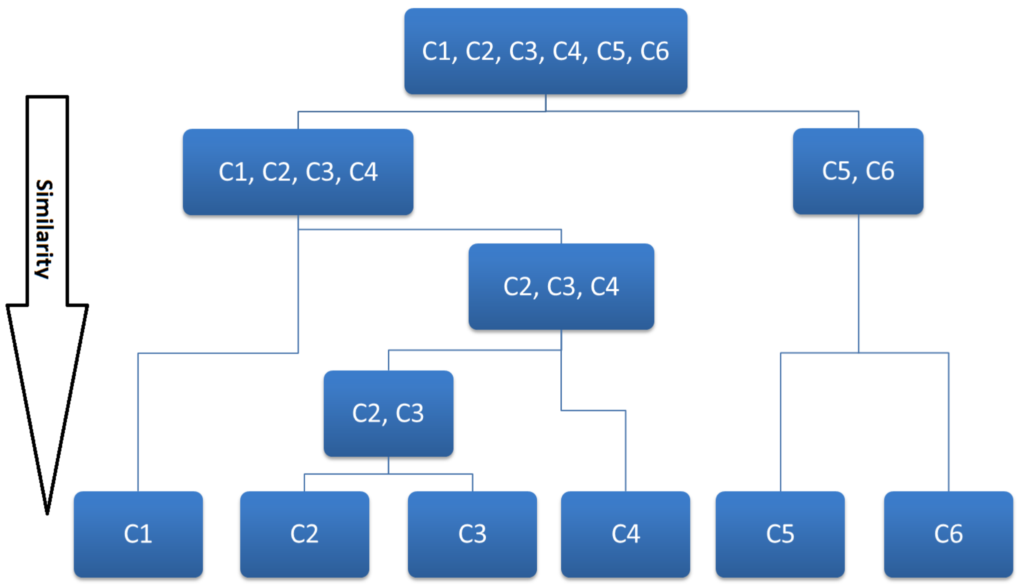


Figure 3.5: An example of a dendrogram tree

aims at traveling through the built dendrogram, in order to extract the best clusters, representing a partition.

To build a dendrogram, the algorithm starts by considering individual components as initial leaf nodes in a binary tree. Next, the two most similar nodes are grouped into a new one, i.e., as a parent of them. For example, in Figure 3.5, the C_2 and C_3 are grouped. This is continued until all nodes are grouped in the root of the dendrogram. Algorithm 2 presents the procedure used to gather similar components onto a dendrogram. It takes a set of potential components as an input. The result of this algorithm is a hierarchical tree representation of candidate clusters.

To identify the best clusters, a depth first search algorithm is used to travel through the dendrogram tree. It starts from the tree root to find the cut-off points. It compares the similarity of the current node with its children. If the current node has a similarity value exceeding the average similarity value of its children, then the cut-off point is in the current node where the children minimize the quality function value. Otherwise, the algorithm continues through its children. Algorithm 3 presents the procedure used to extract clusters of components from a dendrogram. The results of this algorithm are clusters, where each one groups a set of similar components.

3.3.3 Reusable Component Mining from Similar Potential Ones

As previously mentioned, similar components are considered as variants of a common one. Thus, from each cluster of similar components, we extract a common component which is considered as the most reusable compared to the members of the analyzed group.

Algorithm 3: Dendrogram Traversal

```
Input: Dendrogram Tree($dendrogram$)
Output: A Set of Clusters of Potential Components($clusters$)
Stack $traversal$;
$traversal$.push($dendrogram$.getRoot());
while (! $traversal$.isEmpty()) do
    Node $father = traversal$.pop();
    Node $left = dendrogram$.getLeftSon($father$);
    Node $right = dendrogram$.getRightSon($father$);
    if similarity($father$) > (similarity($left$) + similarity($right$) / 2) then
        $clusters$.add($father$)
    else
        $traversal$.push($left$);
        $traversal$.push($right$);
    end
end
return $clusters$
```

3.3.3.1 Method to Identify Reusable Component Based on Similar Ones

Classes composing similar components are classified into two types. The first one consists of classes that are shared by these components. We call these classes as *Shared* classes. In Figure 3.6, *C3, C4, C8* and *C9* are examples of *Shared* classes. The second type is composed of other classes that are diversified between the components. These are called as *Non-Shared* classes. *C1, C2* and *C10* are examples of *Non-Shared* classes in the cluster presented in Figure 3.6.

Since that *Shared* classes are identified in several products to be part of one component, we consider that *Shared* classes form the core of the reusable component. Thus *C3, C4, C8* and *C9* should be included in the component identified from the cluster presented in Figure 3.6. However, these classes may not form a correct component following our quality measurement model. Thus some *Non-Shared* classes need to be added to the reusable component, in order to keep the component quality high. The selection of a *Non-Shared* class to be included in the component is based on the following criteria:

- The quality of the component obtained by adding a *Non-Shared* class to the core ones. This criterion aims at increasing the component quality. Therefore classes maximizing the quality function value are more preferable to be added to the component.

- The density of a *Non-Shared* class in a cluster of similar components. This refers to the occurrence ratio of the class compared to the components of this group. It is calculated based on the number of components including the class to the total number of components composing the cluster. We consider that a class having a high density value contributes to build a reusable component since it keep the component belonging to a larger number of products. For example, in Figure 3.6, the densities of *C2* and *C1* are respectively *66% (2/3)* and *33% (1/3)*. Thus *C2* is more preferable to be included in the component than *C1*, since that *C2* keeps the reusable component belonging to two products, while *C1* keeps it belonging only to one product.

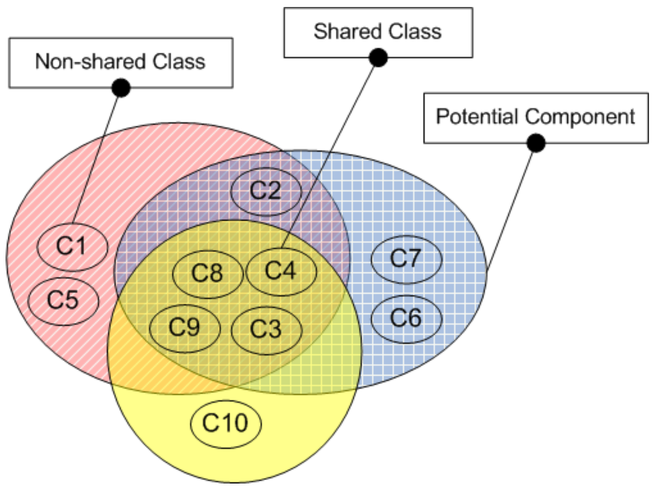


Figure 3.6: An example of a cluster of similar components

3.3.3.2 Algorithm Providing Optimal Solution for Reusable Component Identification

Based on the method given in the previous section, an optimal solution which considers the two criteria of *Non-Shared* class selection can be given through the following algorithm. First, for each cluster of similar components, we extract all candidate subsets of classes among the set of *Non-Shared* ones. Then, the subsets that reach a predefined density threshold are only selected. The density of a subset is the average densities of all classes in this subset. Next, we evaluate the quality of the component formed by grouping core classes with classes of each subset resulting from the previous step. Thus the subset maximizing the quality value is grouped with the core classes to form the reusable component. Only components with a quality value higher than a predefined threshold are retained. Algorithm 4 shows this procedure, such that *Q* refers to the component quality fitness function of *ROMANTIC*, *Q-threshold* refers to the predefined quality threshold and *D-threshold* refers to the predefined density threshold.

Nevertheless the above algorithm is NP-complete problem (i.e., the complexity of identifying all subsets of a collection of classes is $O(2^n)$). This means that the computing time will be accepted only for components with a small number of *Non-Shared* classes. This algorithm is not scalable for a large number of *Non-Shared* classes. For example, *10 Non-Shared* classes need *1024* computational operations, while *20* classes need *1048576* computational operations.

Algorithm 4: Optimal Solution for Mining Reusable Components

```
Input: Clusters of Components($clusters$)
Output: A Set of Reusable Components($RC$)
for each $cluster \in clusters$ do
    $shared = cluster$.getFirstComponent().getClasses;
    $allClasses = \emptyset$;
    for each $component \in cluster$ do
        $shared = shared \cap component$.getClasses();
        $allClasses = allClasses \cup component$.getClasses();
    end
    $nonShared = allClasses - shared$;
    $allSubsets$ = generateAllsubsets($nonShared$);
    $reusableComponent = shared$;
    $bestComp = reusableComponent$;
    for each $subset \in allSubsets$ do
        if Density(subset)$> D - threshold$ then
            if Q($reusableComponent \cup subset$)) $>$ Q(bestComp) then
                $bestComp = reusableComponent \cup subset$;
            end
        end
    end
    if Q(bestComp) $>= Q - threshold$ then
        add($RC$,$bestComp$);
    end
end
return $RC$
```

3.3.3.3 Algorithm Providing Near-Optimal Solution for Reusable Component Identification

As a consequence of the complexity of the algorithm given in the previous section (the optimal result algorithm), we defined an heuristic algorithm as an alternative. This algorithm is as follows. First of all, *Non-Shared* classes are evaluated based on their density. The Classes that do not reach a predefined density threshold are rejected. Then, we identify the greater subset that reaches a predefined quality threshold when it is added to the core classes. To identify the greater subset, we consider the set composed of all *Non-Shared* classes as the initial one. This subset is grouped with the core classes to form a component. If this component reaches the predefined quality threshold, then it represents the reusable component. Otherwise, we remove the *Non-Shared* class having the lesser quality value compared to the quality of the component formed when this class is added to the core ones. We do this until a component reaching the quality threshold or the subset of *Non-Shared* classes becomes empty. Algorithm 5 shows the process of reusable component mining, where *Q* refers to the component quality fitness function of *ROMANTIC*, *Q-threshold* refers to the predefined quality threshold and *D-threshold* refers to the predefined density threshold.

Algorithm 5: Near-Optimal Solution for Mining Reusable Components

```
Input: Clusters of Components($clusters$)
Output: A Set of Reusable Components($RC$)
for each cluster $\in$ clusters do
    $shared = cluster$.getFirstComponent().getClasses;
    $allClasses = \emptyset$;
    for each component $\in$ cluster do
        $shared = shared \cap component$.getClasses();
        $allClasses = allClasses \cup component$.getClasses();
    end
    $nonShared = allClasses - shared$;
    $reusableComponent = shared$;
    for each class $\in$ nonShared do
        if $Density(class) < D - threshold$ then
            $nonShared = nonShared$ - $class$;
        end
    end
    while $(|nonShare| > 0)$ do
        if $Q(reusableComponent \cup nonShare) >= Q - threshold$ then
            add($RC$,$reusableComponent$);
            break;
        else
            removeLessQualityClass($nonShare$, $shared$);
        end
    end
end
return $RC$
```

3.4 Identifying Component Interfaces

A component is used based on its provided and required interfaces. For object-oriented components, the interaction between the components is realized through method calls. In other words, a component provides its services through a set of methods that can be called by the other ones, which requires services of this component. Thus provided interfaces are composed of a set of public methods that are implemented by classes composing the component. In the other hand, required interfaces are composed of methods that are used by the other components (i.e., the provided interfaces of the other components). The identification of component interfaces is based on grouping a set of object-oriented methods into a set of component interfaces. We rely on the following heuristics to identify these interfaces:

Object-oriented interface: in object-oriented design, methods manipulating functionalities of the same object family are implemented by the same object-oriented interface. Thus we consider that a group of methods belonging to the same object-oriented interface has a high probability to belong to the same component interface. We propose a function, called $SI()$, that aims at measuring how much a set of methods M belongs to the same interface. This function returns the size of the greatest subset of M which consists of methods that belong to the same object oriented interface divided by the size of M. Algorithm 6 shows the procedure of $SI()$ function.

Method cohesion: methods access (read/write) the same set of attributes to participate to provide the same services. Thus cohesive methods have more probability to belong the same component interface than those that are not. To measure how much a set of methods is cohesive, we use LCC metric [Bieman 1995] since that it measures direct and indirect dependencies between methods.

Method lexical similarity: the lexical similarity of methods probably indicates to similar implemented services. Therefore methods having a lexical similarity likely belong to the same interface. To this end, we utilize *Conceptual Coupling* metric [Poshyvanyk 2006] to measure methods lexical similarity based on the semantic information obtained from the source code, encoded in identifiers and comments.

Correlation of usage: when a component provides services for another component, it provides them through the same interface. Thus methods that have got called together by the other components are likely to belong to the same interface. To this end, we propose a function *CU()* which measures how much a set of methods M has been called together by the same component. Algorithm 7 shows the precedure of calculating the value of correlation of usage for a set of methods.

According to these heuristics, we define a fitness function used to measure the quality of a group of methods M to form a component interface. Where λ_i are weight values, situated in [0-1]. These are used by the architect to weight each characteristic as needed.

$$Interface(M) = \frac{1}{\sum_i \lambda_i} \cdot (\lambda_1 \cdot SI(M) + \lambda_2 \cdot LCC(M) + \lambda_3 \cdot CS(M) + \lambda_4 \cdot CU(M)) \quad (3.1)$$

Based on this fitness function, we use a hierarchical clustering algorithm to partition a set of public methods into a set of clusters, where each cluster is considered as a component interface.

Algorithm 6: Same Object-Oriented Interface (SI)

```
**Input**: A Set of Methods($M$), a Set of Object-Oriented Interfaces($OOI$)
**Output**: Same Object-Oriented Interface Value ($SI$)
$sizeGreatest = |M \cap OOI$.getFirstInterface().getMethods()$|$;
for each $interface \in OOI$ do
    if $|M \cap interface.getMethods()| > sizeGreatest$ then
        $sizeGreatest = |M \cap interface$.getMethods()$|$;
    end
end
$SI = sizeGreatest \ / \ M$.size();
return $SI$
```

Algorithm 7: Correlation of Usage (CU)

```
**Input**: A Set of Methods($M$), a Set of Components($Com$)
**Output**: Correlation of Usage Value($CU$)
$sizeGreatest = |M \cap Com$.getFirstComponent().getCalledMethods()$|$;
for each $component \in Com$ do
    if $|M \cap component.getCalledMethods()| > sizeGreatest$ then
        $sizeGreatest = |M \cap interface$.getCalledMethods()$|$;
    end
end
$CU = sizeGreatest \ / \ M$.size();
return $CU$
```

3.5 Documentation of Components

The documentation of a component helps the developers to find a component that meets their needs, instead of going through its implementation. The description of the component functionalities forms an important part of its documentation. Thus we propose to identify for each mined component its main functionalities. We do this based on two steps: the identification of the component functionalities and the generation of a description for each of them. These steps are detailed below.

3.5.1 Identifying Component Functionalities

The identification of functionalities provided by a component is based on partitioning the component into a set of sub-components. Such that each sub-component provides a specific functionality. The partitioning is based on the following three properties:

- **Number of provided interfaces.** Different component interfaces may provide different functionalities. Thus as much as the number of provided interfaces is increased, the number of functionalities is increased.

- **Cohesion of classes.** Classes providing the same functionalities must be cohesive. Thus as much as a group of classes is cohesive, they are likely participate to the same specific functionality.

- **Classes coupling.** Classes participating in the same functionality must have a low coupling with other parts in the component.

These properties refer to the specificity of a component defined by *ROMANTIC* quality model. Thus, we use Equation 2.2 as a fitness function in a hierarchical clustering algorithm, in order to decompose component classes into partitions, where each one represents one of the functionality of the analyzed component. This function is as follows:

$$S(E) = \frac{1}{5} \cdot (\frac{1}{|I|} \cdot \sum_{i \in I} LCC(i) + LCC(I) + LCC(E) + Couple(E) + noPub(I)) \quad (3.2)$$

3.5.2 Generation of the Functionality Description

In the previous step, the component classes are partitioned according to their functionalities. In this step, we present how a description of each partition (i.e., functionality) is generated. We consider that a description consists of words that are frequently used to form class names of a partition. In object-oriented languages, a class name is often composed of a set of nouns concatenated by the camel-case notation. These nouns represent a meaningful name for the main purpose of the class. Usually, the first noun in a class name holds the main goal of the class, and so on. Accordingly, we propose the following three steps. Firstly, tokens are extracted by separating the words which form the class names according to the camel-case syntax. For example, *MediaControllerAlbum* is divided into *Media*, *Controller* and *Album*). Secondly, a weight is affected to each extracted token. The tokens which are the first word of a class name are given a large weight (*100%*). Other tokens are given a small weight, such that *75%* for the second word, *50%* for the third word and *25%* for the fourth word. The weight of each token is calculated as follows:

$$Weight(w) = \frac{1}{\sum_i N_i} \cdot (1 \cdot N_1 + 0.75 \cdot N_2 + 0.50 \cdot N_3 + 0.25 \cdot N_4) \quad (3.3)$$

Such that:

- W: refers to a word.

- N_i refers to the number of occurrence of the word W in the position i.

Lastly, we use tokens which have the highest weight to construct the functionality description in an orderly manner. Meaning, the token that has the highest weight will become the first word of the functionality description and so on. Software architects define the number of words as needed.

3.6 Experimental Results and Evaluation

3.6.1 Data Collection

We collect two sets of product variants. These are Mobile Media [3] [Figueiredo 2008] as a small-scale software, and ArgoUML [4] [Couto 2011] as a large-scale one. Mobile Media variants manipulate music, video and photo on mobile phones. They were developed starting from the core implementation of Mobile Media. Then, the other features are added incrementally for each variant. Using the latest version, the user can generate 200 variants. In our experimentation, we considered *8* variants, where each variant contains *43.25* classes on average. ArgoUML [Couto 2011] is a UML modeling tool. It is developed as a software product line. We applied our approach on *9* variants, where each variant is generated by changing a set of the needed features. Each variant contains *2198.11* classes on average.

3.6.2 Evaluation Method and Validation

The proposed approach is applied on the collected product variants. Firstly, we extract a set of potential components from each product variant independently. Then, we identify similar components among ones identified from all products. This is done by using a hierarchical clustering algorithm. Next, a reusable component is identified by analyzing each group of potential ones. In order to validate our approach, we evaluate the reusability of components that are mined based on our approach.

3.6.3 Results

3.6.3.1 Identifying Potential Components

In order to consider a group of classes forming a potential component, its quality function value should exceed a predefined quality threshold. Indeed the selection of this threshold effects on both the quality and the number of the mined components. To help software architects selecting a proper threshold value, we assign the quality threshold values situated in *[0%, 100%]*. We start from a value *0* and then it is incremented by *5%* at each run. The results obtained from Mobile Media and ArgoUML are respectively shown in Figure 3.7 and Figure 3.8, where the value of the threshold is on the X-axis, and the average number of the mined components in a variant is on the Y-axis.

The results show that the number of the mind components is lower than the number of classes composing the products, for low threshold values. The reason behind that is the fact that some of the investigated classes produce the same component. For example, *InvalidPhotoAlbumNameException* and *InvalidImageFormatException* produce the same component, when they are considered as the core for mining a

[3]Available at http://homepages.dcc.ufmg.br/ figueiredo/spl/icse08

[4]Available at http://argouml-spl.tigris.org/

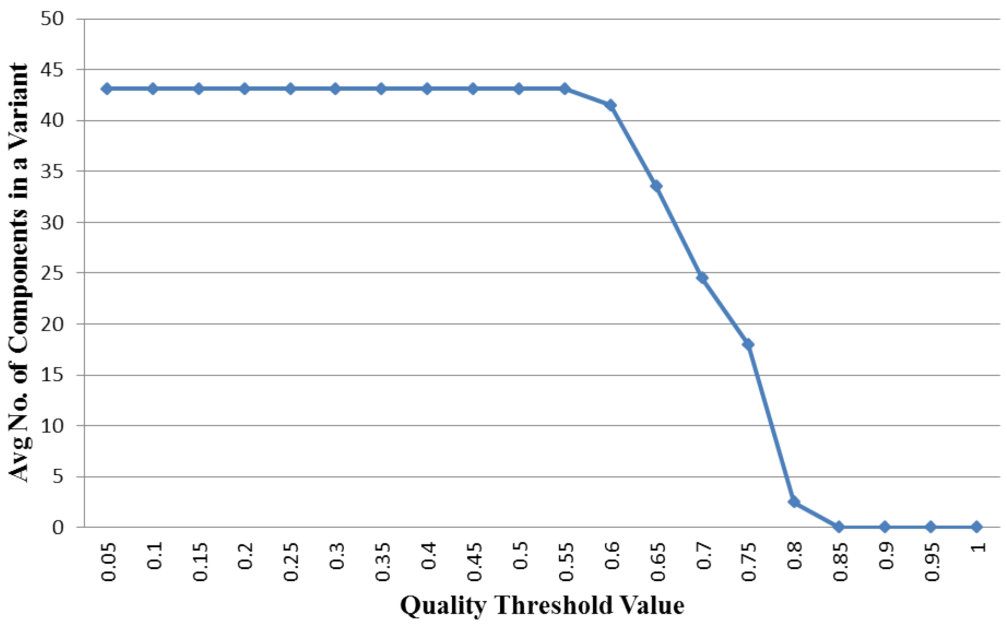


Figure 3.7: Changing threshold value to extract potential components from Mobile Media

potential component. Moreover, the results show that the number of the mined components is fixed as well as the quality threshold value is situated in *[5%, 55%]*. This means that selecting a value in this interval do not represent good choice, since that it does not make a distinction between components having diverse quality values.

In our study, to have a harmony between the number of potential components and the number of classes composing the products, we assign *70%* and *83%* as threshold values respectively for Mobile Media and ArgoUML case studies. Table 3.1 shows the detail results obtained based on these threshold values. It presents the total number of potential components *(TNOPC)* mined based on the analysis of all variants, the average number of classes (size) of these components *(ASOC)*, the average value of the specificity characteristic *(AS)*, the average value of the autonomy characteristic *(AA)* and the average value of the composability characteristic *(AC)*.

Figure 3.9 presents an example of a potential component extracted from ArgoUML. This component is identified by considering *GoClassToNavigableClass* as the core class. The quality fitness function reaches the peak value as well as the *18*th classes is added. Thus classes added after the *18*th one are rejected since that they should be belonged to other components and the dependencies with this component should be realized by component interfaces.

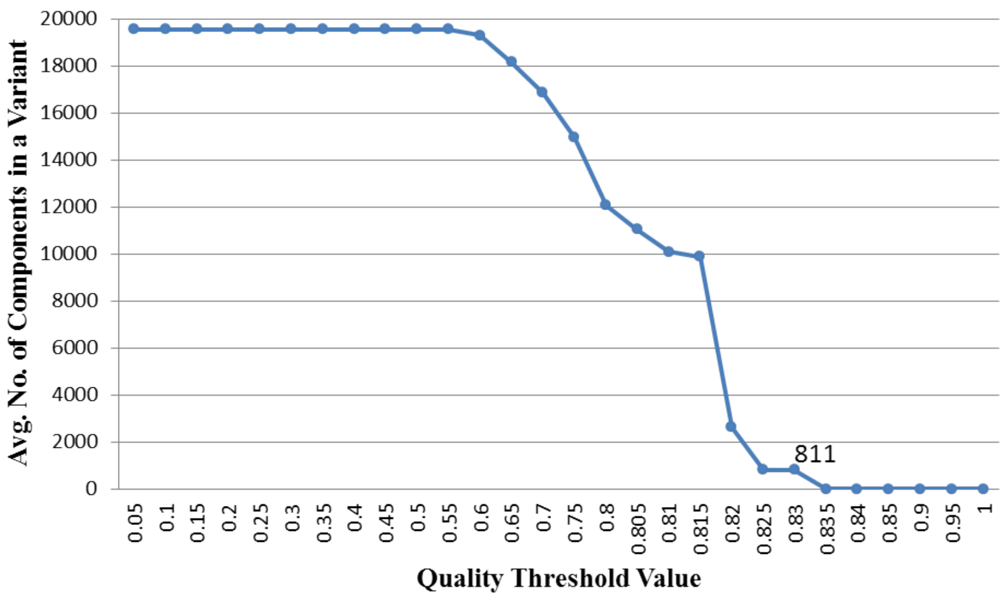


Figure 3.8: Changing threshold value to extract potential components from ArgoUML

Table 3.1: The results of potential components extraction

Family Name	TNOPC	ASOC	AS	AA	AC
Mobile Media	24.50	6.45	0.56	0.71	0.83
ArgoUML	811	11.38	0.64	0.83	0.89

3.6.3.2 Identifying Similar Components

Table 3.2 presents the results of the process of grouping similar potential components into clusters. For each case study, Table 3.2 shows the number of clusters *(NOC)*, the average number of components in the identified clusters *(ANOC)*, the average number of *Shared* classes in these clusters *(ANSC)*, the average value of the specificity characteristic *(ASS)*, the average value of the autonomy characteristic *(AAS)*, and the average value of composability characteristic of the *Shared* classes *(ACS)* in these clusters. The results show that product variants sharing a bunch of similar components. For instance, each variant of Mobile Media has *24.5* components in average. These components are grouped into *42* clusters. This means that each variant shares *5.38* components with the other variants, in average. Thus a reusable component can be identified from these components. In the same way, ArgoUML variants share *5.26* components. Table 3.3 shows an example of a cluster of similar components identified from ArgoUML case study, where *X* refers to that a class is a member in the corresponding product variant. In this example, we note that the components have *5 Shared* classes. These classes have been identified to be

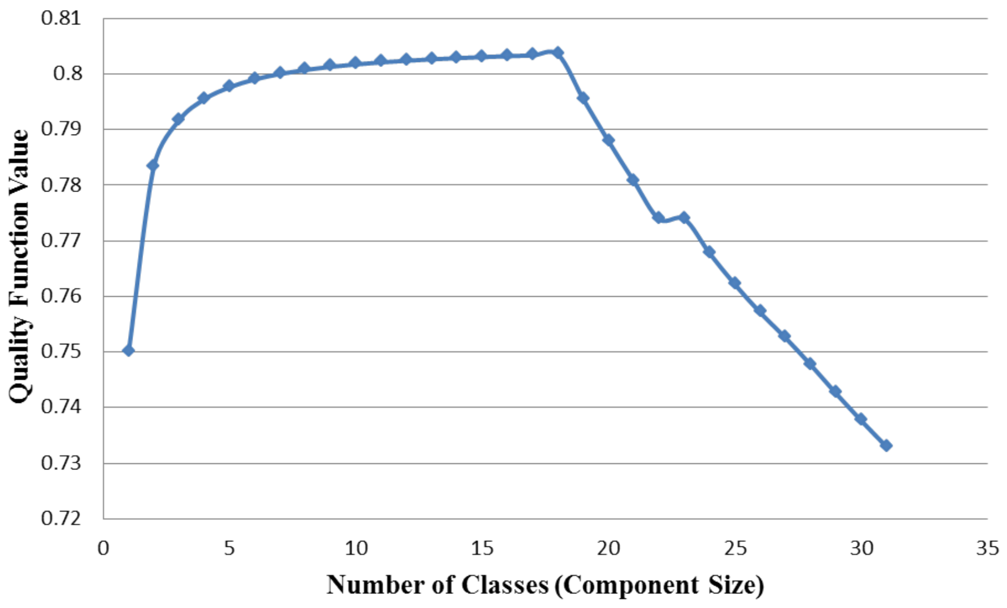


Figure 3.9: An instance of a potential component extracted from ArgoUML

part of the same component in *9* variants of ArgoUML. Thus they can be considered as core classes to form a reusable component that is reused in the *9* variants.

Table 3.2: The results of component clustering

Family Name	**NOC**	**ANOC**	**ANSC**	**ASS**	**AAS**	**ACS**
Mobile Media	42	5.38	5.04	0.59	0.71	0.89
ArgoUML	325	5.26	8.67	0.57	0.87	0.93

3.6.3.3 Reusable Component Mining from Similar Potential Ones

Table 3.4 summarizes the final set of reusable components mined using our approach. Based on our experimentation, we assign *50%* to the density threshold value. For each product, we present the number of the mined components *(NOMC)*, the average component size *(ACS)*, and the average value of the specificity *(AS)*, the autonomy *(AA)*, and the composability *(AC)* of the mined components. The results show that some of the identified clusters do not produce reusable components. For instance, in Mobile Media, the *42* clusters produce only *39* components. This means that three of the clusters are not able to form reusable components. The reason behind that is one of the following two situations. The first one is that the selection of threshold density causes to remove classes that are important to constitute the component, and hence, the component was rejected because it did not exceed the quality threshold value. The second one is that the produced component is already identified from

Table 3.3: An instance of a cluster of similar potential components

Variant No. / **Class Name**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
StreamSource	X				X	X	X		X
ArgoEventTypes	X	X	X	X	X	X	X	X	X
JWindow	X	X	X	X	X	X	X	X	X
TabFigTarget	X	X	X	X	X	X	X	X	X
SortedListModel	X		X		X	X	X		
BooleanSelection2		X	X	X				X	
FileConstants	X	X	X	X	X	X	X	X	X
OclAPIModelInterpreter	X	X	X	X	X	X	X	X	X

another cluster, thus the component is removed to avoid the redundancy.

Table 3.4: The final set of mined components

Family Name	**NOC**	**ACS**	**AS**	**AA**	**AC**
Mobile Media	39	5.61	0.58	0.74	0.90
ArgoUML	324	9.77	0.61	0.84	0.84

Table 3.5 shows examples of a set of reusable components that are mined based on the analysis of Mobile Media. Where, *DOF* refers to the description of the functionalities provided by the considered component, *NOV* refers to the number of variants that contain the component, *NOC* represents the number of classes that form the component. *S, A* and *C* respectively represent the specificity, the autonomy, and the composability of each component. As it is shown in Table 3.5, the second component provides two functionalities, which are *Add Constants Photo Album*, and *Count Software Splash Down Screen*. The former one deals with adding a photo to an album. The letter is dedicated to the splash screen service.

Table 3.5: Some components

DOF	**NOV**	**NOC**	**S**	**A**	**C**
New Constants Screen Album Image	6	6	0.59	0.75	0.94
Add Constants Photo Album	8	10	0.57	0.75	0.89
Count Software Splash Down Screen					
Base Image Constants Album Screen Accessor List	6	9	0.67	0.50	0.85
Controller Image Interface Thread					

3.6.4 Validation

In order to validate the reusability of components that are mined based on our approach, we compare their reusability with ones that are mined from singular

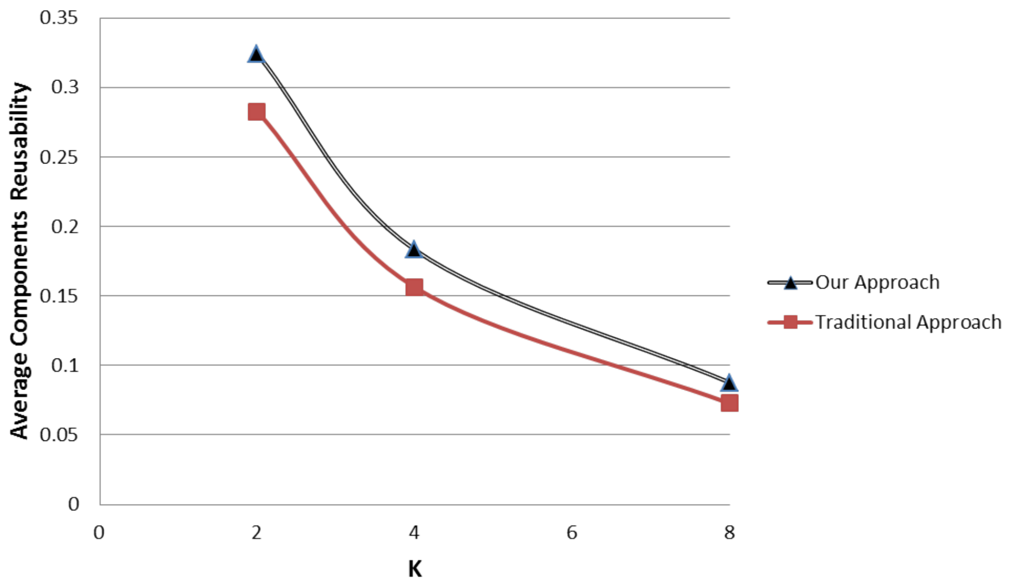


Figure 3.10: The results of reusability validation of Mobile Media components

software. We consider that the reusability of a component is evaluated based on the number of software products that the component can be reused in. For a collection of software products, the reusability is calculated as the ratio between the number of products that can reuse the component to the total number of products. A component can be reused in a product if it provides functionalities required by the product. In other words, we analyze the software functionalities, and then we check if a component provides some of these functionalities. The functionalities required by a product are identified based on potential components extracted from the product.

To prove that our validation can be generalized for other independent product variants, we depend on *K-fold* cross validation method [Han 2006]. In data mining, it is widely used to validate the results of a mining model. The main idea is to evaluate the model using an independent data set. Thus K-fold divides the data set into two parts: train data, and test data. On the one hand, train data are used to learn the mining model. On the other hand, test data are then used to validate the mining model. To do so, K-fold divides the data set into K parts. The validation is applied K times by considering K-1 parts as train data and the other one as test data. After that, the validation result is the average of all K trails. Accordingly, we validate our approach by dividing the product variants into *K* parts. Then, we only mine components from the train variants (i.e., *K-1* parts). Next, we validate the reusability of these components in the test product variants. We evaluate the result by assigning *2, 4* and *8* to the *K* at each run of the validation. The results obtained from Mobile Media and ArgoUML case studies are respectively presented in Figure 3.10 and Figure 3.11. These results show that the reusability of the components

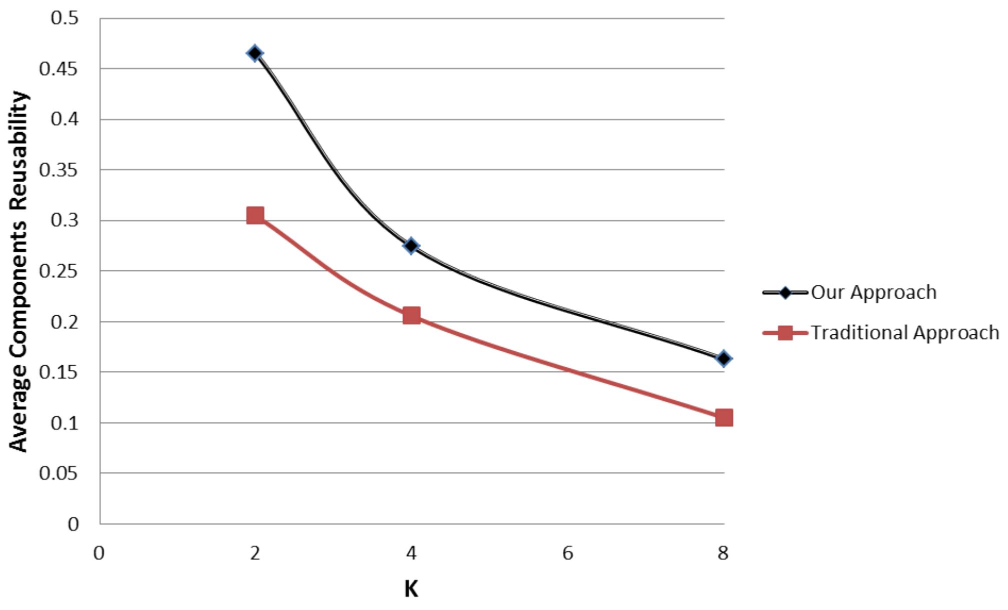


Figure 3.11: The results of reusability validation of ArgoUML components

which is mined from a collection of similar software is better than the reusability of components which is mined from singular software. We note that the reusability is decreased when the number of *K* is increased. The reason is that the number of test variants is decreased. For example, there is only one test variant when *K=8*. The slight difference between the reusability results comes from the nature of our case studies, where these case studies are very similar. Consequently, the resulting components are closely similar. In other words, there are many groups of similar components containing exactly the same classes. This yields a reusable component that is identical to cluster components. Therefore, the reusability has the same value for all of these components. However, our approach remains outperforming the traditional identification approaches.

3.7 Threats to Validity

The presented approach is concerned by two types of threats to validity. These are internal and external.

3.7.1 Threats to Internal Validity

There are five aspects to be considered regarding the internal validity. These are as follows:

1. We select a static analysis technique to identify dependencies between the

classes. However this analysis affects our results by two axes. The first one is that it does not address polymorphism and dynamic binding. However, in object-oriented, the most important dependencies are realized through method calls and access attributes. Thus the ignorance of polymorphism and dynamic binding has not a high impact on the general results of our approach. The second one is that it does not differ from the used and unused source code. This may provide a noise dependencies. However, this situation rarely exists in the case of well designed and implemented software. In contrast, dynamic analysis addresses all of these limitations. But the challenge with dynamic analysis is to identify all use cases of software.

2. We consider that the variability between software products is at the class level. This means that classes that have the same name should have almost the same implementation. While in some situations, classes may have closely similar names, but they are completely unrelated. However, in the case of product variants that is developed by copy-paste-modify technique, the modification is mainly composed of method overriding, adding or deleting, but the main functionalities are still the same ones. By the way, clone detection technique can be used to improve the identification of this similarity, which will be one of our future extensions.

3. Forming a component by adding a *Non-Shared* class to the core ones may cause a dead code (i.e., a piece of code which is executed but there is no need for its result). However this does not have an impact of the reusability, and the functionality as well, of the identified components (the goal of the approach).

4. We use a cluster algorithm to group similar components. However this provides a near optimal solution of the partitioning. Other grouping techniques may provide more accurate solutions, such as search-based algorithms. This will be a future extension of our approach.

5. Due to the lack of models that measure the reusability of object-oriented components, we propose our own empirical measurement to validate the reusability of the mined components. This can threat the reusability validation results.

3.7.2 Threats to External Validity

There are two aspects to be considered regarding the external validity. These are as follows:

1. The approach is experimented via product variants that are implemented by *Java* programming language. Since all object-oriented languages (e.g., *C++* and *C#*) have the same structure (i.e., classes) as well as the same type of dependencies (e.g., method calls and access attributes), they will provide the same results, on average.

2. Only two case studies have been collected in the experimentation (Mobile Media and ArgoUML). However these are used in several research papers that address the problem of migrating products variants into software product line. On average, the selected case studies obtained the same results. Thus these results can be generalized for other similar case studies. By the way, the approach needs to be validated with a large number of case studies. This will be a logical extension of our work.

3.8 Conclusion

In this chapter, we have presented an approach that aims at mining software components from a set of product variants. This is based on the analysis of the commonality and the variability between the products. The approach supposes that mining component based on the analysis of a set of similar software products provides more guarantee for the reusability of the mined components, compared to depending on single software product. To this end, it firstly identifies a set of potential components of each product. Then, similar components are identified to extract reusable components.

Our experimentation includes two case studies. The results show that the reusability of components that have been mined using our approach is better than the reusability of ones that have been mined based on the analysis of singular software products.

Chapter 4

Mining Software Components from Object-Oriented APIs

Contents

4.1 Introduction and Problem Analysis

In the case of object-oriented APIs, e.g., *Standard Template Libraries* in *C++* or *Java SDK*, the functionalities are implemented as object-oriented classes. It is well known that reusing and understanding large APIs, such as *Java SDK* which contains more than 7.000 classes, is not an easy task [Ma 2006] [Uddin 2012]. On the other hand, classes of an API are used following specific usage patterns, in order to provide services to software applications [Acharya 2007] [Wang 2013] [Robillard 2013]. For example, in the *android* API, *Activity*, *GroupView*, *Context*, *LayoutInflater* and *View* are the classes needed to create a simple activity which contains an empty view [Google 2015]. Classes forming a specific usage pattern are used to serve the same functionalities. Thus they are more favorable to be formed as a component.

Existing component identification approaches are designed to mine components from software applications, such as [Mishra 2009], [Kebir 2012a] and [Allier 2011]. Dependencies between classes composing software applications are only realized via structural and behavioral dependencies materialized in the source code. For example, in the case of object-oriented applications, these dependencies are realized via calls between methods, sharing types, etc.

Nevertheless reengineering object-oriented APIs into component-based ones has not been considered in the literature. In this context, we distinguish two kinds of dependencies that characterize their classes: source-code-based and usage-pattern-based ones. On the one hand, source-code-based dependencies refer to structural and behavioral dependencies (similar to software applications). On the other hand, usage-pattern-based ones are those that are revealed only by reuse scenarios: some classes need to be reused together , i.e., simultaneously, by software applications to implement a service. For example, *Activity*, and *Context* should be used together, in the android API, despite they are not structurally dependent. Usage-pattern-based dependencies can be identified by determining which classes are frequently used together. Thus these classes are more favorable to be part of the implementation of the same component. Starting from this observation, in this chapter, we propose an approach that aims at reengineering object-oriented APIs into component-based ones.

The rest of this chapter is organized as follows: Section 4.2 presents the foundations of our approach. In Section 4.3, we present the identification of component interface classes. Section 4.4 presents how APIs are organized as component-based libraries. Experimentation and results are discussed through three APIs in Section 4.5. In Section 4.6 we discuss the differences between software components and frequent usage patterns, and component identification from APIs and software applications. Threats to validity are discussed in Section 4.7. Finally, concluding remarks are presented in Section 4.8.

4.2 The Proposed Approach Foundations

The goal of our approach is at reengineering object-oriented APIs to component-based ones. This is done through two directions. The first one is the identification of groups of classes that can be considered as the object-oriented implementation of the API components. The second one is the identification of how these components can be organized as component-based APIs.

4.2.1 Component Identification

We view a component as a group of API classes that provides coarse grained services to clients of an API. The identification of this group is based on two kinds of dependencies; usage-pattern-based and source-code-based ones. Usage-pattern-based is related to the way of how software applications have used the group of classes. This refers to observations made based on the analysis of previous usages of APIs. We consider that classes frequently used together are more favorable belonging to a single or a few number of components. This is realized through Frequent Usage Patterns (FUPs) that identify recurring patterns, composed of classes frequently used together. Classes composing FUPs represent the gateways to access the API services. Thus they are used to guide the identification of classes composing the provided interfaces of components. Classes composing a FUP may be related to different services that have been used together. Thus they can be mapped to be a part of different component interfaces. Classes of a component interface can be very dependent on other classes that are not directly used by clients of the API. These are identified based on source-code-based dependencies. This means that the component identification process is driven by the identification of its provided interfaces. To this end, the analysis of structural dependencies between classes is used to identify classes forming the core of the component. This is used to form a quality-centric component. This is achieved through the three quality characteristics that should be satisfied by the group of classes forming the component; *Composability*, *Autonomy* and *Specificity*. To this end we rely on the component quality model presented by ROMANTIC [Kebir 2012a] (see Section 2.4.2).

4.2.2 API as Library of Components

We organize the API as layers of components. These layers describe how API components are vertically and horizontally organized. We consider that each layer contains components providing services to components of the layer above and requiring services from components of the layer below.

Classes constituting an API can be categorized into two types. The first one is classes that are directly reused by software applications. These represent the implementation of accessible-services of the API (offered to software applications). Thus components that are identified corresponding to these classes constitute the first layer of the API (i.e., the layer accessed by software applications). The second one is classes representing the rest of API classes. Also, these can be divided into two categories. The first category comprises of classes providing services to the first layer components. These represent the implementation of components constituting the second layer. In the same manner, components composing the other layers are identified. Based on that, we organize component-based APIs as a set of layers describing how their components are organized. Figure 4.1 shows our point of view regarding the API organization.

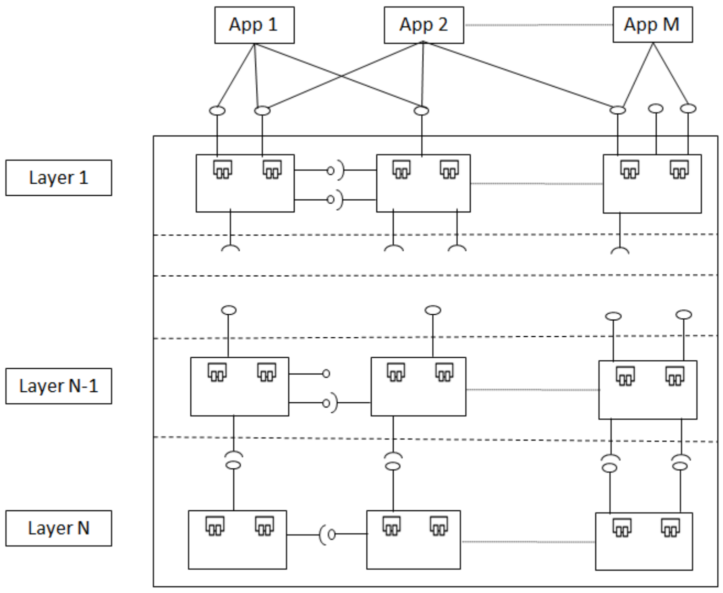


Figure 4.1: Multi-layers component-based API

4.2.3 Principles and Mapping Model

Based on the observations made in the previous sub-sections, the proposed approach can be summarized based on the the following principles:

- In object-oriented APIs, a component is identified as a group of classes.
- To reengineer the entire object-oriented API into component-based one, each class of the API is mapped to be part of at least one component. Each class is mapped either as a class of the component interfaces or as a part of the internal classes of the component.
- Classes frequently used together by software applications provide accessible-user services of the API. Thus they are used to guide the identification of classes composing the provided interfaces of components. These are identified based on FUPs.
- As a FUP can be composed of classes providing multiple services, its classes can be mapped to be a part of different component interfaces.

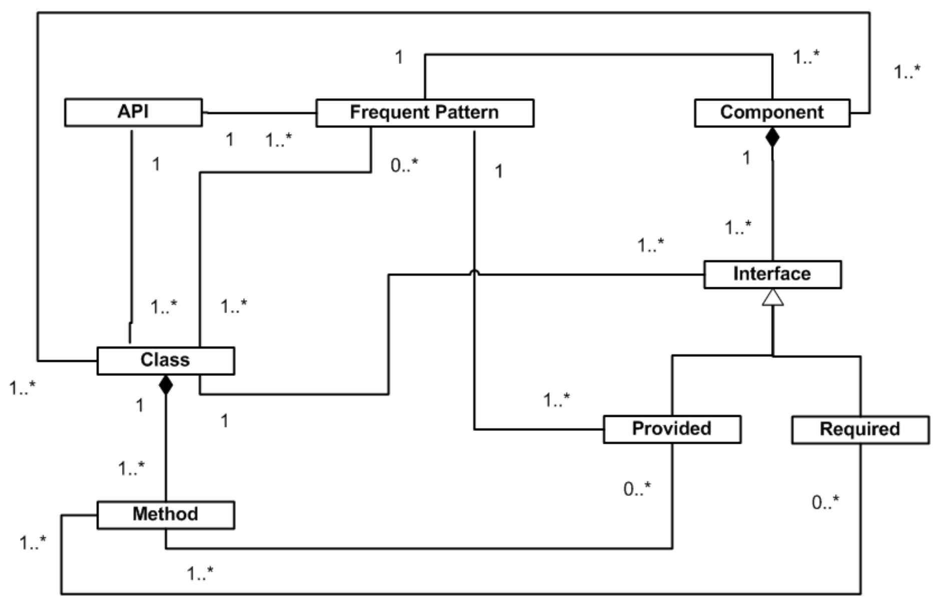


Figure 4.2: Mapping class to component through FUP

- A class of an API can be a part of several FUPs and can participate to implement multiple services. Consequently, a class can be mapped into multiple component interfaces.
- The identification of classes forming the core of the components is driven by the identification of its provided interfaces.
- The analysis of structural dependencies between classes is used to identify classes forming the core of the component.
- Classes that are not reused by software applications are used to structure components of the API layers.
- In a component-based API, the components are vertically and horizontally organized in terms of layers based on the required and provided services between the components.

Based on that, we propose a mapping model, shown in Figure 4.2, that maps class-to-component through FUPs.

4.2.4 Identification Process

We propose the following process to mine components from object-oriented APIs (see Figure 4.3):

- **Identification of frequent usage patterns.** FUPs are identified by analyzing the interactions between the API and its application clients.

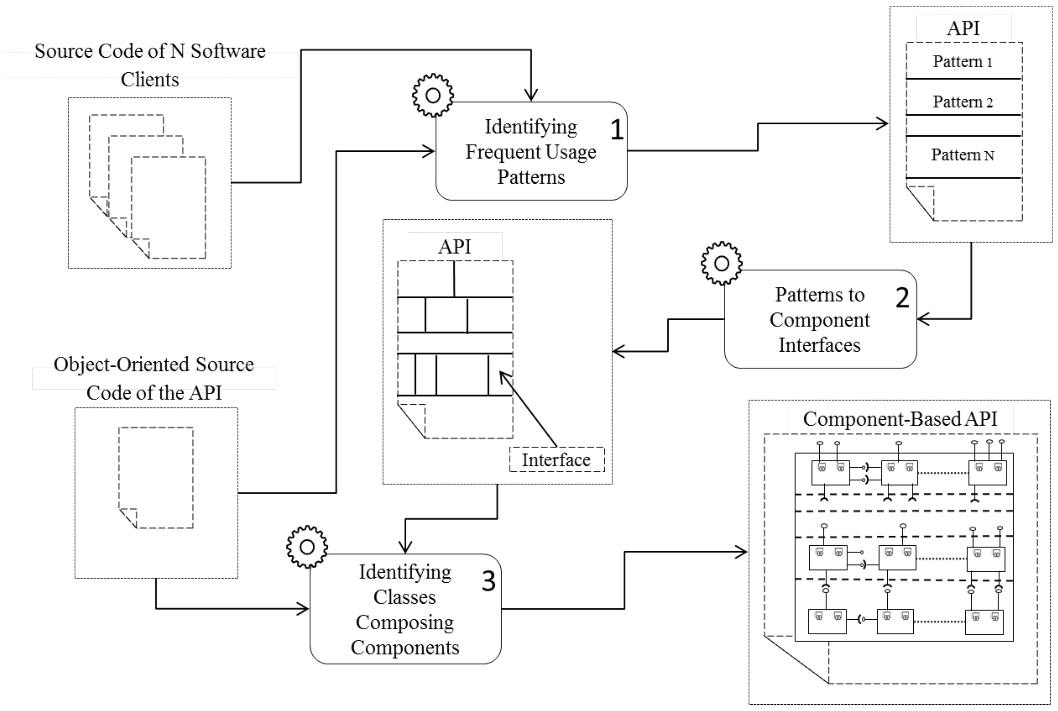

Source Code of N Software Clients
Identifying Frequent Usage Patterns
1
API
Pattern 1
Pattern 2
Pattern N
API
Patterns to Component Interfaces
2
Interface
Object-Oriented Source Code of the API
Component-Based API
Identifying Classes Composing Components
3

Figure 4.3: The process of mining components from an object-oriented API

- **Identification of the interfaces of components.** We partition the set of classes of each FUP into subgroups, where each one is considered as related to the provided interfaces of one component (c.f. Figure 4.4). The partitioning is based on criteria related to structural dependencies, lexical similarity and the frequency of simultaneous reuse.

- **Identification of internal classes of components driven by their provided interfaces.** Classes forming the provided interfaces of a component form the starting point for identifying the rest of the component classes. To identify these classes we rely on the analysis of structural dependencies between classes in the API with those forming the interfaces. We check if these classes are able to form a quality-centric component.

- **Organizing API as Layers of Components.** As each class of the API must be a part of at least one component, we associate classes that do not compose any of the already identified components to new ones. Based on that, we organize component-based APIs as a set of layers. This organization is used-driven. The first layer is composed of components that are used by the software clients, while the second layer is composed of components that provide services used by components of the first layer, and so on. As a result, the API is structured in N layers of components.

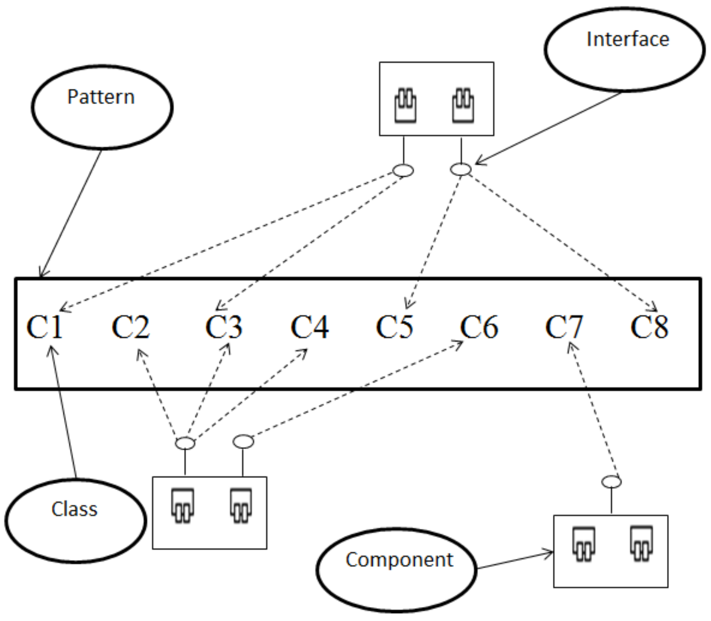


Figure 4.4: From FUP to provided interfaces

4.3 Identification of Component Interfaces

The identification of classes forming an API component is driven by the identification of classes composing the provided interfaces of this component. Classes composing these interfaces are those directly accessed by the clients of the API. Classes belonging to the same interface are those frequently used together. Therefore they are identified from frequent usage patterns. Classes of the API composing frequent usage patterns are identified based on the analysis of how API classes were used by the API clients. API classes used together constitute transactions of usage.

4.3.1 Extracting Transactions of Usage

A transaction of usage is a set of interactions between an API and a client of this API. These interactions consist of calling methods, accessing attributes, inheritance or creating an instance object based on a class of the API. They are identified by statically analyzing the source code of the API and its clients. Transactions are different depending on the choice of API clients. Therefore the choice of the API clients directly affects the type of the resulting patterns. A client can be considered either a class, a group of classes or the whole software application. Figure 4.5 illiterates the types of transactions that can be identified. On the first hand, if we consider that a transaction corresponds to an application class, *{C2, C5}*, *{C3}*, *{C5}*, *{C7}* and *{C7}* are the set of transactions identified for the first application. On the second hand, if a transaction corresponds to a group of application classes, *{C2, C5, C3}*, *{C5, C7}* and *{C7}* are the set of transactions extracted for the first application. On the other hand, if a transaction is identified based on the whole application, *{C2, C5, C3, C7}* is the one corresponding to the first application.

We consider that a group of classes related to the same application functionality reuses API classes that are related to correlated functionalities. Thus we define a client as group of classes forming a functional component in software applications. The idea behind that is to mine patterns related to functionalities composing the applications. Thus, a transaction is a set of API classes that is used by classes composing a client component. Figure 4.6 shows an example. To this end, we use ROMANTIC approach to identify client components composing software applications. Algorithm 8 shows the process of transaction identification. It starts by partitioning each software client into components. Then, for each component, it identifies API classes that are reused by the component classes.

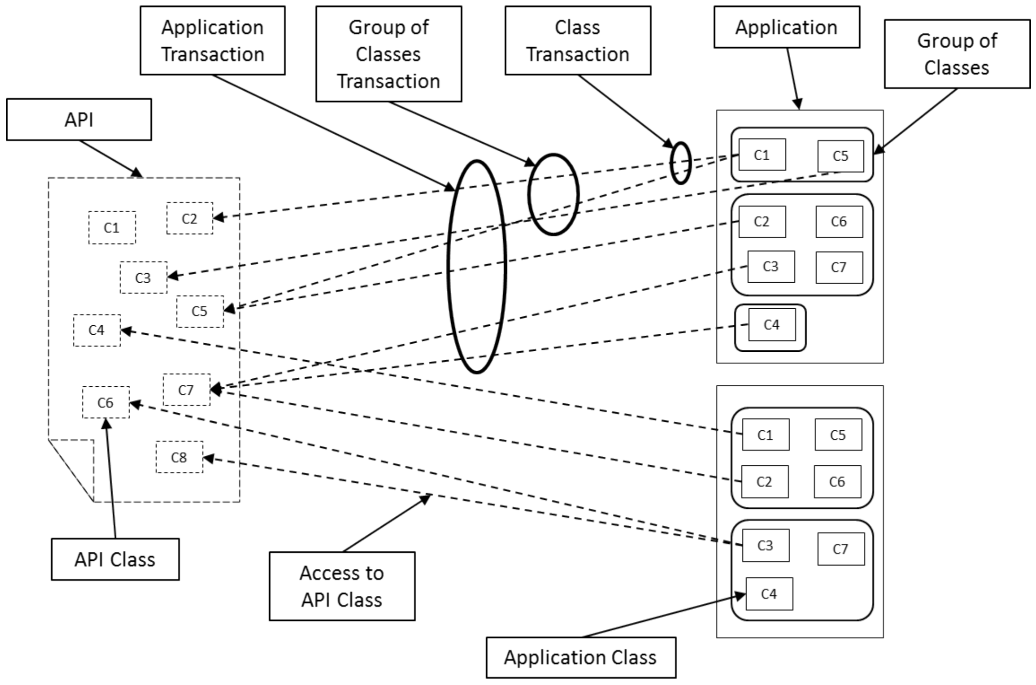


Figure 4.5: Client components using API

4.3.2 Mining Frequent Usage Patterns of Classes

In the previous step, the interactions of all client components with the API are identified as transactions. Based on these transactions, we identify FUPs. A FUP is defined as a set of API classes that are frequently used together by client components. A group of classes is considered as frequent pattern if it reaches a predefined threshold of interestingness metric. This metric is known as *Support*. The *Support* refers to the probability of finding a set of API classes in the transactions. For example, the value of 0.30 *Support*, means that 30% of all the transactions contain the target classes. The following equation refers to the *Support*:

$$S(E1, E2) = P(E1UE2) \tag{4.1}$$

Where:

- $E1$, $E2$ are sets of items.
- S refers to the *Support*.
- P refers to the probability.

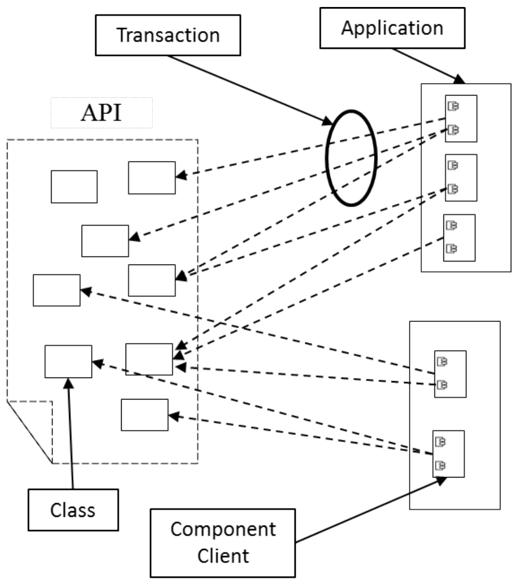


Figure 4.6: Client components using API

Algorithm 8: Identifying Transactions

```
Input: Source Code of a Set of Software Clients($Clients$), API Source
       Code($API$)
Output: A Set of Transactions($trans$)
for each $client \in Clients$ do
    $components$.add(ROMANTIC($client$.sourceCode));
end
for each $com \in components$ do
    $transaction = \varnothing$;
    for each $class \in com$ do
        $transaction$.add(class.getUsedClasses($API$.sourceCode));
    end
    $trans$.add($transaction$);
end
return $trans$
```

4.3.2.1 FUPs Mining Algorithms: an Analysis

The identification of groups of classes forming FUPs can be done based on several algorithms. One of them is the Brute-Force algorithm [Han 2006] that identifies all possible groups of classes. Then, it prunes groups that do not reach the predefined *Support* threshold value. However this algorithm is computationally prohibitive since that the identification of all groups, corresponding to N classes, needs 2^N time complexity [Han 2006]. Another algorithm is the Apriori algorithm that utilizes the property of *anti-monotone* [Han 2006], which means that if a group of classes is considered as infrequent, then all of its supersets must be infrequent as well. Thus they do not need to be generated. However this algorithm still has to generate the candidate groups of classes. For instance, suppose that we have 10^4 frequent groups of classes of size *1*, it requires to generate about 10^7 groups of size *2*. Furthermore, it needs to generate about 10^{30} groups of size *10*. Thus this algorithm does not work in the situation where low *Support* threshold values are selected [Han 2000]. Another algorithm is the Frequent-Pattern Growth (FPGrowth) algorithm [Han 2000]. In this algorithm, these is no need to produce the candidate groups. Instead, it uses a divide-and-conquer technique to mine FUPs. It firstly build a special data structure called Frequent-Pattern tree (FP-tree). This tree is used to compress information of class associations. Then, FPGrowth divides the FP-tree into a collection of databases, such that each one is related to one frequent group of classes.

Table 4.1: An example of transactions composed of API classes

Transaction ID	List of Classes
T1	C1, C2, C5
T2	C2, C4
T3	C2, C3
T4	C1, C2, C4
T5	C1, C3
T6	C2, C3
T7	C1, C3
T8	C1, C2, C3, C5
T9	C1, C2, C3

4.3.2.2 Frequent-Pattern Growth Algorithm

Among the presented algorithms, FPGrowth is the best one since that it outperforms the others in terms of time and space complexity [Han 2006]. Thus we mine FUPs based on the FPGrowth algorithm. To better understand how FPGrowth works, we provide an illustrative example. In this example, we have *9* transactions presented in Table 4.1. The algorithm starts by building the FP-tree corresponds to these transactions. To this end, it firstly scans the transactions to find the frequency of each API class. In our example, the frequencies of *C1, C2, C3, C4* and *C5* are

Table 4.2: Classes ordering inside the transactions

Transaction ID	Ordered Classes
T1	C2, C1, C5
T2	C2, C4
T3	C2, C3
T4	C2, C1, C4
T5	C1, C3
T6	C2, C3
T7	C1, C3
T8	C2, C1, C3, C5
T9	C2, C1, C3

respectively *6, 7, 6, 2* and *2*. Then, the classes are sorted in a descending order according to their frequency values. That is *C2, C1, C3, C4, C5*. Next, The classes inside the transactions are ordered according to their frequency values (see Table 4.2). Then, the tree is build based on the ordered transactions as follows: starting from the root of the tree, which is labeled by a *NULL* value, each transaction is added as a branch in the tree, such that the class which has the highest frequency is firstly added and so on. In the example, the order is *C2, C1, C5* for the first transaction. Whenever a branch shares a common prefix with an already added branch, we only increment the frequency of the shared nodes. Figure 4.7 explains the process of building the FP-tree.

Based on the FP-tree, the algorithm extracts conditional pattern bases and a conditional FP-tree for each frequent class. Conditional pattern bases consist of the collection of paths that co-located with the suffix pattern, while the conditional FP-trees are the subtrees that generate the pattern. For example, the conditional pattern bases corresponding to *C5* is {{*C2:1, C1:1*}, {*C2:1, C1:1, C3: 1*} }, thus the conditional FP-tree is $\langle C2:2, C1:2\rangle$. Paths that do not reach the predefined threshold value are rejected. For example, if the threshold is *2*, the path $\langle C2:2, C1:2, C31\rangle$ is excluded since its frequency is *1*. The set of FUPs identified from our example is {*{C2, C1, C5}*, *{C2, C4}*, *{C2, C1, C3}*, *{C2, C1}*}.

4.3.2.3 Less Commonly Used Classes

The use of the *Support* metric separates the classes of API into two groups according to whether they belong to at least one FUP or not. Classes that do not belong to any of the identified FUPs are the less commonly used classes. As each API class that belongs to a transaction is a class that has been accessed by the clients of the API, therefore it must be a part of the classes composing the interfaces of at least one component. We propose assigning each class of the less commonly used classes to the pattern holding the maximum *Support* value when they are merged together.

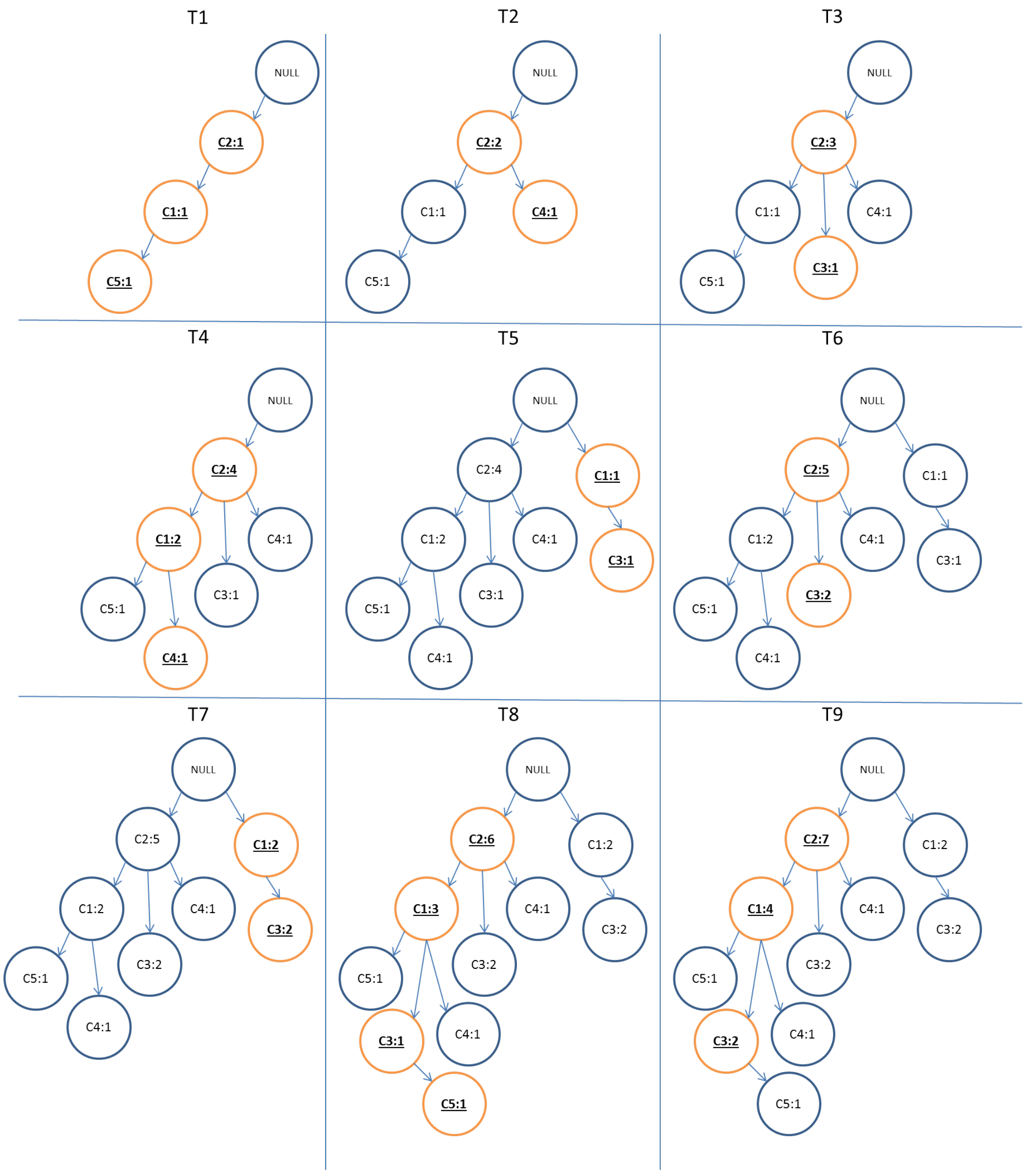


Figure 4.7: Identifying classes composing components

4.3.3 Identifying Classes Composing Component Interfaces from Frequent Usage Patterns

We identify classes composing component interfaces from those composing FUPs. Each FUP is partitioned into a set of groups, where each group represents a component interface. Classes are grouped together according to three heuristics that measure the probability of a set of classes to be a part of the same interface.

1. **Frequency of simultaneous use:** classes composing a FUP are different

compared to their co-reused together by software applications. As much as a collection of classes reused together, the probability of offering the same services is higher. To this end, we rely on *Support* metric to measure the association frequency of a set of classes.

2. **Cohesion:** a group of classes that accesses and shares the same data (e.g., attributes) is strongly related to the same services. Thus we consider that the cohesion of a group of classes indicates to their connectivity. To this end, we use LCC metric [Bieman 1995] to measure the cohesion of a set of classes. We select LCC since it measures both direct and indirect dependencies between the classes.

3. **Lexical similarity:** in most cases, classes of an API are well-documented (i.e., the identifier names are meaningful). Thus their identifier names indicate to the offered services. Thus a group of classes having similar identifier names likely belongs to the same services. To this end, we utilize *Conceptual Coupling* metric [Poshyvanyk 2006] to measure classes lexical similarity based on the semantic information obtained from the source code, encoded in identifiers and comments..

Based on the above heuristics, we propose a fitness function, given below, measuring the ability of a group of classes to form a component interface. This function is used to partition each FUP into groups of classes using a hierarchical clustering algorithm (see Algorithm 2 and Algorithm 3).

$$IQ(E) = \frac{1}{\sum_i \lambda_i} \cdot (\lambda_1 \cdot LCC(E) + \lambda_2 \cdot CC(E) + \lambda_3 \cdot S(E)) \tag{4.2}$$

Where:

- E is a set of object-oriented classes
- *LCC(E)* is the *Cohesion* of E
- *CC(E)* is *Conceptual Coupling* of E
- *S(E)* is the *Support* of E
- λ_1, λ_2, and λ_3 are weight values, situated in [0-1]. These are used by the API expert to weight each characteristic as needed.

4.4 API as Library of Components

4.4.1 Identifying Classes Composing Components

As we mentioned before, the component identification process is driven by the identification of its provided interfaces. This means that API classes forming a component

Algorithm 9: Identifying classes composing components

```
Input: Sets of Provided Interface Classes($interfaces$), API Source
       Code($API$)
Output: A Set of Components($components$)
for each inter in interfaces do
    $comp = inter$.getClasses();
    $bestComp = comp$;
    $searchSpace = API$.getConnectedClasses($inter$);
    while ($|searchClasses| > 1$ ) do
        $c =$ Q.getMaximizeClass($searchSpace$, $comp$);
        $searchSpace$.remove($c$);
        $comp = comp \cup c$;
        if Q(comp)) > Q(bestComp) then
            $bestComp = comp$;
        end
    end
    $components$.add($bestComp$);
end
return components
```

are identified in relation to their structural dependencies with the classes forming provided interfaces of the component. Thus classes having either direct or indirect links with the interface ones compose the search space of classes that may be added to the component. The selection of a group of classes, from the search space, is based on the measurement of the quality of the component, when they are included.

To identify the best group of classes, we need to investigate all subsets of candidate classes. Then, the set that maximizes the component quality is selected. However this requires an exponential time complexity to identify all subsets (i.e., NP-hard problem). Thus we present a heuristic-based technique that identifies a good enough group of classes of the corresponding optimal group.

The identification of these classes is done gradually. In other words, we start to form the group of classes composing the interface ones, and then we add other classes to form a component based on the component quality measurement model. Classes having either direct or indirect links with the interface ones represent the candidate classes to be added to the component. At each step, we add a new API class. This is selected based on the quality value of the component, formed by adding this class to the ones already selected. The class that maximizes the quality value is selected in this step. This is done until all search space classes are investigated.

Each time we add a class, we evaluate the component quality. Then, we select the peak quality value to decide which classes form the component. This means that we exclude classes added after the peak value. As an example, $Class7$ and $Class8$ in Figure 4.8 are excluded from the resulting component because they were added after the quality value reached the peak. Algorithm 9 illustrates the process

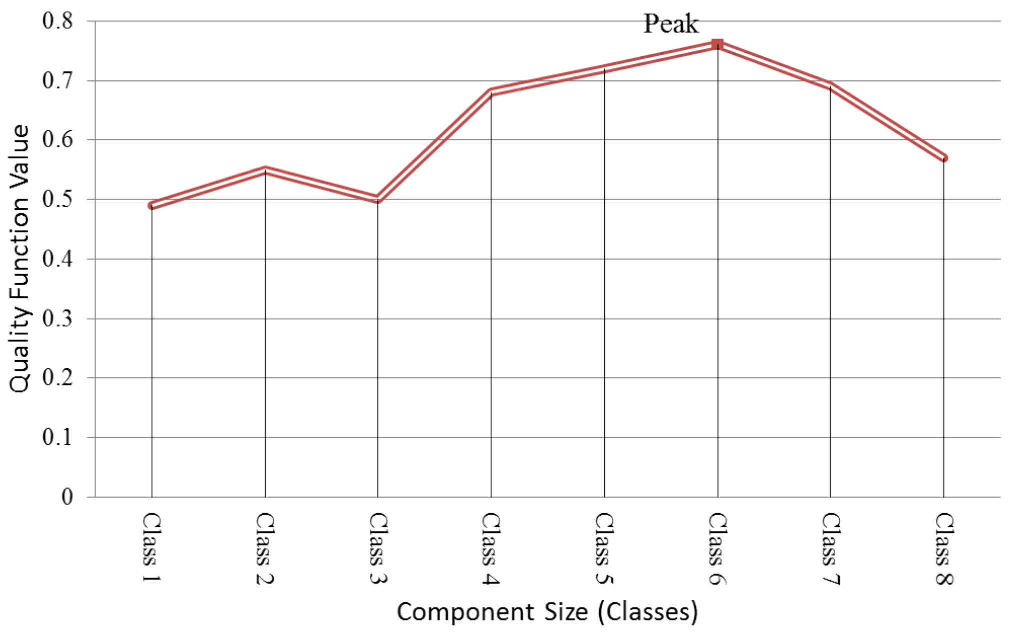


Figure 4.8: Identifying classes composing components

of identification of classes composing a component. In this algorithm, Q refers to the quality fitness function.

4.4.2 Organizing API as Layers of Components

4.4.2.1 Problem Analysis

As we previously mentioned, the API is structured in N layers of components. To identify components of layer L, we rely on components of layer $L-1$. We proceed similarly to the identification of the components of the first layer. We use required interfaces of the components already identified in layer $L-1$ to identify the interfaces provided by components in layer L. This continues until reaching a layer where its components either do not require any interface or they require ones already identified. Figure 4.9 shows an example that illustrates how the components composing each layer are identified, where Figure 4.9.a presents an object oriented API, Figure 4.9.b shows how the first layer components are identified, Figure 4.9.c explains the second layer component identification and Figure 4.9.d shows the resulted component-based API.

4.4.2.2 Identification Algorithm

Each interface that is defined as a required for a component of layer $L-1$ is considered as a provided by a component of layer L except ones provided by the already

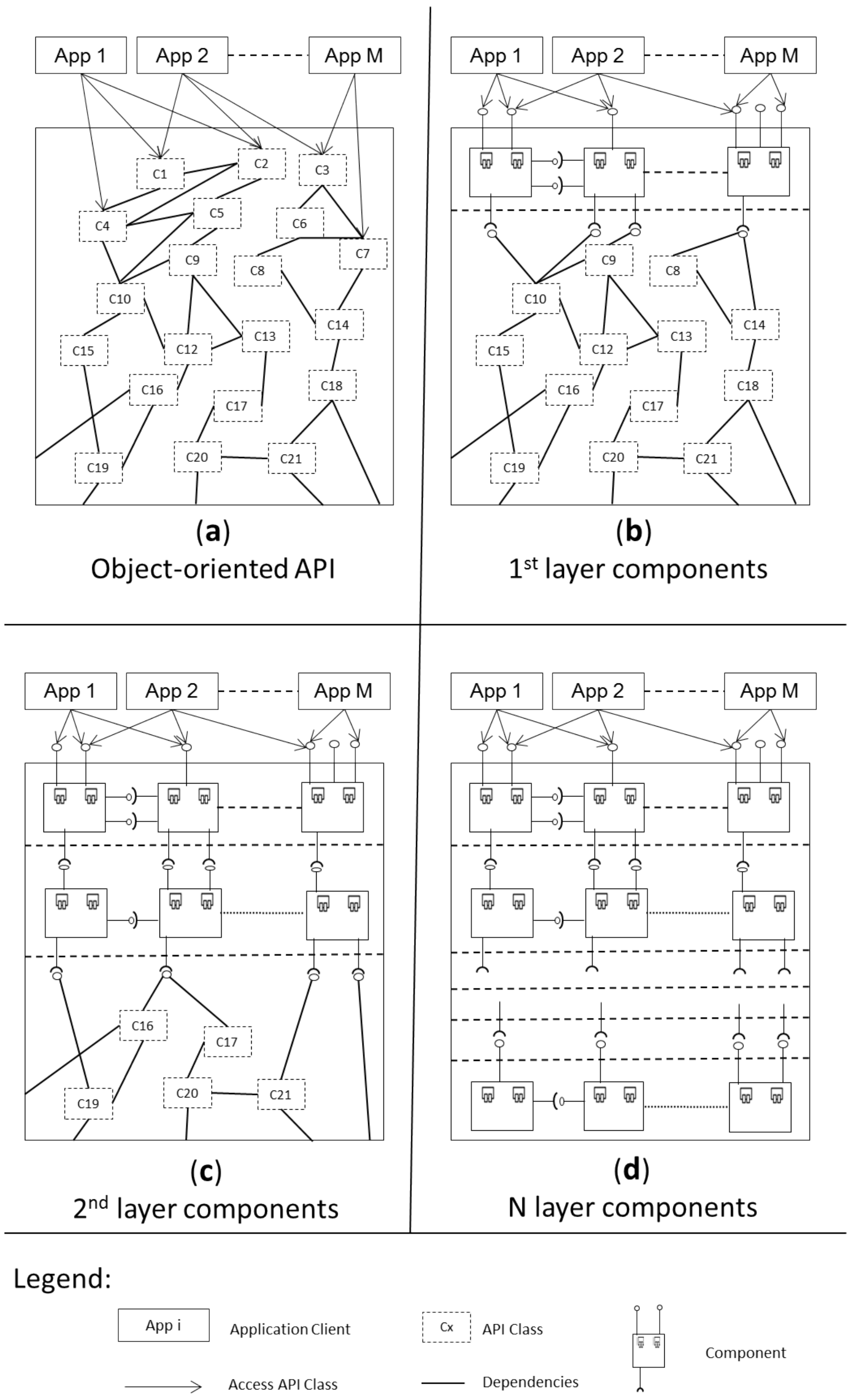


Figure 4.9: Identifying component-based API as layers of components

identified components. The identification of these interfaces is similar to the identification of provided interfaces of the first layer. Thus we consider that each component (already identified) in layer $L - 1$ is a client of the rest of API classes. This means that we collect a set of transactions, such that each transaction composes of classes that have got accessed by a component in layer $L - 1$. These transactions are used to identify FUPs based on FP-Growth algorithm. Similar to the first layer, each FUP is divided into groups of classes composing provided interfaces of components in layer L. The partitioning is based on the (i) the cohesion of classes, (ii) the lexical similarity of these classes and (iii) the frequency of their simultaneous use. Analogously to the identification of the components of the first layer, the other classes composing each component are identified starting from classes composed of its already identified provided interfaces. Algorithm 10 shows the procedure that identifies component-based API as a set of layers composing of components.

Algorithm 10: Organizing API as Layers of Components

```
Input: Source Code of a Set of Application Clients($AppClients$), API
       Source Code($API$)
Output: Component-Based API as Layers of Components($CB - API$)
$clients = AppClients$;
$layerIndex = 1$;
while ($|API| > 1$ ) do
    $transactons$ = extractTransactions($clients$, $API$);
    $FUPs$ = FPGrowth($transactons$, $SupportThreshold$);
    for each $pattern \in FUPs$ do
        ▷IQ refers to Equation 4.2
        $ProvideInterfaces = ProvideInterfaces \cup$clustering($pattern$, $IQ$);
    end
    ▷Identifying classes composing components
    $components$ = Algorithm9($providedInterfaces, API$);
    $CB - API$.addLayer($layerIndex$, $components$);
    $layerIndex = layerIndex + 1$;
    $API = API$ - $components$.getClasses();
    $clients = components$;
end
return $CB - API$
```

4.5 Experimental Results and Evaluation

4.5.1 Experimentation Design

4.5.1.1 Data Collection

We collected a set of 100 *Android−Java* applications from open-source repositories[1]. The average size of these applications in terms of number of classes is 90. Table 4.3 presents the names of the applications. These applications are developed based on classes of the *android* APIs[2]. In our experimentation, we focus on three of these APIs. The first one is the *android.view* composed of 491 classes. This API provides services related to the definition and management of the user interfaces in android applications. The second API is the *android.app* composed of 361 classes. This API provides services related to creating and managing android applications. The last API is the *android* that composes of 5790 classes. This API includes all of the android services.

Table 4.3: The name of the android applications

ADW Launcher	APV	ARMarker	ARviewer	Alerts
Alogcat	AndorsTrail	AndroMaze	AndroidomaticKeyer	AppsOrganizer
AripucaTracker	AsciiCam	Asqare	AugmentRealityFW	AussieWeatherRadar
AutoAnswer	Avare	BansheeRemote	BiSMoClient	BigPlanetTracks
BinauralBeats	Blokish	BostonBusMap	CH-EtherDroid	CVox
CalendarPicker	CamTimer	ChanImageBrowser	CidrCalculator	ColorPicker
CompareMyDinner	ConnectBot	CorporateAddressBook	Countdown	CountdownTimer
CrossWord	CustomMaps	DIYgenomics	Dazzle	Dialer2
DiskUsage	DistLibrary	Dolphin	Doom	DriSMo
DroidLife	DroidStack	Droidar	ExchangeOWA	FeedGoal
FileManager	FloatingImage	Gcstar	GeekList	GetARobotVPNFrontend
GlTron	GoHome	GoogleMapsSupport	GraphView	HeartSong
Hermit	Historify	Holoken	HotDeath	Introspy
LegoMindstroms	Lexic	LibVoyager	LiveMusic	LocaleBridge
Look	LookSocial	MAME4droid	Macnos	Mandelbrot
Mathdoku	MediaPlayer	Ministocks	MotionDetection	NGNStack
NewspaperPuzzles	OnMyWay	OpenIntents	OpenMap	OpenSudoku
Pedometer	Phoenix	PhotSpot	Prey	PubkeyGenerator
PwdHash	QueueMan	RateBeerMobile	AlienbloodBath	SuperGenPass
SwallowCatcher	Swiftp	Tumblife	VectorPinball	WordSearch

4.5.1.2 Research Questions and Evaluation Method

The approach is evaluated on the collected software applications and APIs. We identify client components independently for each software application. Each component in software is considered as a client of the APIs to form a transaction of classes. Then, we mine Frequent Usage Patterns (FUPs) from the identified transactions. Next, from classes composing each FUP, we identify classes composing a set of component interfaces. Then, we identify all component classes starting from

[1]*sourceforge.net*, *code.google.com*, *github.com*, *gitorious.org*, and *aopensource.com*

[2]We select android API level 14 as a reference

ones composing their interfaces. Lastly, the final results obtained by our approach are presented.

We evaluate the obtained components by answering the three following research questions.

- **RQ1: Does the Approach Reduce the Understandability Efforts?** This research question aims at measuring the saved efforts in the API understandability that are brought by migrating object-oriented APIs into component-based ones.
- **RQ2: Are the Mined Components Reusable?** As our approach aims at mining reusable components, we evaluate the reusability of the resulted component. This is based on measuring how much related classes are grouped into the same components.
- **RQ3: Is the Identification of Provided Interfaces Based on FUPs Useful?** The proposed approach identifies the provided interfaces of the components based on how clients have used the API classes (i.e., FUPs). Thus, this research question evaluates how much benefit the use of FUPs brings by comparing components identified by our approach with the ones identified without taking FUPs into account.

4.5.2 Results

4.5.2.1 Extracting Transactions of Usage

The average number of client components identified from each software is 4.5 and the average number of classes composing each component is 18.73. Table 4.4 shows the average number of transactions per software application ($ANTIC$), the average transaction size in terms of classes (ATS), and the percentage of components that have used the API (PCU). The last column of this table shows an example of transactions.

The results show that *android*, *view*, and *app* APIs have been used respectively by only 54%, 29% and 32% of client components. In addition, we note that each client component has used the API classes intensively compared to the number of classes composing it. For example, the transaction size is 17.91 classes for the *view* API, where the average number of classes per component is 18.73. This is due to the fact that classes that serve the same services in software applications, and consequently depend on the same API classes, are grouped together in the same client component.

4.5.2.2 Mining Frequent Usage Patterns of Classes

The identification of FUPs relies on the value of the *Support* threshold. The number and the size of the mined FUPs depend on this value. For all application domains where FUPs are used (e.g., data mining), this value is determined by domain experts.

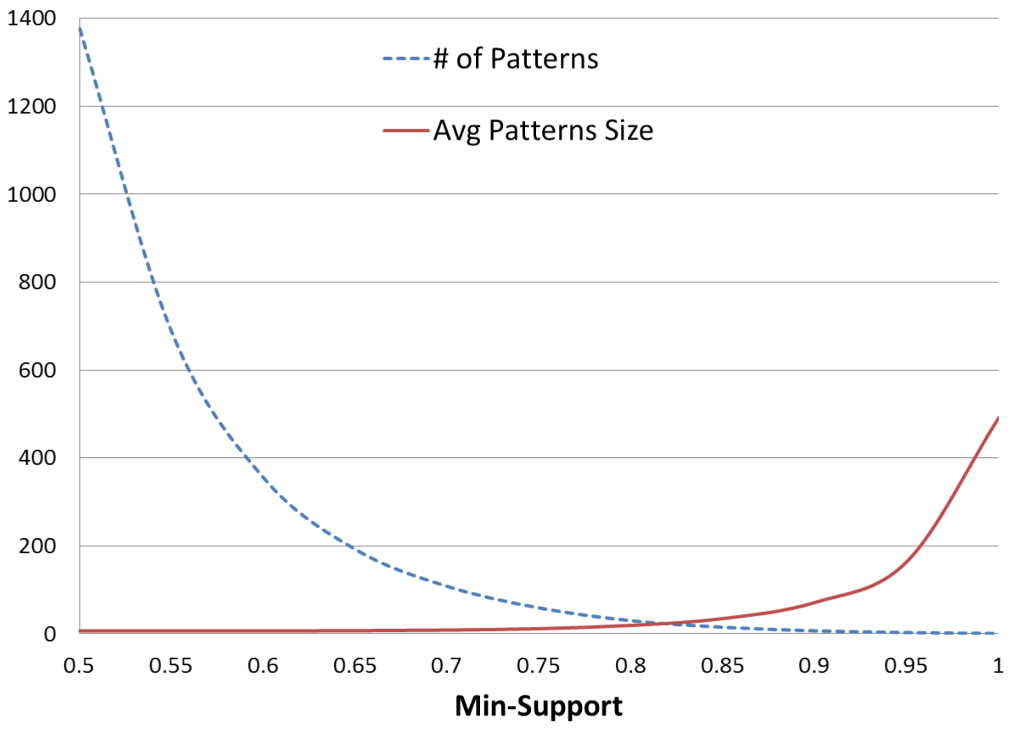


Figure 4.10: Changing the support threshold value to mine FUPs in *android* API

Table 4.4: The Identification of Transactions

API	ANTIC	ATS	PCU	Example
android	2.61	64.82	0.54	Bitmap, Path, Log, Activity, Location, Canvas, Paint, ViewGroup, MotionEvent, View, TextView, GestureDetector
view	1.51	17.91	0.29	MenuItem, Menu, View, ContextMenu, WindowManager, MenuInflater, Display, LayoutInflater
app	1.58	10.90	0.32	ProgressDialog, Dialog, AlertDialog, Activity, ActionBar, Builder, ListActivity

In our approach, to help API experts to determine this value, we assign the *Support* threshold values situated in [30%-100%]. We give for each *Support* value the number of the mined FUPs and the average size of the mined FUPs for each API. Figure 4.10, Figure 4.11 and Figure 4.12 respectively refer to the results of the *android*, the *view* and the *app* APIs. The results show that the number of mined FUPs is directly proportional to the *Support* value, while the average size of the mined FUPs is inversely proportional.

Based on their knowledge of the API, API experts can select the value of the *Support*. For example, if the known average number of API classes used together

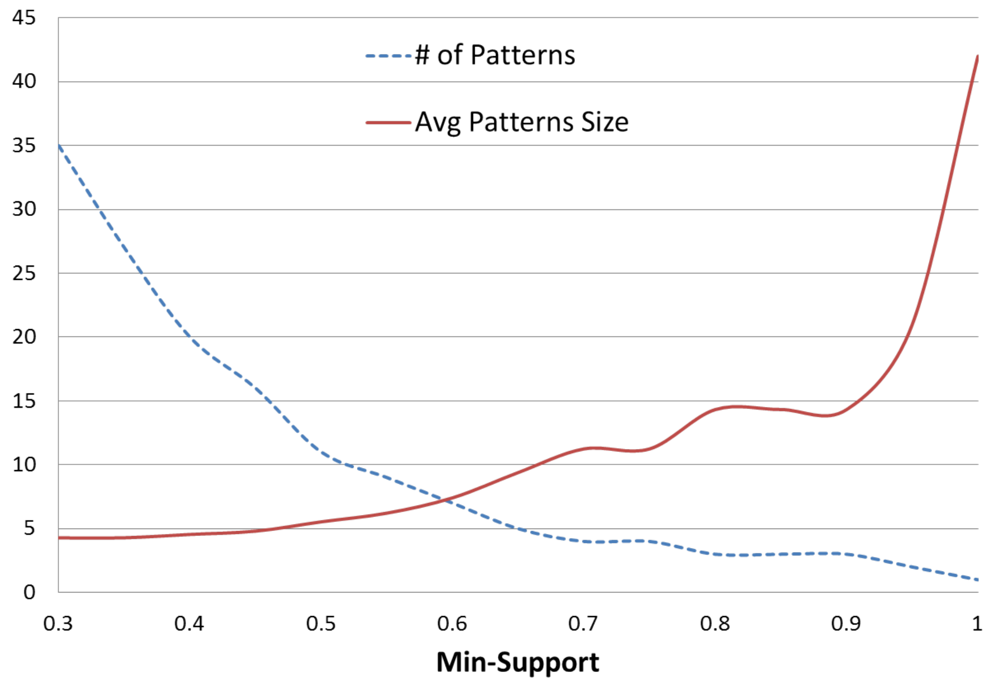


Figure 4.11: Changing the support threshold value to mine FUPs in *view* API

to implement an application service is N, then the experts can choose the *Support* value corresponding to FUPs having N as the average size. Based on the obtained results and our knowledge on android APIs, we select the *Support* threshold values as 60%, 45%, and 45% respectively for the *android*, the *view* and the *app* APIs.

Table 4.5 shows examples of the mined FUPs. For instance, the FUP related to *view* API contains 10 classes. The analysis of this FUP shows that it corresponds to three services: animation (*Animation* and *AnimationUtils* classes), view (*Surface, SurfaceView, SurfaceHolder, MeasureSpec, ViewManager* and *MenuInflater* classes), and persistence of the view states (*AbsSavedState* and *AccessibilityRecord* classes). These services are dependent. Animation service needs the view service and the data of animation view needs to be persistent.

4.5.2.3 Identifying Classes Composing Component Interfaces from Frequent Usage Patterns

In Table 4.6, we present the results of interface identification in terms of the average number of component interfaces identified from a FUP ($ANCIP$), the average number of classes composing component interfaces ($ACIS$) and the total number of component interfaces in the API ($TNCI$). The last column of this table presents examples of component interfaces identified from the FUPs given in Table 4.5.

The results show that FUPs contain classes corresponding to a different set of

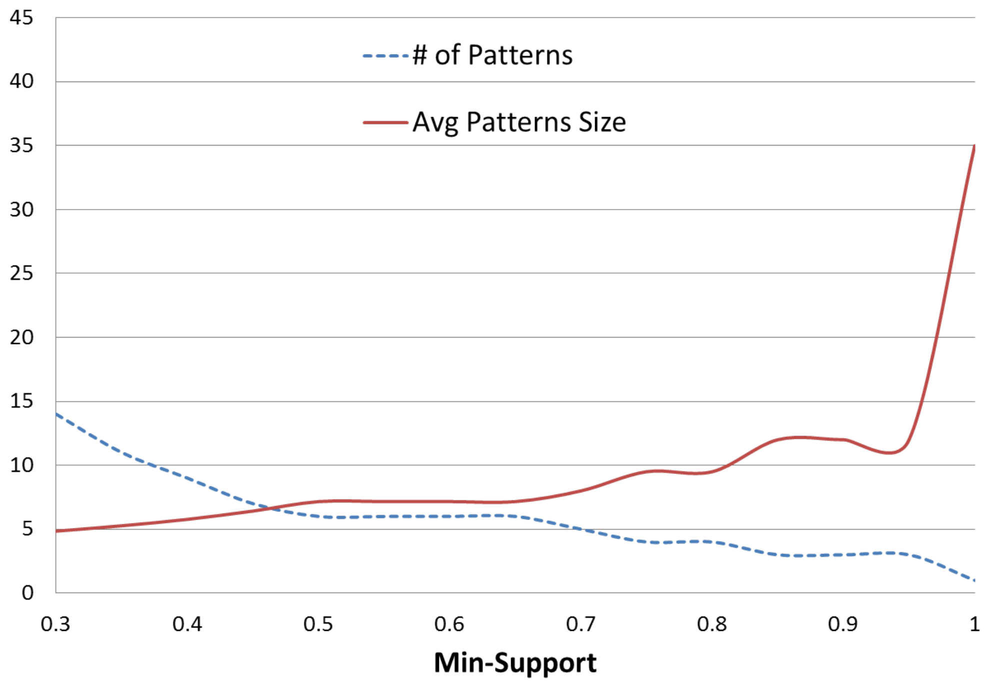


Figure 4.12: Changing the support threshold value to mine FUPs in *app* API

Table 4.5: Examples of the Mined FUPs

API	Example
android	Intent, Context, Log, SharedPreferences, View, TextView, Toast, Activity, Resources
view	Surface, Animation, AnimationUtils, AccessibilityRecord, ViewManager, MenuInflater, AbsSavedState, SurfaceView, SurfaceHolder, MeasureSpec
app	Dialog, Activity, ProgressDialog

services. In average, each FUP is divided into *1.57*, *2.17* and *2.5* services, such that each service is provided by *5.62*, *2.94* and *4* classes respectively for *android*, *view* and *app* APIs. Figure 4.13 shows an instance of partitioning a FUP into component interfaces from *view* API. The analysis of classes composing the identified component interfaces shows that they are related to three services; animation, view and persistent of the view states.

4.5.2.4 Identifying Classes Composing Components

Table 4.7 presents the results related to the mined components composing the first API layer. For each API, we give the number of the mined components (NMC)

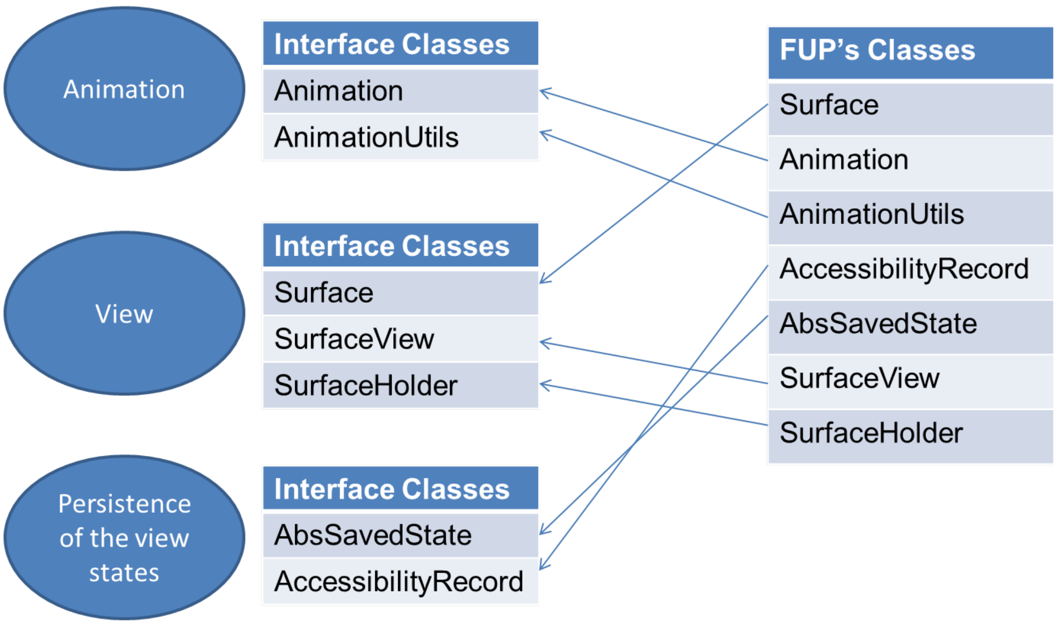


Figure 4.13: An instance of partitioning a FUP into component interfaces from *view* API

Table 4.6: Identification of Component Interfaces from FUPs

API	ANCIP	ACIS	TNCI	Example
android	1.57	5.62	232	Activity, View, TextView, Toast
view	2.17	2.94	19	Surface, SurfaceView, SurfaceHolder
app	2.50	4	10	Dialog, ProgressDialog

and the average number of classes composing the mined components (ACS). The last column of this table shows examples of classes composing components identified started from classes composing provided component interfaces presented in Table 4.6. The results show that the services offered by classes of *android*, *view* and *app* APIs are identified as 232, 19 and 10 components respectively. This means that developers only require to interact with these components to get the needed services from these APIs.

4.5.2.5 Final Results

Table 4.8 shows the final results obtained by our approach. For each API, we firstly give the size of the API in terms of the number of object-oriented classes composing the API and the number of the identified components. Secondly, we present the total number of used entities (classes and respectively components) by the software clients. The results show that classes participating in providing related services are grouped into one component. Furthermore, the total number of cohesive and

Table 4.7: Identifying Classes Composing Components

API	NMC	ACS	Example
android	232	19.99	Activity, View, TextView, Toast, Drawable, GroupView, Window, Context, ColorStateList, LayoutInflater
view	19	7.49	Surface,SurfaceView, SurfaceHolder, MockView, Display, CallBack
app	10	5.86	Dialog, ProgressDialog, AlertDialog

decoupled services is identified for each API. For instance, *android* API consists of 497 components (coarse-grained services), while *view* and *app* APIs contain 43 and 55 components respectively.

Table 4.8: The Final Results

API Name	API Entity	API size	No. of used Entities
android	$object-oriented$	5790	491
	CB	497	54
view	$object-oriented$	491	42
	CB	43	17
app	$object-oriented$	361	45
	CB	55	5

4.5.3 Answering Research Questions

4.5.3.1 RQ1: Does the Approach Reduce the Understandability Efforts?

The efforts spent to understand such an API is directly proportional to the complexity of the API. This complexity is related to the number of API elements and the individual element's complexity. On the one hand, the reduction in the number of elements composing the API is obtained by grouping classes collaborating to provide one coarse-grained service into one component. The results show that the average number of identified components for the studied APIs is 11% (((497/5790) + (43/491) + (55/361)) /3) of the number of classes composing the APIs. This means that the API size is significantly reduced by mapping class-to-component. On the other hand, the reduction in the individual element complexity is done by migrating object-oriented APIs into component-based ones. Meaning, components define their required and provided interfaces, while object-oriented classes at least do not define required interfaces (e.g., a class may call a large number of methods belonging to a set of classes without an explicit specification of these dependencies). The results show that the average number of used components for the APIs is 4% (((54/491) + (17/42) + (5/45)) /3) of the number of used classes. This means that the effort spent to understand API entities is significantly reduced in the case of software applications developed based on API components compared to the development based on API classes. Note that, developers only need to understand the component interfaces, but not the whole component implementation.

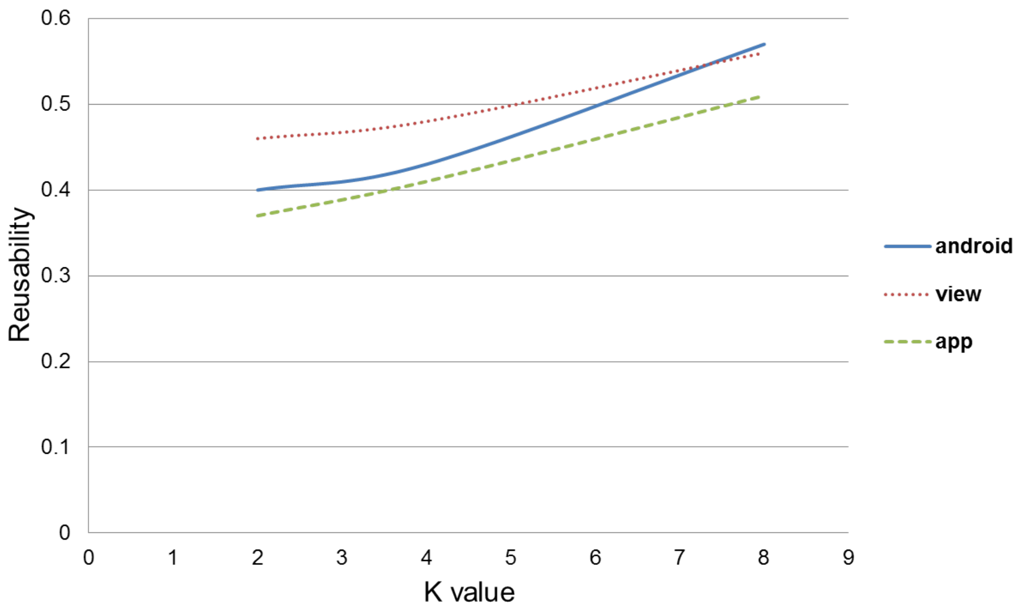
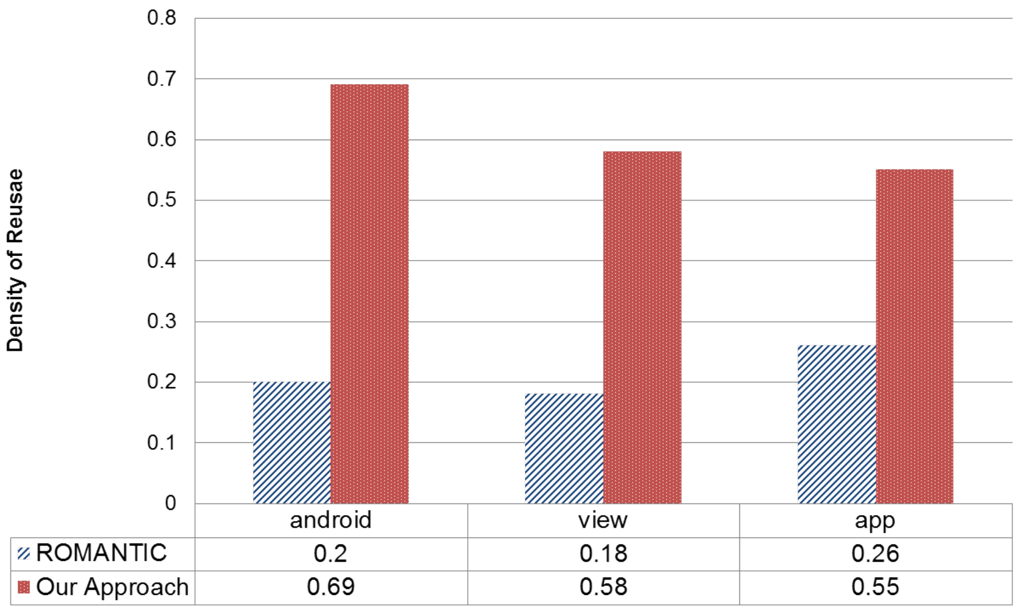


Figure 4.14: Reusability validation results

4.5.3.2 RQ2: Are the Mined Components Reusable?

We consider that the reusability of a software component is related to the number of used classes among all ones composing the software component. Thus, we calculate the reusability of the component based on the ratio between the numbers of used classes composing the component to the total number of classes composing the component. To prove that our resulted component-based APIs could be generalized to another independent set of client applications, we rely on $K-fold$ cross validation method [Han 2006]. The main idea is to evaluate the model using an independent client applications. Thus K-fold divides the set of client applications into K parts. Then, the identification process is applied K times by considering, each time, $K-1$ different parts for the identification process and by using the remaining part to measure the reusability. Next, we take the average of all K trial results. In our experiment, we set K to 2, 4, and 8.

Figure 4.14 presents the results of this measurement. These results show that the reusability results is distributed in a disparate manner. The reason behind this disputation is the size of the train and test data as well as the size of the API. For instance, the average reusability for the *app* API is 37% when the number of train clients is 50 application clients, while it is 51% when the number of train clients is 88 application clients. Thus, the reusability of the components increases when the number of train client applications increases. The results show that our approach identifies reusable components, where the average reusability for all APIs is 47%.



	android	view	app
▨ ROMANTIC	0.2	0.18	0.26
🟥 Our Approach	0.69	0.58	0.55

Figure 4.15: Density validation results

4.5.3.3 RQ3: Is the Identification of Provided Interfaces Based on FUPs Useful?

To prove the utility of using FUPs during the identification process, we compare the components mined based on our approach with ones mined using ROMANTIC approach, which does not take FUPs into consideration. This is based on the density of using the component provided interfaces by application clients. The density refers to the ratio between the number of used interface classes to the total number of interface classes for each component. Figure 4.15 shows the average density for the two identification approaches. These results show that our approach outperforms ROMANTIC approach. For instance, the application clients need to reuse a larger number of components of ones mined based on ROMANTIC with less density of provided interface classes compared to component mined based on our approach. For instance, the average usage density of classes composing provided interfaces of ROMANTIC components is 21%, while it is 61% for components mined by our approach for all APIs.

4.6 Discussion

4.6.1 Component and Frequent Usage Pattern

FUPs and components serve the reuse needs in two different ways. Components are entities that can be directly reused and integrated into software applications, while FUPs are guides for reuse and not entities for reuse. In addition, components and FUPs are structurally different. Related to *Specificity* characteristic, classes composing a component serve a coherent body of services, while classes composing a FUP may be related to different services. Concerning *Autonomy* characteristic, dependencies of component's classes are mostly internal, which forms an autonomous entity. FUP's classes can be very dependent on other classes that are not directly used by clients of APIs. Concerning *Composability* characteristic, a component is structured and reused via interfaces, while FUPs are not directly reusable entities.

4.6.2 Component Identification: APIs VS Software Applications

Classes of APIs are reused by developers to develop their own software applications. However, a software application offers services that can be used directly to meet the needs of end-users. In addition, APIs and software applications are different compared to relationships between the classes that compose them. Classes composing software applications are structural and behavioral dependent to provide the expected services. For APIs, two kinds of relationships characterize their classes. On the one hand, some classes are structural and behavioral dependent, to provide reusable services for software applications. On the other hand, some classes need to be reused together, i.e., simultaneously, by software applications to implement end-user services (e.g., *JFrame* and *Layout* classes in *java.swing* API).

Dependencies between classes composing object-oriented applications have been exploited by approaches that aim to identify components from object-oriented applications. In an analogous manner, we rely on relationships between API classes to identify components. This is based on both the analysis of the structural and behavioral dependencies between classes and the frequency of simultaneous reuse of these classes by clients of APIs.

4.7 Threats to Validity

Two types of threats to validity concern the proposed approach. These are internal and external.

4.7.1 Threats to Internal Validity

There are five aspects to be considered regarding the internal validity. These are as follows:

1. The validation of the understandability, respectively the reusability, of the resulted component-based APIs is not directly measured. On the first hand,

the understandability is measured through the complexity of the resulted API, while in some cases a complex API can be understandable if it is well documented. However, for the same API, the understandability of a complex version is worse than the understandability of a less complex one, even if both versions are already documented.

On the other hand, the reusability is measured based on the number of used classes among the ones composing the components. Although the reusability of components needs to be measured based on their interfaces, this provides an indication of how the component interfaces will be reused by the future software clients.

2. We use FPGrowth algorithm to mine FUPs. Nevertheless this algorithm has a limitation of ignoring classes that their patterns' support values do not reach the support threshold (i.e., less commonly used classes). Thus some of API classes may not be presented by a FUP. However we attach each class of them to a FUP holding the maximum support value when it is added. By doing this, we guarantee that each API class that have reused by software applications is attached to at least one FUP.

3. As our approach is used-driven, the results depend on the quality and the number of usages of the API. This means that identified FUPs rely on the considered software clients. Therefore the identification of provided interfaces and then their corresponding components depends on API clients. Consequently it is essential to select clients having the largest number of usages of the API.

4. In the case of facing a NP-hard problem, we rely on heuristic algorithms instead of optimal algorithms. Therefore this affects the accuracy of the results. However these heuristics guarantee good enough solutions, such as clustering algorithms.

5. Similar to the approach presented in Chapter 3, we rely on the static analysis to identify dependencies between the classes (please refer to the first point of Section 3.7.1).

4.7.2 Threats to External Validity

There are two aspects to be considered regarding the external validity. These are as follows:

1. The presented approach is experimented via APIs that are implemented by *Android* programming language. However the obtained results can be generalized for any object-oriented API. The reason behind this generalization is the fact that all object-oriented languages (e.g., *C++* and *C#*) are structured in terms of classes and their relationships are realized through method calls, access attributes, etc.

2. The way that API classes are reused together may strongly depend on the choice of the subject software applications, i.e., different software applications may use API classes following different patterns, ending up in different components. To prove that our resulted component-based APIs could be generalized to another independent set of software applications, we rely on $K - fold$ cross validation method. The cross validation presents whether components identified on n-K software applications can be reused by K software applications.

4.8 Conclusion

In this chapter, we presented an approach aims at mining software components from object-oriented APIs. This is based on static analysis of the source code of both the APIs and their software clients, in order to analyze the way that the software clients have used the API classes. The component identification process is used-driven. This means that components are identified starting from classes composing their interfaces. Classes composing the provided interface of the first layer components compose FUPs. Then, the API is organized by a set of layers, where each layer composes of components providing services to the others composing the above layer, and so on.

To validate the presented approach, we experimented it by applying on a set of open source *Java* applications as clients for three android APIs. The validation is done through three research questions. The first one is related to the reusability, while the second indicates to the understandability. The results show that our approach improves the reusability and the understandability of the API. The last research question aims at comparing our approach with a traditional component identification approach. The results prove that our approach outperforms the traditional one.

Chapter 5

Recovering Software Architecture of a Set of Similar Object-Oriented Product Variants

Contents

5.1 Introduction

Software Product Line Architecture (SPLA) does not only describe the system structure at a high level of abstraction, but also describes the variability of a SPL by capturing the variability of architecture elements [Pohl 2005a]. This is done by (i) describing how components can be configured to form a concrete architecture, (ii) describing shared components and (iii) describing individual architecture characteristics of each product.

However, developing a SPLA from scratch is a highly costly task [Clements 2002, Pohl 2005a]. Otherwise, SPLA can be recovered based on the analysis of existing software product variants. This is done through the exploitation of the commonality and the variability across the source code of product variants, and thus the extraction of the architecture of each software product and the variability between them. As it is mentioned in the state-of-the-art chapter, there are few approaches that recover SPLA from a set of product variants. However these approaches suffer from two main limitations. The first one is that the architecture variability is partially addressed since they recover only some variability aspects, no one recovers the whole SPLA. For example, [Koschke 2009, Frenzel 2007] do not identify dependencies among the architectural elements. The second one is that they are not fully-automatic since they rely on the domain knowledge which is not always available, such as [Pinzger 2004] and [Kang 2005].

To address this limitation, we propose, in this chapter, an approach that automatically recovers the architecture of a set of software product variants. Our contribution is twofold:

- The identification of architecture variability concerning component and configuration variabilities.
- The identification of architecture dependencies existing between the architecture elements.

In order to validate the proposed approach, we experimented on two families of open-source product variants; Mobile Media and Health Watcher. The evaluation shows that our approach is able to identify the architectural variability and the dependencies as well.

The rest of this chapter is organized as follows: the background needed to understand the approach is presented in Section 5.2. In Section 5.3, we analyze the architecture variability. Then, the recovery process of the architecture variability is presented in Section 5.4. Next, the architecture variability is identified in Section 5.5. In Section 5.6, dependencies between architecture elements are recovered, while the identification of groups of variability is presented in Section 5.7. Our experimentation is discussed in Section 5.8. Threat to validity are discussed in Section 5.9. A conclusion of this chapter is presented in Section 5.10.

5.2 Background

5.2.1 Software Variability

The main theme in SPLE is software variability. It is related to the susceptibility and flexibility to change [Clements 2002]. The variability in a SPL is realized at different levels of abstraction during the development life cycle, e.g., requirement, and design. For instance, at the requirement level, it is originated starting from the differences in users' wishes, and does not carry any technical sense [Pohl 2005a]. This is related to a set of features that are needed to be included in such an application (e.g., the user needs camera, WIFI, and color screen features in the phone). Usually, this variability is documented by feature modeling language [Kang 1990]. At the design level, the variability have more details related to technical senses to form the applications architecture. These technical senses describe how the applications are built and implemented with regard to the point of view of software architects [Pohl 2005a]. Such technical senses are those related to which software components are included in the application (e.g., video recorder, capture a photo, and media store components), how these components interact by their interfaces (e.g., video recorder provides a video stream interface to media store), and what topology forms the architectural configuration (i.e., how components are composited) [Nakagawa 2011]. All of these technical senses are described via SPLA [Pohl 2005a].

5.2.2 Formal Concept Analysis

Formal Concept Analysis (FCA) is a mathematical data analysis technique developed based on lattice theory [Ganter 2012]. FCA is applied to support various tasks in data mining and software engineering domains. It allows the analysis of the relationships between a set of objects described by a set of attributes. In this context, maximal groups of objects sharing the same attributes are called formal concepts. These are extracted and then hierarchically organized into a graph called a concept lattice. Each formal concept consists of two parts. The first allows the representation of the objects covered by the concepts called the extent of the concept. The second allows the representation of the set of attributes shared by the objects belonging to the extent. This is called the intent of the concept. Concepts can be linked through sub-concept and super-concept relationship [Ganter 2012] where the lattice defines a partially ordered structure. A concept A is a sub-concept of the super-concept B, if the extent of the concept B includes the extent of the concept A and the intent of the concept A includes the intent of the concept B.

The input of FCA is called a formal context. A formal context is defined as a triple $K = (O, A, R)$ where O refers to a set of objects, A refers to a set of attributes and R is a binary relation between objects and attributes. This binary relation indicates to a set of attributes that are held by each object (i.e., $R \subseteq O \times A$). Table 5.1 shows an example of a formal context for a set of bodies of waters and their attributes. An X in a cell indicates that an object holds the corresponding attribute.

Table 5.1: Formal context example

	Natural	Artificial	Stagnant	Running	Inland	Maritime	Constant
River	X			X	X		X
Sea	X		X			X	X
Reservoir		X	X		X		X
Channel				X	X		X
Lake	X		X		X		X

As stated before, a formal concept consists of extent E and intent I, with E a subset of objects O ($E \subseteq O$) and I a subset of attributes A ($I \subseteq A$). A pair of extent and intent (E, I) is considered a formal concept, if and only, if E consists of only objects that shared all attributes in I and I consists of only attributes that are shared by all objects in E. The pair ("*river, lake*", "*inland, natural, constant*") is an example of a formal concept of the formal context in Table 5.1.

Fig. 5.1 shows the concept lattice of the formal context presented in Table 5.1. It consists of 13 formal concepts. For more details about FCA, refer to [Ganter 2012].

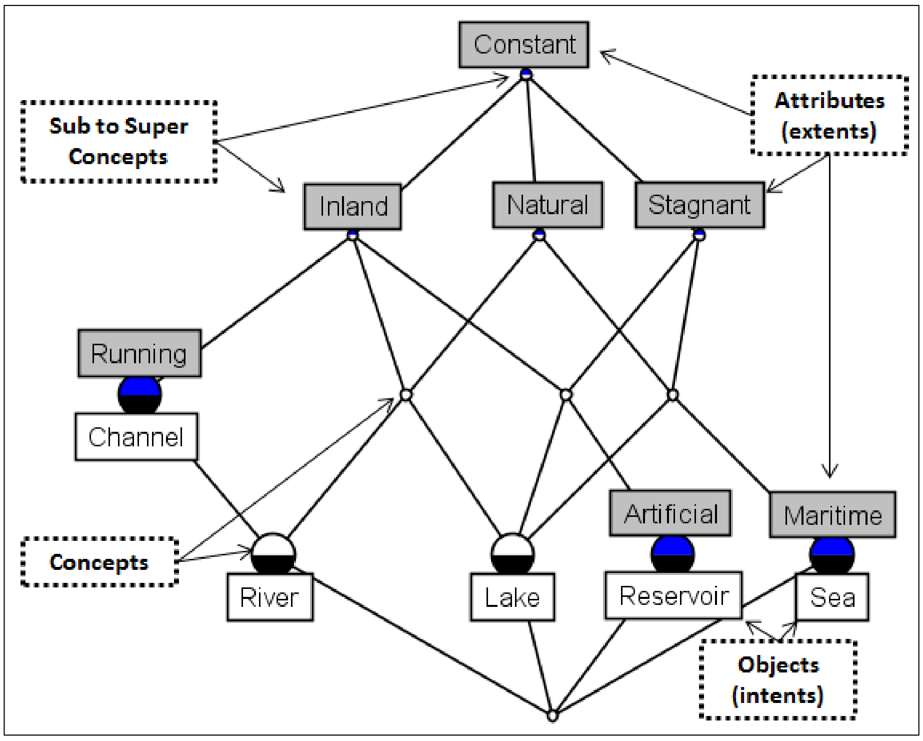


Figure 5.1: The concept lattice of the formal context in Table 5.1

5.3 Architecture Variability Analysis

SPLA aims at realizing the software variability at the architecture level. This is done by exploring the commonality and the variability of architecture elements, i.e., component, connector and configuration variability. In this chapter, we focus on component and configuration variability. This means that connector variability is not considered since that connectors are not considered as first class concepts in many architecture description languages such as [Magee 1996] [Luckham 1996] [Canal 1999]. To better understand the architecture variability, we rely on the example provided in Figure 5.2. This example schemes three product variants related to an audio player product family. Each variant diverges in the set of components constituting its architecture as well as the links between these components.

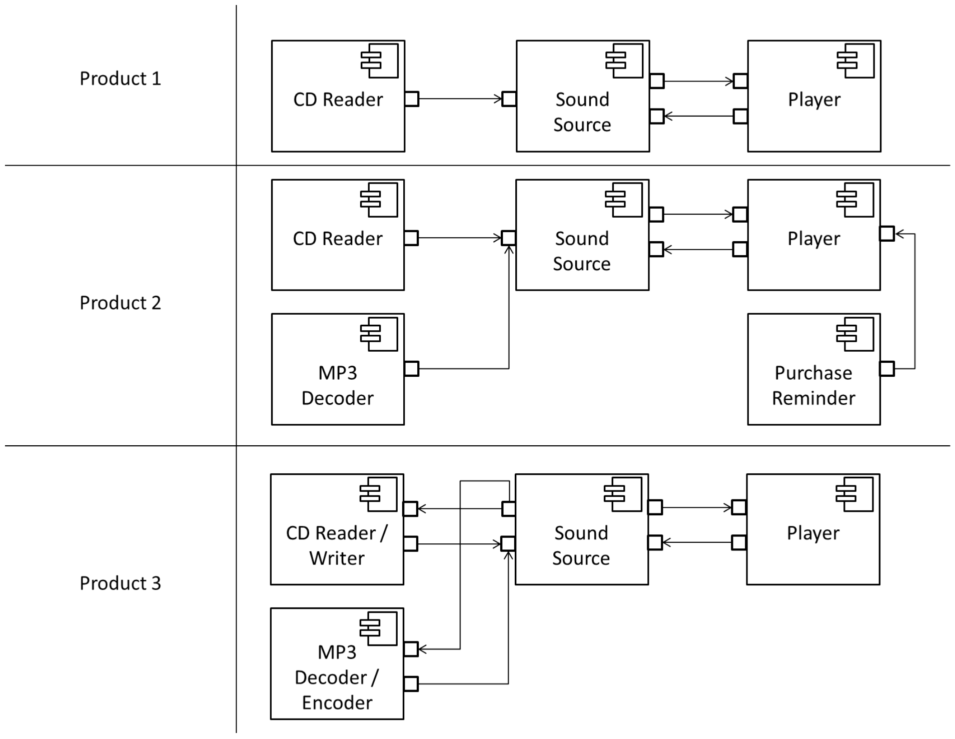


Figure 5.2: An example of architecture variability

Components are considered as the main building unit of an architecture. The variability of components can be considered following two dimensions. The first one is related to the existence of several components having the same architectural meaning, i.e., almost provide the same functionalities. We call these ones component variants. For example, in Figure 5.2, *MP3 Decoder* and *MP3 Decoder / Encoder* are examples of component variants. The second dimension is related to the commonality and the variability between component variants. This is realized thought internal and external variabilities. Internal variability refers to the divergence related to the implementation details of component variants which may lead to variability in the set of functionalities provided by these component variants, e.g., *Decoder* only or *Decoder/Encoder* functionalities. External variability refers to the way that the component interacts with other components. This is realized through the variability in the component interfaces. In our example, the *Sound Source* component variants have either one or two interfaces.

Furthermore architecture configuration does not only define the topology of how components are composited and connected, but also defines the set of included components. Thus configuration variability is represented in terms of presence/absence of components on the one hand, and presence/absence of component-to-component links on the other hand. These respectively refer to the commonality (mandatory) and the variability (optionality) of component and component-link. For example, in Figure 5.2, *Sound Source* is a mandatory component, while *Purchase Reminder* is an optional one. The links that connect *Player* and *Sound Source* are mandatory links, while the link that connect *MP3 Decoder/Encoder* and *Sound Source* is an optional link.

The identification of component and component-link variability is not enough to define a valid architectural configuration. This also depends on the identification of architectural element dependencies, i.e., constraints, that may exist between the elements of the architectures. For instance, components providing antagonism functionalities have an exclude relationship. Furthermore, a component may need other components to perform its services.

5.4 Architecture Variability Recovery Process

Based on the observations made in the previous section, we propose the following process to recover SPLA of a set of product variants (see Figure 5.3):

Identifying the architecture of each single product: we rely on *ROMANTIC* approach to extract the architecture of a single product. It statically analyze the object-oriented source code of each single product.

Identifying component variants: we identify component variants based on the identification of components providing similar functionalities. Then, we analyze their variability in terms of internal and external variability.

Identifying configuration variability: we identify configuration variability based on both the identification of mandatory and optional components and links between these components.

Identifying architecture dependencies: to identify dependencies between the optional components, we rely on Formal Concept Analysis (FCA) that analyzes the distribution of optional components between the products.

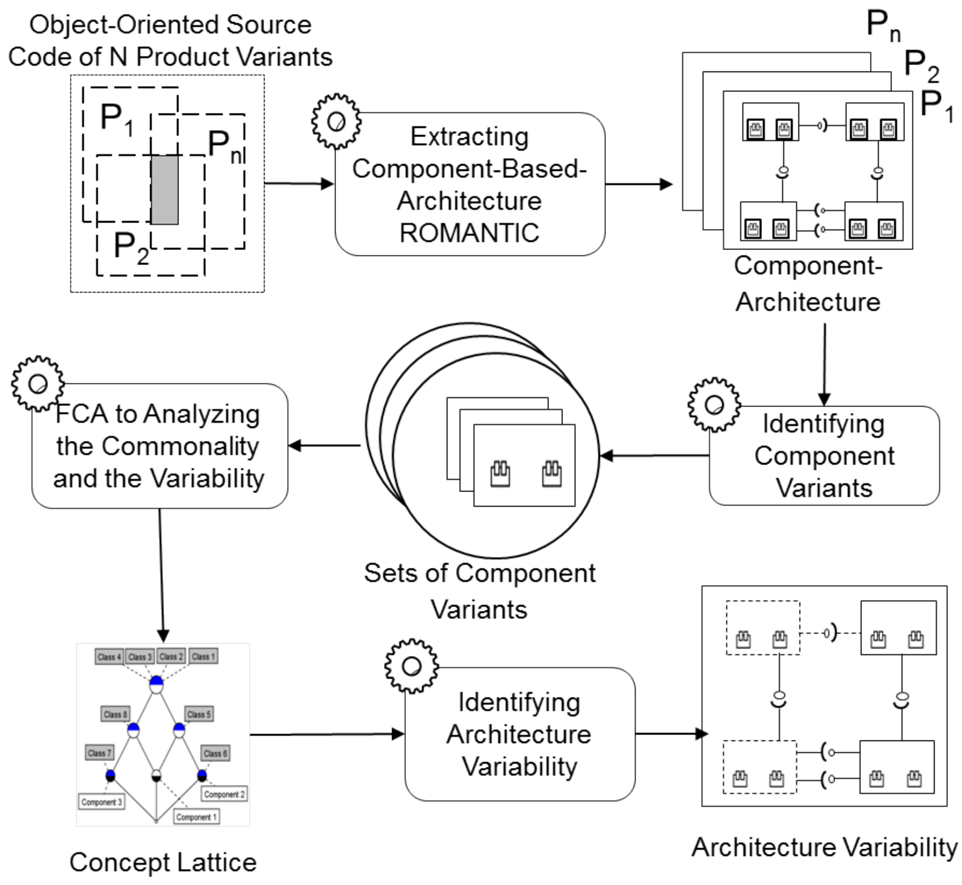

Object-Oriented Source Code of N Product Variants
P1
Pn
P2
Extracting Component-Based-Architecture ROMANTIC
Pn
P2
P1
Component-Architecture
FCA to Analyzing the Commonality and the Variability
Identifying Component Variants
Sets of Component Variants
Identifying Architecture Variability
Concept Lattice
Architecture Variability

Figure 5.3: The process of architectural variability recovery

5.5 Identifying the Architecture Variability

The architecture variability is mainly materialized either through the existence of variants of the same architectural element (i.e., component variants) or through the configuration variability. In this section, we show how component variants and configuration variability are identified.

5.5.1 Identifying Component Variants

The selection of a component to be used in an architecture is based on its provided and required services. The provided services define the role of the component. However, other components may provide the same, or similar, core services. Each may also provide other specific services in addition to the core ones. Considering these components, either as completely different or as the same, does not allow the variability related to components to be captured. Thus, we consider them as component variants. We define component variants as a set of components providing the same core services and differing concerning few secondary ones. In Figure 5.2, *MP3 Decoder* and *MP3 Decoder / Encoder* are considered as component variants. We identify component variants based on the identification of components providing similar functionalities. Then, we analyze their variability in terms of implemented services and provided and required interfaces.

5.5.1.1 Identification of Groups of Similar Components

We identify component variants based on their similarity. Similar components are those sharing the majority of their classes and differing in relation to some others. Components are identified as similar based on the strength of similarity links between their implementing classes. For this purpose, we use cosine similarity metric [Han 2006] where each component is considered as a text document composed of the names of its classes. We use a hierarchical clustering algorithm [Han 2006] to gather similar components into clusters. It starts by considering components as initial leaf nodes in a binary tree. Next, the two most similar nodes are grouped into a new one that forms their parent. This grouping process is repeated until all nodes are grouped into a binary tree (see Algorithm 2). All nodes in this tree are considered as candidates to be selected as groups of similar components. To identify the best nodes, we use a depth first search algorithm (see Algorithm 3). Starting from the tree root to find the cut-off points, we compare the similarity of the current node with its children. If the current node has a similarity value exceeding the average similarity value of its children, then the cut-off point is in the current node. Otherwise, the algorithm continues through its children. The results of this algorithm are clusters where each one is composed of a set of similar components that represent variants of one component.

5.5.1.2 Identification of Internal Variability

Internal structure variability is related to the implementation of components. Component variants are implemented by object-oriented classes. Thus we identify internal variability in terms of class variability. We distinguish two types of classes. The first one refers to classes that represent the commonality. These are classes that belong to all variants of the component. We call them common classes. The second type refers to classes representing the variability. These are classes that do not belong to all variants of the component. We call them variable classes.

Table 5.2: An example of formal context of three component variants

	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Class 8
Variant 1	X	X	X	X	X			X
Variant 2	X	X	X	X	X	X		
Variant 3	X	X	X	X			X	X

The identification of common/variable classes and the distribution of variable classes is achieved using Formal Concept Analysis (FCA). To this end, we build the formal context, such as each component variant is considered as an object and

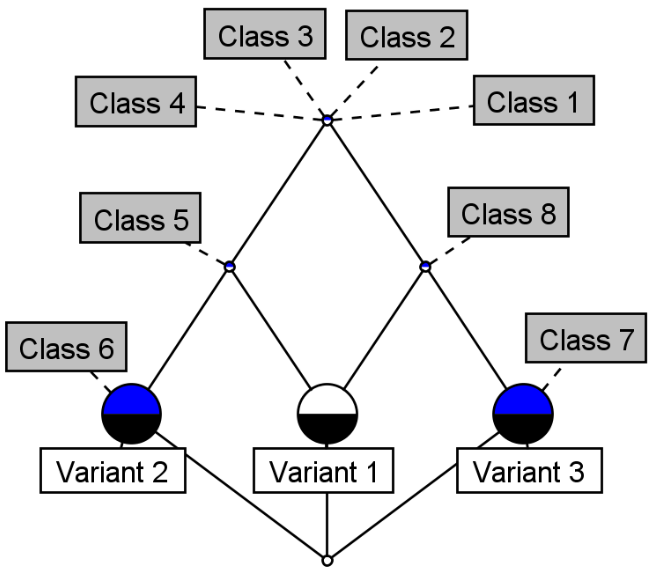


Figure 5.4: A lattice example of component variants

each class is an attribute in this formal context. Table 5.2 shows an example of a formal context built using three component variants. A cross in the cell (V, C) denotes to that the variant V holds the class C. In the *Lattice* generated based on this formal context, common classes are grouped in the root, while the variable ones are hierarchically distributed to the non-root nodes. The leave nodes represent component variants. These variants have all classes that are attached to the nodes in the path to the root node.

Figure 5.4 shows the *Lattice* extracted based on the formal context presented in Table 5.2. On the first hand, the commonality of their implementation is represented by common classes grouped together on the top of the *Lattice* (i.e., the root). *Class1*, *Class2, Class3* and *Class4* are the common classes. On the other hand, the variability of their implementation is represented by variable classes distributed on the non root nodes. For instance, *Class8* belongs to two variants; *Variant1* and *Variant3*, while *Class6* belongs only to one variant; *Variant2.*

5.5.1.3 Identification of External Variability

The interaction between components is realized through their interfaces; provided and required interfaces. Provided interfaces are abstract description of services in providing components and required by other components. In object-oriented components, an interface is the abstraction of a group of method invocations, access attributes or inheritances. This group provides accessing to the component services. As component variants are identified from different products, thus each variant may have some interfaces that are different compared to the other variants. For example, in Figure 5.5, *CD Reader* and *CD Reader/Writer* variants differ compared to their interfaces. In the former, it has only one provided interface that provides the functionality of reading the *CD* content. In the latter, it has an additional required interface compared to the first variant. This interface requires a source of information to be written on the *CD*. The interfaces of a component can be classified into mandatory and optional interfaces. Mandatory interfaces are ones existing in all variants of the component (e.g., the provided interface in Figure 5.5). Optional interfaces are those that are not mandatory (e.g., the required interface in Figure 5.5).

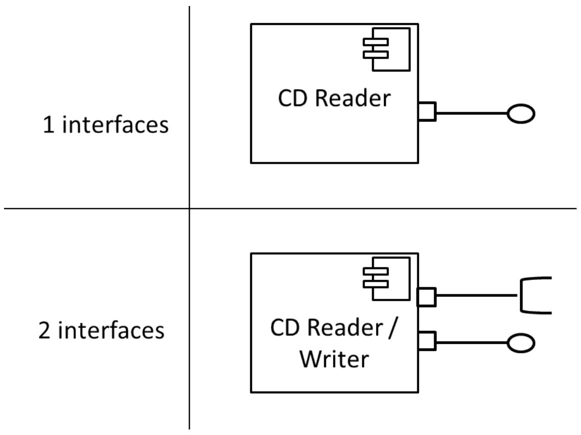


Figure 5.5: An instance of interface variability

To identify interface variability, we proceed as follows. For each variant, we identify its interfaces in the corresponding product in which the variant has been identified. This is done using the approach presented by [Kebir 2012a] which identifies interfaces as groups of methods. To identify whether interfaces are similar or not, we rely on the textual similarity between their implemented methods. If the similarity exceeds a pre-defined threshold value, then they are considered as the same interface, otherwise, they are different ones. The intersection of the sets of interfaces from all the products determines all mandatory interfaces for the given component. The other interfaces are optional ones. Algorithm 11 shows the procedure of identifying mandatory and optional interfaces of a set of component variants, where the function called *identifyInterfaces()* refers to the approach presented by [Kebir 2012a].

5.5.2 Identifying Configuration Variability

The architectural configuration is defined based on the list of components composing the architecture, as well as the topology of the links existing between these components. Thus the configuration variability is related to these two aspects; the lists of core (mandatory) and optional components and the list of core and optional links between the selected components.

Algorithm 11: Identifying Interface Variability

```
Input: A Set of Component Variants ($CV$)
Output: A Set of Mandatory and Optional Interfaces ($MI$, $OI$))
$MI$ = identifyInterfaces($CV$.getFirstVariant());
$allInterfaces = \emptyset$;
for each $v \in CV$ do
    $MI = MI \cap$ identifyInterfaces($v$);
    $allInterfaces = allInterfaces \cup$ identifyInterfaces($v$);
end
$OI = allInterfaces - MC$;
return $MI, OI$
```

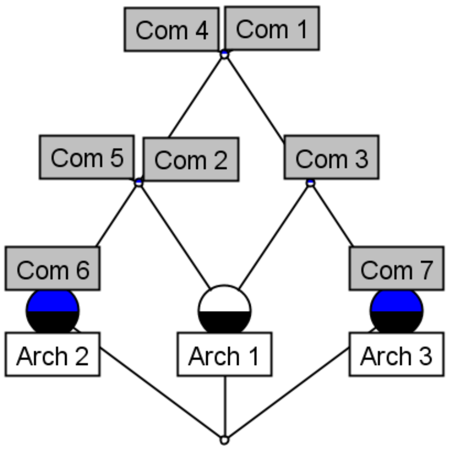


Figure 5.6: A lattice example of similar configurations

5.5.2.1 Identification of component variability

To identify mandatory and optional components, we use Formal Concept Analysis (FCA) to analyze architecture configurations. We present each software architecture as an object and each member component as an attribute in the formal context. In the concept *Lattice*, common attributes are grouped into the root while the variable ones are hierarchically distributed among the non-root concepts.

Figure 5.6 shows an example of a *Lattice* for three similar architecture configurations. The common components (the core ones) are grouped together at the root concept of the lattice (the top). In Figure 5.6 *Com1* and *Com4* are the core components presented in the three architectures. By contrast, optional components are represented in all *Lattice* concepts except the root. e.g., according to the *Lattice* of Figure 5.6,*Com2* and *Com5* present in *Arch1* and *Arch2* but not in *Arch3*.

5.5.2.2 Identification of component-link variability

A component-link is defined as a connection between two components. Each connection is composed of a provided interface of the providing component and a required interface of the other component. Figure 5.7 shows an example of how components are linked through their interfaces.

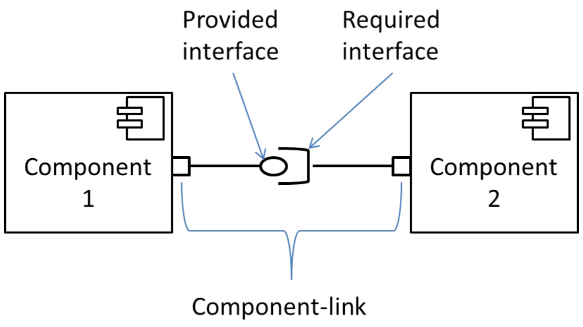


Figure 5.7: An example of component-to-component link

A component may be linked with different sets of components. A component may have links with a set of components in one product, and it may have other links with a different set of components in another product. Thus the component-link variability is related to the component variability. This means that the identification of the link variability is based on the identified component variability. For instance, the existence of a link between *Component A* and *Component B* is related to the selection of *Component A* and *Component B* in the architecture. Thus considering a core link (mandatory link) is based on the occurrence of the linked components, but not on the occurrence in the architecture of products. According to that, a core link is defined as a link occurring in the architecture configuration as well the linked components are selected. To identify the component-link variability, we proceed as follows. For each architectural component, we collect the set of components that are connected to it in each product. The intersection of the sets extracted from all the products determines all core links for the given component. The other links are optional ones.

5.6 Identifying Architecture Dependencies

We distinguish two types of dependencies. The first one is dependencies related to the variability at the requirement level (e.g., feature variability). For example, if a feature requires another feature, then the implementation component(s) of the first feature require the implementation component(s) of the second feature. These dependencies can be of five kinds: alternative, OR, AND, require, and exclude dependencies. The second one refers to dependencies related to the optional component distribution. These dependencies help to decide which optional component(s) is probably could be selected after the selection of another optional component(s)).

5.6.1 Identification of Dependencies Related to Feature Variability

To identify them, we rely on the same concept *Lattice* generated in the previous section. In the *Lattice*, each node groups a set of components representing the intent (e.g., *Com5* and *Com2*) and a set of architectural configurations representing the extent (e.g., *Arch2*). The configurations are represented by paths starting from their concepts to the *Lattice* concept root. The idea is that each object is generated starting from its node up going to the top. This is based on sub-concept to super-concept relationships (c.f. Section 5.2.2). This process generates a path for each object. A path contains an ordered list of nodes based on their hierarchical distribution; i.e., sub-concept to super-concept relationships).

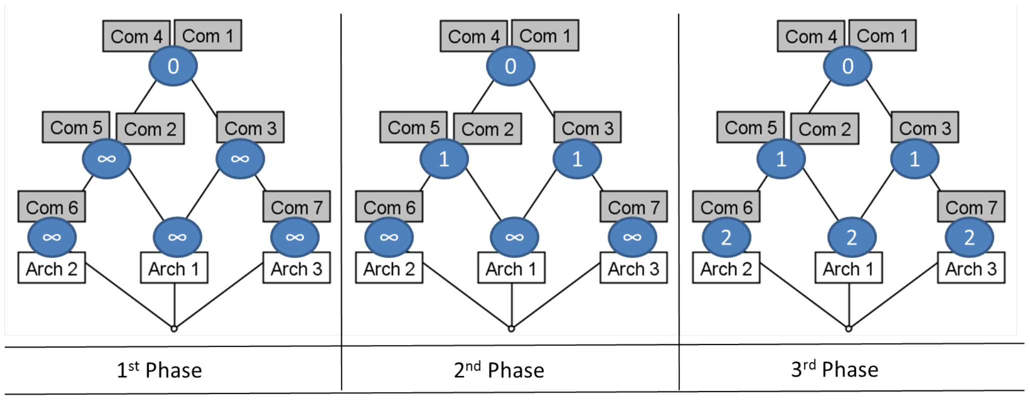


Figure 5.8: An example of BFS process

The extraction of these paths is based on two steps. The first one is node numbering. This is done using Breadth First Search (BFS) algorithm [Cormen 2009]. Since that BFS identifies the distance of such a node from another node, we use it to identify the order of the nodes in the paths. Starting from the root node (i.e., the top), BFS visits the nodes at distance *1*, then it visits the nodes at distance 2 and so on. Figure 5.8 shows the process of how BFS orders the nodes. In the second step, starting from a node that holds an extent (e.g., *Arch1*), we go up through links guiding us to nodes that carry a lower numbering and so on. This is recursively continued, where the termination condition is a reach of the node having *0* numbering. Figure 5.9 presents the paths identified in our example. There are three paths respectively presented by solid, dashed and double dashed arrows. For instance, the path corresponding to *Arch1* includes the node of *Com3*, the node of *Com5* and *Com 2* and the node of *Com1* and *Com4*.

According to these paths, we propose extracting the dependencies between each pair of nodes as follows:

5.6.1.1 Required Dependency

This constraint refers to the obligation selection of a component to select another one; i.e., *Component B* is required to select *Component A*. Based on the extracted

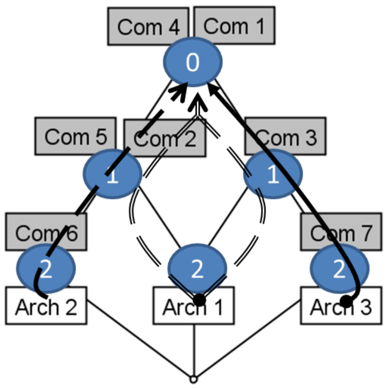


Figure 5.9: An example of paths extracted from FCA lattice

paths, we analyze their nodes by identifying parent-to-child relation (i.e., top to down). Thus node A requires node B if node B appears before node A in all path, i.e., node A is a sub-concept of the super-concept corresponding to node B. In other words, to reach node A in any path, it is necessary to traverse node B. For example, if we consider *Lattice* of the Figure 5.6, *Com6* requires *Com2* and *Com5* since *Com2* and *Com5* are traversed before *Com6* in all paths including *Com6* and linking root node to object nodes.

5.6.1.2 Exclude and Alternative Dependencies

Exclude dependency refers to the antagonistic relationship; i.e., *Component A* and *Component B* cannot occur in the same architecture. This relation is extracted by checking all paths. A node is excluded with respect to another node if they never appear together in any of the existing paths; i.e., there is no sub-concept to super-concept relationship between them. This means that there exists no object exists containing both nodes. For example, if we consider *Lattice* of Figure 5.6, *Com6* and *Com7* are exclusives since they never appear together in any of the *Lattice* paths. Algorithm 12 presents the procedure of extracting pairs of nodes that have the exclude dependency.

Alternative dependency generalizes the exclude one by exclusively selecting only one component from a set of components. It can be identified based on the exclude dependencies. Indeed, a set of nodes in the lattice having each an exclude constraint with all other nodes forms an alternative situation. For example, if node A is excluded with nodes B and C on the one hand, and node B is excluded with node C on the other hand, then the group of A, B and C forms an alternative situation.

5.6.1.3 AND Dependency

This is the bidirectional version of the required constraint; i.e., *Component A* requires *Component B* and vice versa. More generally, the selection of one component among a set of components requires the selection of all the other components. Ac-

Algorithm 12: Identifying Exclude Pairs

```
Input: All Pairs of Lattice Nodes and Paths($Pairs, Paths$)
Output: A Set of Pairs Having Exclude Dependency($ED$)
$ED = \emptyset$;
for each pair $\in$ Pairs do
    $isFound$ = false;
    for each path $\in$ Paths do
        if path.contains(pair) then
            $isFound$ = true;
            break;
    end
    if $isFound == false$ then
        $ED = ED \cup pair$ ;
end
return $ED$
```

cording to the built *Lattice*, this relation is found when a group of components is grouped in the same concept node in the *Lattice*; i.e., the whole node should be selected and not only a part of its components. For example if we consider *Lattice* of Figure 5.6, *Com2* and *Com5* are concerned with an AND dependency.

5.6.1.4 OR Dependency

When some components are concerned by an OR dependency, this means that at least one of them should be selected; i.e., the configuration may contain any combination of the components. Thus, in the case of absence of other constraints any pair of components is concerned by an OR dependency. Thus pairs concerned by required, exclude, alternative, or AND dependencies are ignored as well as those concerned by transitive require constraints; e.g., *Com6* and *Com7* are ignored since they are exclusives.

The process of identifying the OR groups is as follows. Firstly, we check the relationships between each pair of nodes. Pairs that have required, exclude, alternative, and AND relation are ignored. All pairs having transitive require constraints are also ignored. The reason of the exclusion is that these constraints break the OR one. Then, the remaining pairs of nodes are assigned an OR relation. Next, we analyze these pairs by testing the relation of their nodes. Pairs sharing a node need to be resolved (e.g., in Figure 5.6, a pair of (*Com5*-*Com2*, *Com7*) and a pair of (*Com5*-*Com2*, *Com3*), where *Com5*-*Com2* is a shared node). The resolution is based on the relation between the other two nodes (e.g., *Com3* and *Com7*). If these nodes have a require relation, then we select the highest node in the lattice (i.e., the parent causes the OR relation to its children). If the relation is excluded or alternative one, then we remove all OR relations (i.e., an exclude constraint violates an OR one). In the case of sharing an OR relation, the pairs are grouped to one OR relation. AND relation will not occur in this case according to AND definition.

Algorithm 13 shows the procedure of identifying groups of OR dependency.

Algorithm 13: Identifying OR-Groups

```
Input: all pairs (ap), require dependencies (rd), exclude dependencies (ed)
       and alternative dependencies (ad)
Output: sets of nodes having OR dependencies (orGroups)
OrDep = ap.exclusionPairs(rd, ed, ad);
OrDep = orDep.removeTransitiveRequire(rd);
ORPairsSharingNode = orDep.getPairsSharingNode();
for each $p \in ORPairsSharingNode$ do
    if otherNodes.getDependency() == require then
        orDep.removePair(childNode);
    else if otherNodes.getDependency()= exclude || alternative then
        orDep.removeAllPairs(p);
end
orGroups = orDep.getpairssharingOrDep();
return orGroups
```

5.6.2 Identification of Dependencies Related to Optional Component Distribution

The distribution of optional components consists of the identification of the association rules between them, and the ratio of component occurrences in the products (e.g., *Component A* has occurred in *80%* of products). Association rules refer to the frequency of co-occurrences between two groups of components. For example, if an architectural configuration contains *Component A* and *Component B*, it has a probability of *70%* of also containing *Component C* and *Component D*. We rely on FPGrowth algorithm [Han 2006] to mine the association rules. Each architectural configuration is considered as a transaction and each component is an item that can be existed in such a configuration. For instance, in Figure 5.6, we see that if a configuration contains *Com1*, *Com2*, and *Com4*, there is a *100%* probably that it also contains *Com5*. Additionally, Figure 5.6 shows, for instance, that *Com6* and *Com3* are present respectively in *33%* and *67%* of the configurations.

5.7 Identification of Groups of Variability

In the previous steps, mandatory and optional components, as well as the dependencies among them are identified. However, understanding a large number of dependencies is a challenge facing the software architect. Furthermore, some of these are overlapping dependencies. This means that many dependencies represent constraints on a shared set of components (e.g., a component has an OR relationship with components having AND relationships among them-self). Such dependencies need to be hierarchically represented in a tree form, in order to facilitate the task of the software architect (e.g., similar to feature model).

The idea behind that is to identify the variability among groups (i.e., identifying dependencies among groups of dependencies). For example, an OR relationship is existed among groups of alternative ones. The identification of these dependencies is as follows. As an AND can be considered as one entity, thus it is not allowed to their components to have internal dependencies (i.e., it is impossible to have another dependency, e.g., an OR, as a member in the AND). Thus the relation with AND is an external relation. For alternative and OR dependencies, it is allowed to take an AND relation as a member. In addition, the internal dependency between alternative and OR dependencies is allowed. In other words, an alternative dependency can take as a member an OR dependency and vice versa. According to that, the AND dependency have a high priority to be added before the others while OR and alternative have the same priority.

The construction of a hierarchical representation of the tree is as follows. First, we start from the root of the tree by directly connecting all mandatory components to it. At this stage, we do not have a hierarchy. Then, we add optional components based on their relationships. Groups of components having AND dependencies are added by creating an abstract node that carries out these components. The relation between the parent and the children is an AND dependency. Next, Alternative dependencies are represented by an abstract node that carries out these components. Then, OR relations are applied by adding an abstract node as a parent to components having OR relation. In the case where the relation is between a set of components having AND relation as well as alternative relation, the connection is made with their abstract node (i.e., the abstract node corresponding to the AND dependency as well as the alternative one becomes a child of the OR parent). Next, the remaining components are directly added to the root with optional notation. Finally, the cross-tree relations are added (i.e., required and exclude relations). Accordingly, we propose a procedure to identify the hierarchical tree. This procedure is presented in Algorithm 14.

5.8 Experimental Results and Evaluation

5.8.1 Experimentation Design

5.8.1.1 Data Collection

We select two sets of product variants. These sets are Mobile Media[1] (MM) and Health Watcher[2] (HW). We select these products because they were used in many research projects aiming at addressing the problem of migrating product variants into a SPL. Our study considers 8 variants of MM and 10 variants of HW. MM variants manipulate music, video and photo on mobile phones. They are developed

[1]Available at: http://ptolemy.cs.iastate.edu/design-study/#mobilemedia.

[2]Available at: http://ptolemy.cs.iastate.edu/design-study/#healthwatcher.

Algorithm 14: Identifying Hierarchically Representation

```
Input: Sets of Dependencies ($OR, AND, Require, Exclude, Alternative$)
       and Mandatory and Optional Components ($MC, OC$)
Output: A Tree ($tree$)
$tree$.root.addChildren($MC$);
for each $and \in AND$ do
    $tree$.addChild($and$);
end
for each $or \in OR$ do
    for each $node \in or$ do
        if $AND$.isContiant(node) then
            $tree$.remove($node$);
            $nodeOR$.addChildren($node$);
        else
            $nodeOR$.addChildren($node$);
        end
        $tree$.addChild($nodeOR$);
    end
end
for each $alt \in Alternative$ do
    for each $node \in alt$ do
        if $OR$.isContiant(node) then
            break;
        else if $AND$.isContiant(node) then
            $tree$.remove($node$);
            $nodeAlt$.addChildren($node$);
        else
            $nodeAlt$.addChildren($node$);
        end
        $tree$.addChild($nodeAlt$);
    end
end
$tree$.addChildren($OC$.getRemainingOptional());
$tree$.addExcludeCrossTree($Exclude$);
$tree$.addRequireCrossTree($Require$);
return $tree$
```

starting from the core implementation of MM. Then, the other features are added incrementally for each variant. HW variants are web-based applications that aim at managing health records and customer complaints. The size of each variant of MM and HW, in terms of classes, is shown in Table 5.3.

Table 5.3: Size of MM variants and HW ones

Name	1	2	3	4	5	6	7	8	9	10	Avg.
MM	25	34	36	36	41	50	60	64	X	X	43.25
HW	115	120	132	134	136	140	144	148	160	167	136.9

5.8.1.2 Research Questions and Evaluation Method

Our experimentation aims at showing how the proposed approach is applied to identify the architectural variability and validating the obtained results. To this end, we applied it on the collected case studies. We utilize *ROMANTIC* approach [Kebir 2012a] to extract architectural components from each variant independently. Then, the components derived from all variants are the input of the clustering algorithm to identify component variants. Next, we identify the architecture configurations of the products. These are used as a formal context to extract a concept lattice. Then, we extract the core (mandatory) and optional components as well as the dependencies among optional-component. Next, we present the results of identifying groups of variability. Finally, we show a model that contains all of the identified variability.

In order to evaluate the resulted architecture variability, we study the following research questions:

- **RQ1: Are the identified dependencies correct?** This research question goals at measuring the correctness of the identified component dependencies.
- **RQ2: What is the precision of the recovered architectural variability?** This research question focuses on measuring the precision of the resulting architecture variability. This is done by comparing it with a pre-existed architecture variability model.

5.8.2 Results

5.8.2.1 Component Based Architecture Extraction

Table 5.4 shows the results of component extraction from each variant independently, in terms of the number of components, for each variant of MM and HW. The results show that classes related to the same functionality are grouped into the same component. The difference in the numbers of the identified components in each variant has resulted from the fact that each variant has a different set of user's requirements. On average, a variant contains 6.25 and 7.7 main functionalities respectively for MM and HW.

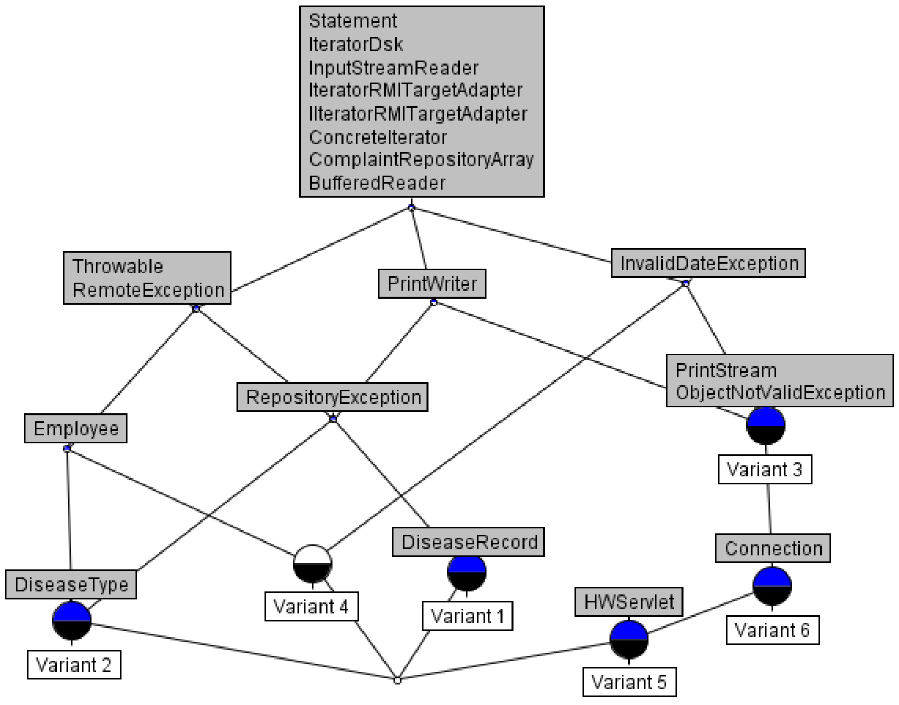


Figure 5.10: The distribution of classes composing the component variants

Table 5.4: Component extraction results

Name	1	2	3	4	5	6	7	8	9	10	Avg.	Total
MM	3	5	5	5	7	7	9	9	X	X	6.25	50
HW	6	7	9	10	7	9	8	8	7	6	7.7	77

5.8.2.2 Identifying Component Variants

Table 5.5 summarizes the results of component variants in terms of the number of components having variants (NOCV), the average number of variants of a component (ANVC), the maximum number of component variants (MXCV) and the minimum number of component variants (MNCS). The results show that there are many sets of components sharing the most of their classes. Each set of components mostly provides the same functionality. Thus, they represent variants of the same architectural component. Table 5.6 presents an instance of 6 component variants identified from HW, where X means that the corresponding class is a member in the variant. By analyzing these variants, it is clear that these components represent the same architectural component. In addition to that, we noticed that there are some component variants having the same set of classes in multiple product variants. For internal component variability, we provide the *Lattice* that presents the distribution of classes composing the component variants, see Figure 5.10. From this *Lattice*, we can note that there are *8* common classes between the variants. These represent the implementation of shared services. Additionally, the distribution of variable classes is easy to recognize. For example, the difference between *Variant3* and *Variant6* is that *Variant6* has an additional variable class (i.e., *Connection*).

Table 5.5: Component variants identification

Name	NOCV	ANVC	MXCV	MNCV
MM	14	3.57	8	1
HW	18	4.72	10	1

Table 5.6: Instance of 6 component variants

Class Name	Variant 1	Variant 2	Variant 3	Variant 4	Variant 5	Variant 6
BufferedReader	X	X	X	X	X	X
ComplaintRepositoryArray	X	X	X	X	X	X
ConcreteIterator	X	X	X	X	X	X
DiseaseRecord	X					
IIteratorRMITargetAdapter	X	X	X	X	X	X
IteratorRMITargetAdapter	X	X	X	X	X	X
DiseaseType		X				
InputStreamReader	X	X	X	X	X	X
Employee		X		X		
InvalidDateException			X	X	X	X
IteratorDsk	X	X	X	X	X	X
PrintWriter	X	X	X		X	X
ObjectNotValidException			X		X	X
RemoteException	X	X		X		
PrintStream			X		X	X
RepositoryException	X	X				
Statement	X	X	X	X	X	X
Throwable	X	X		X		
HWServlet					X	
Connection					X	X

5.8.2.3 Analyzing Architecture Configuration: Communality and Variability

The identification of component variants allows us to identify the architecture configurations. Table 5.7 and Table 5.8 show respectively the configuration of MM and HW variants, where *X* means that the component is a part of the product variants. The results show that the products are similar in their architectural configurations and differ considering other ones. The reason behind the similarity and the difference is the fact that these products are common in some of their user's requirements and variable in some others. These architecture configurations are used as a formal context to extract the concept lattice. We use the Concept Explorer[3] tool to generate the concept lattice. We give the concept lattices of MM and HW respectively in Figure 5.11 and Figure 5.12.

In Table 5.9, the numbers of core (mandatory) and optional components are given for MM and HW, together with examples of their association rules and component occurrence ratios. An example of an association rule in MM is "when a configuration has *Com5* and *Com8*, it is *86%* likely to also contain *Com9*". The results show that there are some components that represent the core architecture, while some others represent delta (optional) components.

Table 5.7: Architecture configuration for all MM variants

Variant No.	Com1	Com2	Com3	Com4	Com5	Com6	Com7	Com8	Com9	Com10	Com11	Com12	Com13	Com14
1	X		X		X									
2	X		X		X			X	X					
3	X		X		X			X	X					
4			X		X			X	X		X			
5	X		X		X			X	X		X	X		
6			X		X		X	X	X		X	X		
7		X		X	X		X	X	X	X	X	X		
8		X		X	X	X	X	X		X			X	X

Table 5.8: Architecture configuration for all HW variants

Variant No.	Com1	Com2	Com3	Com4	Com5	Com6	Com7	Com8	Com9	Com10	Com11	Com12	Com13	Com14	Com15	Com16	Com17	Com18
1		X	X	X												X	X	X
2	X	X	X	X	X	X	X											
3	X	X	X	X	X	X	X	X	X									
4	X	X	X	X		X	X	X	X	X	X							
5		X	X	X		X	X				X	X						
6		X	X	X	X	X	X	X	X		X							
7		X	X	X	X	X			X	X	X							
8		X	X		X	X		X		X	X		X					
9		X	X		X	X		X			X			X				
10		X	X			X					X			X	X			

[3]Presentation of the Concept Explorer tool is available in [Yevtushenko 2000]

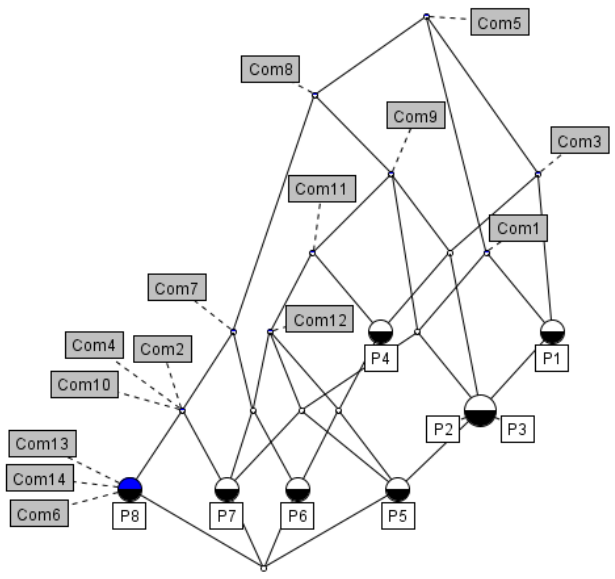


Figure 5.11: The concept lattice of MM architecture configurations

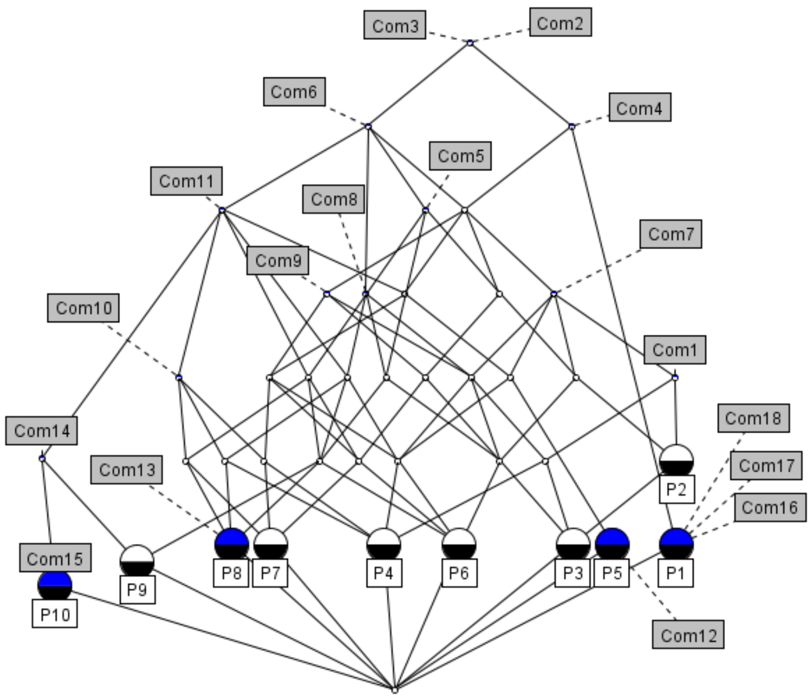


Figure 5.12: The concept lattice of HW architecture configurations

Table 5.9: Mandatory and optional components

Product Name	MM		HW	
Mandatory	1		2	
Optional	13		16	
Some Association Rules	Com5, Com8 =>Com9 : 86%		Com2, Com3, Com11 =>Com6 : 100%	
	Com1, Com5 =>Com8. Com9 : 80%		Com2, Com3, Com4 =>Com6 : 86%	
	Com5, Com12 =>Com8, Com9, Com11 : 100%		Com2, Com3, Com12 =>Com4, Com6, Com7, Com11 : 100%	
Some Component Occurrence Ratio	Component	Ratio	Component	Ratio
	Com8	80%	Com6	90%
	Com12	38%	Com7	50%
	Com4	12%	Com14	20%

5.8.2.4 Identifying Components Dependencies

The results of the identification of optional-component dependencies are given in Table 5.10 and Table 5.11 respectively for MM and HW (*Com 5* from MM and *Com2, Com3* from HW are excluded since they are mandatory components). The dependencies are represented between all pairs of components in MM (where R= Require, E= Exclude, O= OR, RB = Required By, TR = Transitive Require, TRB = Transitive Require By, and A = AND). Table 5.12 shows a summary of MM and HW dependencies between all pairs of components. This includes the number of direct require constrains (NRC), the number of exclude ones (NE), the number of AND groups (NOA), and the number of OR groups (NO). Alternative constrains is represented as exclude ones. The results show that there are dependencies among components that help the architect to avoid creating invalid configuration. For instance, a design decision of AND components indicates that these components depend on each other, thus, they should be selected all together.

Table 5.10: Component dependencies of MM

	Com1	Com2	Com3	Com4	Com6	Com7	Com8	Com9	Com10	Com11	Com12	Com13	Com14
Com1	X		R		E	E				O	E	E	E
Com2		X	E	A	RB	R	TR		A			RB	RB
Com3	RB	E	X	E	E		O		E			E	E
Com4		A	E	X	RB	R	TR		A			RB	RB
Com6	E	R	E	R	X	TR	TR	E	R	E	E	A	A
Com7	E	RB		RB	TRB	X	R	O	RB			TRB	TRB
Com8		TRB	O	TRB	TRB	RB	X	RB	TRB	TRB	TRB	TRB	TRB
Com9					E	O	R	X		RB	TRB	E	E
Com10		A	E	A	RB	R	TR		X			RB	RB
Com11	O				E		TR	R		X	RB	E	E
Com12	E				E		TR	TR		R	X	E	E
Com13	E	R	E	R	A	TR	TR	E	R	E	E	X	A
Com14	E	R	E	R	A	TR	TR	E	R	E	E	A	X

5.8.2.5 Identifying Groups of Variability

After the identification of mandatory and optional components as well as the dependencies among the optional ones, the elements of AVM are extracted. Based on

Table 5.11: Component dependencies of HW

	Com1	Com4	Com5	Com6	Com7	Com8	Com9	Com10	Com11	Com12	Com13	Com14	Com15	Com16	Com17	Com18
Com1	X	TR		TR	R					ALT	ALT	E	ALT	ALT	ALT	ALT
Com4	TRB	X		O	RB		RB			TRB	E	E	E	RB	RB	RB
Com5			X	R	O	O	O		O	E	TRB		E	E	E	E
Com6	TRB	O	RB	X	RB	RB	RB	TRB	RB	TRB	TRB	TRB	TRB	E	E	E
Com7	RB	R	O	R	X	O	O		O	RB	E	E	E	E	E	E
Com8			O	R	O	X	O		O	E	RB		E	E	E	E
Com9		R	O	R	O	O	X		O	E	E	E	E	E	E	E
Com10				TR				X	R	E	RB	E	E	E	E	E
Com11			O	R				RB	X	RB	TRB	RB	TRB	E	E	E
Com12	ALT	TR	E	TR	R	E	E	E	R	X	ALT	E	ALT	ALT	ALT	ALT
Com13	ALT	E	TR	TR	E	R	E	R	TR	ALT	X	E	ALT	ALT	ALT	ALT
Com14	E	E		TR	E		E	E	R	E	E	X	RB	E	E	E
Com15	ALT	E	E	TR	E	E	E	E	TR	ALT	ALT	R	X	ALT	ALT	ALT
Com16	ALT	R	E	E	E	E	E	E	E	ALT	ALT	E	ALT	X	A	A
Com17	ALT	R	E	E	E	E	E	E	E	ALT	ALT	E	ALT	A	X	A
Com18	ALT	R	E	E	E	E	E	E	E	ALT	ALT	E	ALT	A	A	X

Table 5.12: Summarization of MM and HW dependencies

Name	NDR	NE	NA	NO
MM	17	20	6	3
HW	18	62	3	11

these elements, the AVM is built. We use the FeatureIDE[4] tool to visualize the tree on form of feature model. Figure 5.13 shows the trees of both MM and HW, where $Com_i => Com_j$ refers to a required constrain, and $\neg\ (Com_i \wedge Com_j)$ refers to an exclude constraint.

5.8.2.6 Final Results

To the best our knowledge, there is no architecture description language supporting all kinds of the identified variability. The existing languages, like [Hendrickson 2007], are mainly focused on modeling component variants, links and interfaces, while they do not support dependencies among components such as AND-group, OR-group, and require. Thus, on the first hand, we use notations presented in [Hendrickson 2007] to represent the concept of component variants and links variability. On the other hand, we propose notations inspired from feature modeling languages to model the dependencies among components. For the purpose of understandability, we document the resulting components by assigning a name based on the most frequent tokens in their classes' names. Figure 5.14 shows the architectural variability model identified for MM variants, where the large boxes denote to design decisions (constraints). For instance, core architecture refers to components that should be selected to create any concrete product architecture. In MM, there is one core components manipulating the base controller of the product. This component

[4]Presentation of the FeatureIDE tool is available in [Thüm 2014]

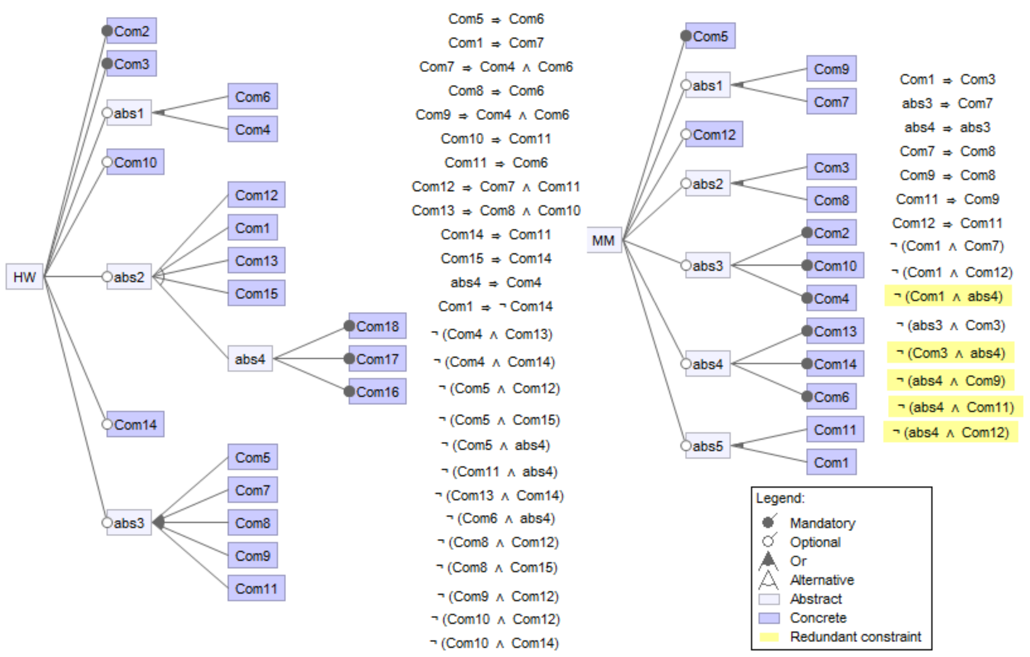


Figure 5.13: The architecture variability trees of MM and HW

has two variants. A group of *Multi Media Stream*, *Video Screen Controller*, and *Multi Screen Music* components represents an AND design decision.

5.8.3 Answering Research Questions

5.8.3.1 RQ1: Are the identified dependencies correct?

The identification of component dependencies is based on the occurrence of components. e.g., if two components never selected to be included in a concrete product architecture, we consider that they hold an exclude relation. However, this method could provide correct or incorrect dependencies. To evaluate the accuracy of this method, we manually validate the identified dependencies. This is based on the functionalities provided by the components. For instance, we check if the component functionality requires the functionality of the required component and so on. The results show that 79% of the required dependencies are correct. As an example of a correct relation is that *SMS Controller* requires *Invalid Exception* as it performs an input/output operations. On the other hand, it seems that *Image Util* does not require *Image Album Vector Stream*. Also, 63% of the exclude constrains are correct. For AND and OR dependencies, we find that 88% of AND groups are correct, while 42% of OR groups are correct. Thus, the precision of identifying dependencies is 68% in average.

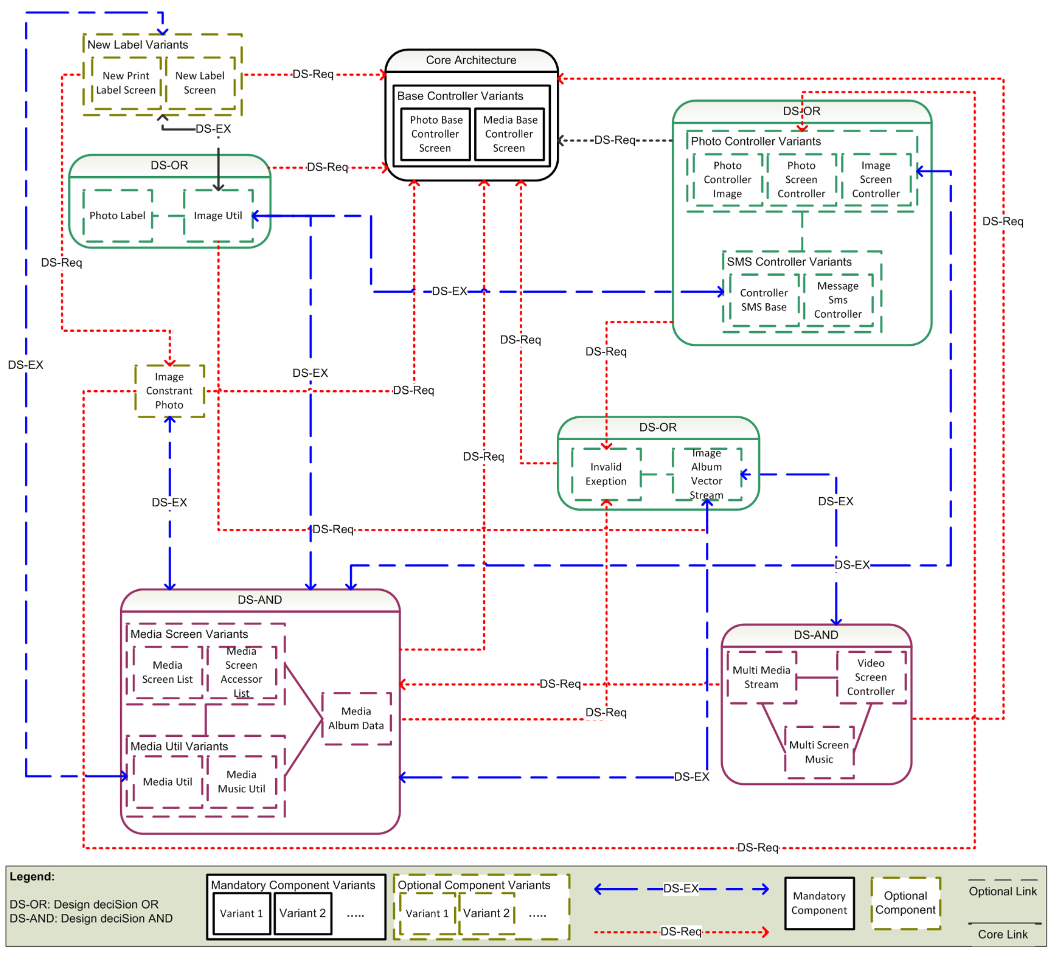


Figure 5.14: Architectural variability model for MM

5.8.3.2 RQ2: What is the precision of the recovered architectural variability?

In our case studies, MM is the only case study that has an available architecture model containing some variability information. In [Figueiredo 2008], the authors presented the aspect oriented architecture for MM variants. This contains information about which products had added components, as well as in which product a component implementation was changed (i.e., component variants). We manually compare both models to validate the resulting model. Figure 5.15 shows the comparison results in terms of the total number of components in the architecture model (TNOC), the number of components having variants (NCHV), the number of mapped components in the other model (NC), the number of unmapped components in the other model (NUMC), the number of optional components (NOC) and the number of mandatory ones (NOM). The results show that there are some variation between the results of our approach and the pre-existed model. The reason behind this variation is the idea of compositional components. For instance, our approach identifies only one core component compared to 4 core components in the other model. Our approach grouped all classes related to the controller components together in one core components. On the other hand, the other model divided the controller component into *Abstract Controller*, *Album Data*, *Media Controller*, and *Photo View Controller* components. In addition, the component related to handling exceptions is not mentioned in the pre-existed model at all.

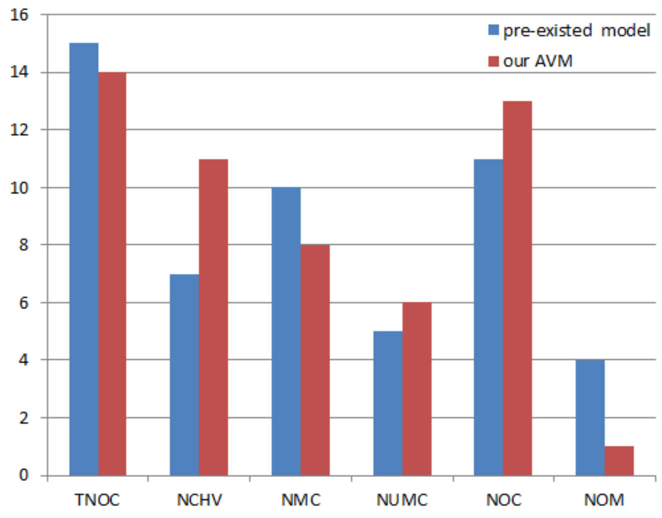


Figure 5.15: The results of the MM validation

5.9 Threats to Validity

Two types of threats to validity concern the proposed approach. These are internal and external.

5.9.1 Threats to Internal Validity

There are five aspects to be considered regarding the internal validity. These are as follows:

1. We identify component variants based on the textual similarity between the classes composing the components. While in some situations, a set of components may be implemented through sets of classes that are textual similar, but they are completely unrelated. However, in the case of product variants that is developed by copy-paste-modify technique, the modification is mainly composed of method overriding, adding or deleting, but the main functionalities are still the same ones.

2. Dependencies among components are identified based on component occurrences in the product architectures. Thus, the identified dependencies maybe correct or incorrect. However the accuracy of the approach is increased as much as the number of product variants is increased.

3. The input of our approach is the components independently identified form each product variants using *ROMANTIC* approach. Thus. the accuracy of the obtained variability depends on the accuracy of *ROMANTIC* approach.

4. We manually evaluate the research questions. For the first question, we check the component functionalities to evaluate the identified dependencies. For the second question, the identified architecture variability is also manually compared to the existing model. However this should be done by the domain experts to be precise.

5. Similar to the approach presented in Chapter 3, we rely on the static analysis to identify dependencies between the classes (please refer to the first point of Section 3.7.1).

5.9.2 Threats to External Validity

There are two aspects to be considered regarding the external validity. These are as follows:

1. The proposed approach is experimented through product variants that are implemented by *Java* programming language. However the obtained results can be generalized for any object-oriented language. The reason behind this generalization is the fact that all object-oriented languages (e.g., *C++* and *C#*) are structured in terms of classes and their relationships are realized through method calls, access attributes, etc.

2. In the experimentation, only two case studies have been collected; Mobile Media and Health Watcher. However these are used in several research projects that address the problem of migrating products variants into software product line. On average, the selected case studies obtained the same results. Thus these results can be generalized for other similar case studies. By the way, the approach needs to be validated with a large number of case studies. This will be a logical extension of our work.

5.10 Conclusion

In SPLA, the variability is mainly represented in terms of components and configurations. In the case of migrating product variants to a SPL, identifying the architecture variability among the product variants is necessary to facilitate the software architect's tasks. Thus, in this chapter, we proposed an approach to recover the architecture variability of a set of product variants.

The recovered variability includes mandatory/optional components, the dependencies among optional components (e.g., require, etc.), the variability of component-links, and component variants. We rely on FCA to analyze the variability. Then, we propose two heuristics. The former is to identify the architecture variability. The latter is to identify the architecture dependencies. The proposed approach is validated through two sets of product variants derived from Mobile Media and Health Watcher. The results show that our approach is able to identify the architectural variability and the dependencies as well.

Chapter 6

Conclusion and Future Direction

Contents

The ultimate goal of this dissertation is to support systematic software reuse. Towards this goal, we propose to reverse engineering some core assets related to CBSE and SPLE by analyzing existing object-oriented software. To do so, we address the following research problems:

1. **Reverse engineering software components by analyzing existing object-oriented software:** the identification consists of analyzing the source code, in order to extract a piece of code (i.e., cluster of classes) that can constitute the implementation components.

2. **Recovering SPLA by analyzing existing software product variants:** in the case of migrating product variants to a SPL, identifying the architecture variability among the product variants is necessary to facilitate the software architect's tasks. SPLA is recovered based on the analysis of the source code of the existing product variants.

6.1 Summary of Contributions

The contributions of this dissertation are:

1. We proposed an approach that aims at mining reusable components from a set of similar object-oriented software product variants. The main idea is to analyze the commonality and the variability of product variants, in order to identify pieces of code that may form reusable components. Our motivation is that components mined based on the analysis of several existing software products will be more useful (reusable) for the development of new software products than those mined from singular ones. During this contribution, we mainly answered the following research questions:

- What is a component compared to an object?
- How to identify potential components constituting a software application?
- How to identify components providing similar functionalities?
- Which component to be extract from a group of similar ones?
- How to validate the reusability of the mined components?

To validate our approach, we apply it onto two families of open-source product variants. We propose an empirical measurement to evaluate the reusability of the mined components. According to this measurement, the results show that the reusability of the components mined based on our approach is better than the reusability of those mined from singular software.

2. We proposed an approach that aims at reverse engineering component-based APIs from object-oriented APIs. This approach exploits specificity of API entities by statically analyzing the source code of both APIs and their software clients to identify groups of API classes that are able to form components. This is based on two criteria. The first one is the probability of classes to be reused together by API clients. The second one is related to the structural dependencies among classes and thus their ability to form a quality-centric component. The component identification process is used-driven. This means that components are identified starting from classes composing their interfaces. Classes composing the provided interface of the first layer components compose FUPs. Then, the API is organized by a set of layers, where each layer composes of components providing services to the others composing the above layer, and so on. During this contribution, we answered the following research questions:

   - How to identify components from object-oriented APIs?
   - What are the limitation of existing approaches?
   - What are dependencies that can be existed between API classes?
   - How to invest the way that software applications used API classes to identify components?
   - How to identify frequent usage patterns of classes?
   - How to map frequent usage patterns to component interfaces?
   - How to identify classes composing the component based on the interface classes?
   - How components are organized in the component-based API?
   - How to validate reusability and understandability of the resulting component-based APIs?

   In order to validate the presented approach, we experimented it by applying on a set of 100 open source *Java* applications as clients for three *Android* APIs.

The validation is done through three research questions. The first one is related to the reusability of the mined components, while the second indicates to the understandability of the resulting component-based APIs. The results show that our approach improves the reusability and the understandability of the API. The last research question aims to measure how much benefit the use of FUPs brings. To this end, we compare our approach with a traditional component identification approach that is designed for software applications. The results prove that our approach outperforms the traditional one.

3. We proposed an approach that aims at automatically recovering the architecture of a set of software product variants. This is done through the exploitation of the commonality and the variability across the source code of product variants. Our contribution is twofold: the identification of architecture variability concerning both component and configuration variabilities on the one hand, and the identification of architecture dependencies existed between the architectural elements on the other hand. The latter is done by capturing the commonality and the variability at the architectural level using formal concept analysis. During this contribution, we answered the following research questions:

   - What is architecture variability?
   - How to identify component variability?
   - How to recover configuration variability?
   - How to identify dependencies existing between components using FCA?
   - How to validate the recovered SPLA?

   In order to validate the proposed approach, we experimented on two families of open-source product variants. The approach was validated using twos research questions. The first one measures the precision of the recovered architectural variability. The second one evaluates the identified architectural dependencies. The experimental evaluation shows that our approach is able to identify the architectural variability and the dependencies as well.

6.2 General Limitations

The main limitations related to the presented approaches are:

1. We relied on static analysis techniques to analyze dependencies between object oriented classes. This means that our approaches suffer two main limitations. The first one is that they do not address polymorphism and dynamic binding dependencies. However, in object-oriented, the most important dependencies are realized through method calls and access attributes. Thus the ignorance of polymorphism and dynamic binding has not a high impact on the performance

of our approaches. The second one is that they do not differ from the used and unused source code. This may provide a noise dependencies by taking into account unused source codes. However, this situation rarely exists in the case of well designed and implemented software. In contrast, dynamic analysis addresses all of these limitations. But the challenge with dynamic analysis is to identify all use cases of software.

2. We use cluster algorithms to group similar components and classes into disjoint clusters. However this provides a near optimal solution of the partitioning. Other grouping techniques may provide more accurate solutions, such as search-based algorithms. This will be a future extension of our approach to investigate simulating annulling and genetic algorithms.

3. Our approaches are experimented via software systems (e.g. product variants, APIs and API clients)that were implemented using *Java* programming language. Other object-oriented languages (e.g., *C++* and *C#*) have not been tested. To prove that our approaches work also for all object-oriented software systems. We plan to apply them on case studies developed using *C++* and *C#*.

4. In some cases, we need software architects to provide threshold values.

6.3 Future Directions

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

6.3.1 Addressing New Related Aspects:

1. **Migrating the identified object-oriented components into existing component models.** The presented approaches identify a component as a cluster of object-oriented classes representing the implementation of this component. This constitutes the first step of the reengineering process of object-oriented software into component-based software. Thus we plan to extend the approaches by transforming the object-oriented implementation of the identified components into an equivalent component-based one, such as OSGI. For example, we propose to address problems related to the transformation of object dependencies to component interfaces such as inheritance and exception handling.

2. **Reverse engineering software architectures and components based on dynamic analysis.** To address the static analysis techniques, we plan to extend the approaches by analyzing object-oriented software based on dynamic analysis. For example, we propose to dynamically analyze the API client applications by identifying execution traces corresponding to the use cases.

This allows to identify the internal dependencies between classes used by client applications and other ones composing the API.

3. **Mapping the requirement variability and the architecture variability.** In the case of reengineering a SPL from product variants, mapping the identified architectural variability with the requirement variability is an important task. Thus we plan to extend the results of SPLA identification by mapping them to the feature model which realized the variability at the requirement level. For example, we propose to identify component(s) implementing given feature(s).

6.3.2 Tools Support and Experimentations

1. **Developing a visual environment.** The presented approaches can be extended by providing a visual environment, such that domain experts are allowed to interact with the approaches at each step of the identification process, and modify the obtained results as needed. To do so, we propose to develop an Eclipse plug-in that implements our approaches.

2. **Experimenting with large number of case studies.** The performance of our approaches is based on the number of case studies. This refers to the number of the product variants for the approaches presented to analyze product variants and the number of API clients for the approach presented to analyze APIs. Thus we plan extending the evaluation of the proposed approaches by conducting more case studies which provides better test of the approaches and generalizes the results as well.

3. **Validating the approaches by human experts.** The results of the presented approaches are validated based on heuristic measurements that we proposed. However this is considered as an approximation validation. To better validate the approaches, we plan to validate the results based on human experts. This can be realized by giving our results to a set of software developers and asking them to build a new software applications.

Publications

The PhD was started in September 2012. During this period, we have published the following research papers published in international conferences:

- *Recovering Architectural Variability of a Family of Product Variants* - Anas Shatnawi, Abdelhak Seriai, Houari A. Sahraoui. 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: 17-33. (Selected by ICSR 2015 to be extended for a special issue of JSS "Journal of System and Software").

- *Mining Software Components from Object-Oriented APIs* - Anas Shatnawi, Abdelhak Seriai, Houari A. Sahraoui, Zakarea Al-Shara. 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. (Selected by ICSR 2015 to be extended for a special issue of JSS "Journal of System and Software").

- *Mining Reusable Software Components from Object-Oriented Source Code of a Set of Similar Software* - Anas Shatnawi, Abdelhak-Djamel Seriai. In IEEE 14th International Conference on Information Reuse & Integration, IRI 2013, San Francisco, CA, USA, August 14-16, 2013. IEEE 2013: 193-200.

- *Service Identification Based on Quality Metrics : Object-Oriented Legacy System Migration Towards SOA* - Seza Adjoyan, Abdelhak-Djamel Seriai, Anas Shatnawi. In The 26th International Conference on Software Engineering and Knowledge Engineering, SEKE 2014, Vancouver, BC, Canada, July 1-3, 2013. Knowledge Systems Institute Graduate School 2014: 1-6.

Bibliography

[Acharya 2007] Mithun Acharya, Tao Xie, Jian Pei and Jun Xu. *Mining API patterns as partial orders from source code: from usage scenarios to specifications.* In Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 25–34. ACM, 2007. (Cited on page 60.)

[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 29.)

[Allier 2009] Simon Allier, Houari A Sahraoui and Salah Sadou. *Identifying components in object-oriented programs using dynamic analysis and clustering.* In Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research, pages 136–148. IBM Corp., 2009. (Cited on pages 19, 20, 21, 23, 24, 25 and 26.)

[Allier 2010] Simon Allier, Houari A Sahraoui, Salah Sadou and Stéphane Vaucher. *Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces.* In Component-Based Software Engineering, pages 216–231. Springer, 2010. (Cited on pages 20, 21, 23, 24, 25 and 26.)

[Allier 2011] Simon Allier, Salah Sadou, Houari Sahraoui and Régis Fleurquin. *From Object-Oriented Applications to Component-Oriented Applications via Component-Oriented Architecture.* In 2011 9th Working IEEE/IFIP Conference on Software Architecture (WICSA), pages 214–223. IEEE, 2011. (Cited on pages 7, 12, 18, 20, 21, 23, 24, 25, 26, 34 and 60.)

[Bachmann 2000] Felix Bachmann, Len Bass, Jeromy Carriere, Paul C Clements, David Garlan, James Ivers, Robert Nord and Reed Little. *Software architecture documentation in practice: Documenting architectural layers.* 2000. (Cited on page 18.)

[Bass 2012] Len Bass, Paul Clements and Rick Kazman. Software architecture in practice. Addison-Wesley Professional, 3rd édition, 2012. (Cited on page 13.)

[Baster 2001] Greg Baster, Prabhudev Konana and Judy E. Scott. *Business Components: A Case Study of Bankers Trust Australia Limited.* Commun. ACM, vol. 44, no. 5, pages 92–98, May 2001. (Cited on page 15.)

[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 pages 29, 46 and 71.)

[Birkmeier 2009] Dominik Birkmeier and Sven Overhage. *On component identification approaches–classification, state of the art, and comparison.* In Component-Based Software Engineering, pages 1–18. Springer, 2009. (Cited on page 34.)

[Boussaidi 2012] G.E. Boussaidi, A.B. Belle, S. Vaucher and H. Mili. *Reconstructing Architectural Views from Legacy Systems.* In 2012 19th Working Conference on Reverse Engineering (WCRE), pages 345–354, Oct 2012. (Cited on pages 20, 21, 23, 24, 25 and 26.)

[Cai 2000] Xia Cai, Michael R Lyu, Kam-Fai Wong and Roy Ko. *Component-based software engineering: technologies, development frameworks, and quality assurance schemes.* In Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific, pages 372–379. IEEE, 2000. (Cited on page 6.)

[Canal 1999] Calos Canal, Ernesto Pimentel and José M Troya. *Specification and refinement of dynamic software architectures.* In Software Architecture, pages 107–125. Springer, 1999. (Cited on page 93.)

[Chardigny 2008a] 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 pages 18 and 29.)

[Chardigny 2008b] 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 16, 20, 23, 25, 26, 27 and 29.)

[Chardigny 2009] Sylvain Chardigny. *Extraction d'une architecture logicielle à base de composants depuis un système orienté objet. Une aproche par exploration.* PhD thesis, Université de Nantes, 2009. (Cited on pages 19, 35 and 37.)

[Chardigny 2010] Sylvain Chardigny and Abdelhak Seriai. *Software architecture recovery process based on object-oriented source code and documentation.* In Software Architecture, pages 409–416. Springer, 2010. (Cited on pages 18, 19, 20, 22, 25 and 26.)

[Chihada 2015] Abdullah Chihada, Saeed Jalili, Seyed Mohammad Hossein Hasheminejad and Mohammad Hossein Zangooei. *Source code and design* *conformance, design pattern detection from source code by classification approach.* Applied Soft Computing, vol. 26, pages 357–367, 2015. (Cited on pages 20, 21, 25 and 26.)

[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 12 and 13.)

[Clements 2002] P. Clements and L. Northrop. *Software product lines: practices and patterns.* 2002. (Cited on pages 5, 6, 14, 90 and 91.)

[Constantinou 2011] Eleni Constantinou, George Kakarontzas and Ioannis Stamelos. *Towards open source software system architecture recovery using design metrics.* In 2011 15th Panhellenic Conference on Informatics (PCI), pages 166–170. IEEE, 2011. (Cited on pages 18, 20, 22, 25 and 26.)

[Cormen 2009] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest and Clifford Stein. Introduction to algorithms. MIT press, 2009. (Cited on pages 20, 21 and 101.)

[Couto 2011] Marcus Vinicius Couto, Marco Tulio Valente and Eduardo Figueiredo. *Extracting software product lines: A case study using conditional compilation.* In 15th European Conference on Software Maintenance and Reengineering (CSMR2011), pages 191–200. IEEE, 2011. (Cited on page 49.)

[DeBaud 1998] Jean-Marc DeBaud, Oliver Flege and Peter Knauber. *PuLSE-DSSA-a method for the development of software reference architectures.* In Proceedings of the third international workshop on Software architecture, pages 25–28. ACM, 1998. (Cited on pages 6 and 14.)

[Demeyer] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz. Object-oriented reengineering patterns. (Cited on page 12.)

[Dubinsky 2013] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker and Krzysztof Czarnecki. *An exploratory study of cloning in industrial software product lines.* In Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, pages 25–34. IEEE, 2013. (Not cited.)

[Ducasse 2009] 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 pages 23 and 25.)

[Dugerdil 2013] Philippe Dugerdil and David Sennhauser. *Dynamic Decision Tree for Legacy Use-case Recovery.* In Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, pages 1284–1291, New York, NY, USA, 2013. ACM. (Cited on pages 19, 20, 21, 24, 25 and 26.)

[Elhaddad 2012] Younis R Elhaddad. *Combined Simulated Annealing and Genetic Algorithm to Solve Optimization Problems.* In World Academy of Science, Engineering and Technology (WASET), 2012. (Cited on page 21.)

[Erdemir 2011] Ural Erdemir, Umut Tekin and Feza Buzluca. *Object Oriented Software Clustering Based on Community Structure.* In 2011 18th Asia Pacific Software Engineering Conference (APSEC), pages 315–321. IEEE, 2011. (Cited on pages 19, 20, 21, 22, 23, 25 and 26.)

[Ferreira da Silva 1996] M Ferreira da Silva and Claudia Maria Lima Werner. *Packaging reusable components using patterns and hypermedia.* In Software Reuse, 1996., Proceedings Fourth International Conference on, pages 146–155. IEEE, 1996. (Cited on page 5.)

[Figueiredo 2008] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Francisco Dantas*et al. Evolving software product lines with aspects.* In ACM/IEEE 30th International Conference on Software Engineering (ICSE'08), pages 261–270. IEEE, 2008. (Cited on pages 49 and 115.)

[Filman 2004] Robert Filman, Tzilla Elrad, Siobhán Clarke*et al.* Aspect-oriented software development. Addison-Wesley Professional, 2004. (Cited on page 5.)

[Fischer 2014] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon and Alexander Egyed. *Enhancing clone-and-own with systematic reuse for developing software variants.* In Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, pages 391–400. IEEE, 2014. (Not cited.)

[Frakes 1996] William Frakes and Carol Terry. *Software Reuse: Metrics and Models.* ACM Comput. Surv., vol. 28, no. 2, pages 415–435, June 1996. (Cited on page 5.)

[Frakes 2005] William B Frakes and Kyo Kang. *Software reuse research: Status and future.* IEEE transactions on Software Engineering, vol. 31, no. 7, pages 529–536, 2005. (Cited on pages 5 and 7.)

[Frenzel 2007] Pierre Frenzel, Rainer Koschke, Andreas PJ Breu and Karsten Angstmann. *Extending the reflexion method for consolidating software variants into product lines.* In 14th Working Conference on Reverse Engineering (WCRE), pages 160–169. IEEE, 2007. (Cited on pages 8, 20, 22, 23, 24, 25, 26, 31 and 90.)

[Ganter 2012] Bernhard Ganter and Rudolf Wille. Formal concept analysis: mathematical foundations. Springer Science & Business Media, 2012. (Cited on pages 21, 91 and 92.)

[Garlan 2000] David Garlan. *Software Architecture: A Roadmap.* In Proceedings of the Conference on The Future of Software Engineering, ICSE '00, pages 91–101, New York, NY, USA, 2000. ACM. (Cited on pages 15 and 16.)

[Gasparic 2014] Marko Gasparic, Andrea Janes, Alberto Sillitti and Giancarlo Succi. *An Analysis of a Project Reuse Approach in an Industrial Setting.* In Software Reuse for Dynamic Systems in the Cloud and Beyond, pages 164–171. Springer, 2014. (Cited on pages 6, 7 and 34.)

[Gomaa 2005] H. Gomaa. *Designing Software Product Lines with UML.* In Software Engineering Workshop - Tutorial Notes, 2005. 29th Annual IEEE/NASA, pages 160–216, April 2005. (Cited on page 14.)

[Google 2015] Google. *API Guides (http://developer.android.com/reference/packages.html)*, 2015. (Cited on page 60.)

[Griss 1997] Martin L Griss. *Software reuse architecture, process, and organization for business success.* In Computer Systems and Software Engineering, 1997., Proceedings of the Eighth Israeli Conference on, pages 86–89. IEEE, 1997. (Cited on page 5.)

[Hamza 2009] Haitham S Hamza. *A Framework for Identifying Reusable Software Components Using Formal Concept Analysis.* In Sixth International Conference on Information Technology: New Generations (ITNG), 2009, pages 813–818. IEEE, 2009. (Cited on pages 18, 20, 21, 23, 25 and 26.)

[Han 2000] Jiawei Han, Jian Pei and Yiwen Yin. *Mining frequent patterns without candidate generation.* In ACM SIGMOD Record, volume 29, pages 1–12. ACM, 2000. (Cited on page 68.)

[Han 2006] Jiawei Han, Micheline Kamber and Jian Pei. Data mining, southeast asia edition: Concepts and techniques. Morgan kaufmann, 2006. (Cited on pages 21, 39, 54, 68, 83, 96 and 104.)

[Harman 2001] Mark Harman and Bryan F Jones. *Search-based software engineering.* Information and Software Technology, vol. 43, no. 14, pages 833 – 839, 2001. (Cited on page 20.)

[Harman 2010] Mark Harman. *Why Source Code Analysis and Manipulation Will Always be Important.* In 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010), pages 7–19, 2010. (Cited on page 16.)

[Hasheminejad 2015] SMH Hasheminejad and S Jalili. *CCIC: Clustering analysis classes to identify software components.* Information and Software Technology, vol. 57, pages 329–351, 2015. (Cited on pages 7, 18, 20, 23, 25, 26 and 34.)

[Heineman 2001] George T Heineman and William T Councill. *Component-based software engineering*. Putting the Pieces Together, Addison-Westley, 2001. (Cited on pages 5, 6, 27 and 34.)

[Hendrickson 2007] S. A. Hendrickson and A. van der Hoek. *Modeling Product Line Architectures Through Change Sets and Relationships*. In Proc. of the 29th Inter. Conf. on Software Engineering, ICSE '07, pages 189–198, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 113.)

[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 27.)

[Kang 1990] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak and A Spencer Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Rapport technique, DTIC Document, 1990. (Cited on page 91.)

[Kang 2005] Kyo Chul Kang, Moonzoo Kim, Jaejoon Lee and Byungkil Kim. *Feature-oriented re-engineering of legacy systems into product line assets–a case study*. In Software Product Lines, pages 45–56. Springer, 2005. (Cited on pages 8, 19, 20, 22, 23, 25, 26, 32 and 90.)

[Kebir 2012a] 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, 15, 20, 21, 23, 24, 25, 26, 27, 29, 34, 35, 37, 60, 61, 98 and 107.)

[Kebir 2012b] Selim Kebir, Abdelhak-Djamel Seriai, Allaoua Chaoui and Sylvain Chardigny. *Comparing and combining genetic and clustering algorithms for software component identification from object-oriented code*. In Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering, pages 1–8. ACM, 2012. (Cited on pages 7, 20, 21, 22, 23, 24, 25 and 26.)

[Kolb 2005] Ronny Kolb, Dirk Muthig, Thomas Patzke and Kazuyuki Yamauchi. *A case study in refactoring a legacy component for reuse in a product line*. In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), pages 369–378. IEEE, 2005. (Cited on pages 18, 20, 22, 24, 25 and 26.)

[Kolb 2006] Ronny Kolb, Dirk Muthig, Thomas Patzke and Kazuyuki Yamauchi. *Refactoring a legacy component for reuse in a software product line: a case study*. Journal of Software Maintenance and Evolution: Research and Practice, vol. 18, no. 2, pages 109–132, 2006. (Cited on pages 18, 20, 22, 24, 25 and 26.)

[Koschke 2009] Rainer Koschke, Pierre Frenzel, Andreas PJ Breu and Karsten Angstmann. *Extending the reflexion method for consolidating software variants into product lines*. Software Quality Journal, vol. 17, no. 4, pages 331–366, 2009. (Cited on pages 8, 20, 22, 23, 24, 25, 26, 31 and 90.)

[Land 2009] Rikard Land, Daniel Sundmark, Frank Lüders, Iva Krasteva and Adnan Causevic. *Reuse with software components-a survey of industrial state of practice*. In Formal Foundations of Reuse and Domain Engineering, pages 150–159. Springer, 2009. (Cited on page 6.)

[Langelier 2005] Guillaume Langelier, Houari Sahraoui and Pierre Poulin. *Visualization-based analysis of quality for large-scale software systems*. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pages 214–223. ACM, 2005. (Cited on page 22.)

[Lanza 2003] Michele Lanza and Stéphane Ducasse. *Polymetric views-a lightweight visual approach to reverse engineering*. Software Engineering, IEEE Transactions on, vol. 29, no. 9, pages 782–795, 2003. (Cited on page 22.)

[Lau 2007] Kung-Kiu Lau and Zheng Wang. *Software component models*. Software Engineering, IEEE Transactions on, vol. 33, no. 10, pages 709–724, 2007. (Cited on page 6.)

[Leach 2012] Ronald J Leach. Software reuse: Methods, models, costs. AfterMath, 2012. (Cited on page 5.)

[Lethbridge 2003] Timothy C Lethbridge, Janice Singer and Andrew Forward. *How software engineers use documentation: The state of the practice*. IEEE Software, vol. 20, no. 6, pages 35–39, 2003. (Cited on page 18.)

[Linden 2007] Frank J. van der Linden, Klaus Schmid and Eelco Rommes. Software product lines in action: The best industrial practice in product line engineering. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. (Cited on page 6.)

[Luckham 1996] D Luckham. *Rapide: A language and toolset for simulation of distributed systems by partial orderings of events*, 1996. (Cited on page 93.)

[Lüer 2002] Chris Lüer and André Van Der Hoek. Composition environments for deployable software components. Citeseer, 2002. (Cited on pages 15, 27 and 34.)

[Ma 2006] Homan Ma, Robert Amor and Ewan Tempero. *Usage patterns of the Java standard API*. In Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC 2006), pages 342–352. IEEE, 2006. (Cited on page 60.)

[Magee 1996] Jeff Magee and Jeff Kramer. *Dynamic structure in software architectures*. ACM SIGSOFT Software Engineering Notes, vol. 21, no. 6, pages 3–14, 1996. (Cited on page 93.)

[McIlroy 1968] M Douglas McIlroy, JM Buxton, Peter Naur and Brian Randell. *Mass-produced software components.* In Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany, pages 88–98. sn, 1968. (Cited on page 5.)

[Mende 2008] Thilo Mende, Felix Beckwermert, Rainer Koschke and Gerald Meier. *Supporting the grow-and-prune model in software product lines evolution using clone detection.* In 12th European Conference on Software Maintenance and Reengineering (CSMR), pages 163–172. IEEE, 2008. (Cited on pages 18, 20, 22, 23, 24, 25 and 26.)

[Mende 2009] Thilo Mende, Rainer Koschke and Felix Beckwermert. *An evaluation of code similarity identification for the grow-and-prune model.* Journal of Software Maintenance and Evolution: Research and Practice, vol. 21, no. 2, pages 143–169, 2009. (Cited on pages 12, 18, 20, 22, 23, 24, 25 and 26.)

[Mishra 2009] SK Mishra, Dharmender Singh Kushwaha and Arun Kumar Misra. *Creating Reusable Software Component from Object-Oriented Legacy System through Reverse Engineering.* Journal of object technology, vol. 8, no. 5, pages 133–152, 2009. (Cited on pages 18, 20, 21, 23, 24, 25, 26 and 60.)

[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 5.)

[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 14.)

[Moshkenani 2012] Zahra Sadri Moshkenani, Sayed Mehran Sharafi and Bahman Zamani. *Improving Naïve Bayes Classifier for Software Architecture Reconstruction.* In Instrumentation & Measurement, Sensor Network and Automation (IMSNA), 2012 International Symposium on, volume 2, pages 383–388. IEEE, 2012. (Cited on pages 12, 18, 20, 21, 25 and 26.)

[Müller 2000] Hausi A Müller, Jens H Jahnke, Dennis B Smith, Margaret-Anne Storey, Scott R Tilley and Kenny Wong. *Reverse engineering: A roadmap.* In Proceedings of the Conference on the Future of Software Engineering, pages 47–60. ACM, 2000. (Cited on page 12.)

[Nakagawa 2011] Elisa Yumi Nakagawa, Pablo Oliveira Antonino and Martin Becker. *Reference architecture and product line architecture: a subtle but critical difference.* In Software Architecture, pages 207–211. Springer, 2011. (Cited on pages 6 and 91.)

[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 13.)

[Pinzger 2004] Martin Pinzger, Harald Gall, Jean-Francois Girard, Jens Knodel, Claudio Riva, Wim Pasman, Chris Broerse and Jan Gerben Wijnstra. *Architecture recovery for product families*. In Software Product-Family Engineering, pages 332–351. Springer, 2004. (Cited on pages 8, 12, 19, 20, 22, 23, 24, 25, 26, 32 and 90.)

[Pohl 2005a] Klaus Pohl, Günter Böckle and Frank J. van der Linden. Software product line engineering: Foundations, principles and techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. (Cited on pages 6, 14, 90 and 91.)

[Pohl 2005b] Klaus Pohl, Günter Böckle and Frank J. van der Linden. Software product line engineering: Foundations, principles and techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. (Cited on page 14.)

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

[Poshyvanyk 2006] Denys Poshyvanyk and Andrian Marcus. *The conceptual coupling metrics for object-oriented systems*. In Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on, pages 469–478. IEEE, 2006. (Cited on pages 46 and 71.)

[Razavizadeh 2009] Azadeh Razavizadeh, Hervé Verjus, Sorana Cimpan and Stéphane Ducasse. *Multiple viewpoints architecture extraction*. In Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009, pages 329–332. IEEE, 2009. (Cited on pages 12, 20, 22, 25 and 26.)

[Riva 2002] Claudio Riva and Jordi Vidal Rodriguez. *Combining static and dynamic views for architecture reconstruction*. In Proceedings. Sixth European Conference on Software Maintenance and Reengineering, 2002, pages 47–55. IEEE, 2002. (Cited on pages 18, 19, 20, 23, 24, 25 and 26.)

[Robillard 2013] M.P. Robillard, E. Bodden, D. Kawrykow, M. Mezini and T. Ratchford. *Automated API Property Inference Techniques*. IEEE Transactions on Software Engineering, vol. 39, no. 5, pages 613–637, 2013. (Cited on page 60.)

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

[Rumbaugh 2004] James Rumbaugh, Ivar Jacobson and Grady Booch. Unified modeling language reference manual, the. Pearson Higher Education, 2004. (Cited on page 18.)

[Sametinger 1997] Johannes Sametinger. Software engineering with reusable components. Springer Science & Business Media, 1997. (Cited on pages 7 and 34.)

[Seriai 2014] Abderrahmane Seriai, Salah Sadou, Houari Sahraoui and Salma Hamza. *Deriving component interfaces after a restructuring of a legacy system.* In 2014 IEEE/IFIP Conference on Software Architecture (WICSA), pages 31–40. IEEE, 2014. (Cited on pages 20, 21, 23, 25 and 26.)

[She 2011] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski and Krzysztof Czarnecki. *Reverse engineering feature models.* In The 33rd International Conference on Software Engineering (ICSE 2011), pages 461–470. IEEE, 2011. (Cited on page 12.)

[Shiva 2007] Sajjan G Shiva and Lubna Abou Shala. *Software Reuse: Research and Practice.* In International Conference on Information and Technology (ITNG 2007), pages 603–609, 2007. (Cited on page 5.)

[Sneed 2006] Harry M Sneed. *Integrating legacy software into a service oriented architecture.* In Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on, pages 11–pp. IEEE, 2006. (Cited on page 12.)

[Stojanović 2005] Zoran Stojanović and Ajantha Dahanayake. Service-oriented software system engineering: challenges and practices. IGI Global, 2005. (Cited on page 5.)

[Szyperski 2002] Clemens Szyperski. Component software: beyond object-oriented programming. Pearson Education, 2002. (Cited on pages 15, 27 and 34.)

[Tavares 2008] A. L. C. Tavares and M. T. Valente. *A Gentle Introduction to OSGi.* SIGSOFT Softw. Eng. Notes, vol. 33, no. 5, pages 8:1–8:5, August 2008. (Not cited.)

[Thüm 2014] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake and Thomas Leich. *Featureide: An extensible framework for feature-oriented software development.* Science of Computer Programming, vol. 79, pages 70–85, 2014. (Cited on page 113.)

[Tonella 2001] Paolo Tonella and Alessandra Potrich. *Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers.* In Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001), pages 376–385. IEEE, 2001. (Cited on page 12.)

[Uddin 2012] Gias Uddin, Barthélémy Dagenais and Martin P. Robillard. *Temporal Analysis of API Usage Concepts*. In Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, pages 804–814, Piscataway, NJ, USA, 2012. IEEE Press. (Cited on page 60.)

[Vliet 2008] Hans van Vliet. *Software Engineering: Principles and Practice*. 2008. (Cited on page 12.)

[von Detten 2012] Markus von Detten. *Archimetrix: A Tool for Deficiency-Aware Software Architecture Reconstruction*. In 2012 19th Working Conference on Reverse Engineering (WCRE), pages 503–504. IEEE, 2012. (Cited on pages 20, 21, 25 and 26.)

[von Detten 2013] Markus von Detten, Marie Christin Platenius and Steffen Becker. *Reengineering component-based software systems with Archimetrix*. Software & Systems Modeling, pages 1–30, 2013. (Cited on pages 20, 21, 25 and 26.)

[Wang 2013] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie and Dongmei Zhang. *Mining Succinct and High-coverage API Usage Patterns from Source Code*. In Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, pages 319–328, Piscataway, NJ, USA, 2013. IEEE Press. (Cited on page 60.)

[Weinreich 2012] R. Weinreich, C. Miesbauer, G. Buchgeher and T. Kriechbaum. *Extracting and Facilitating Architecture in Service-Oriented Software Systems*. In 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), pages 81–90, Aug 2012. (Cited on pages 18, 19, 20, 22, 24, 25 and 26.)

[Yevtushenko 2000] [A. Serhiy Yevtushenko. *System of data analysis "Concept Explorer"*. (In Russian) Proc. of the 7th National Conf. on Artificial Intelligence KII, Russia, vol. 79, pages 127–134, 2000. (Cited on page 110.)

[Yinxing 2010] Xue Yinxing, Xing Zhenchang and Jarzabek Stan. *Understanding Feature Evolution in a Family of Product Variants*. Reverse Engineering, Working Conference on, vol. 0, pages 109–118, 2010. (Cited on page 7.)

[Yuan 2014] Eric Yuan, Naeem Esfahani and Sam Malek. *Automated Mining of Software Component Interactions for Self-adaptation*. In Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, pages 27–36, New York, NY, USA, 2014. ACM. (Cited on page 12.)

[Zhang 2010] Qifeng Zhang, Dehong Qiu, Qubo Tian and Lei Sun. *Object-oriented software architecture recovery using a new hybrid clustering algorithm*. In Seventh International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), 2010, volume 6, pages 2546–2550. IEEE, 2010. (Cited on pages 18, 20, 21, 22, 23, 25 and 26.)

[Ziadi 2012] Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva and Mikal Ziane. *Feature identification from the source code of product variants.* In Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, pages 417–422. IEEE, 2012. (Cited on page 12.)

Supporting Reuse by Reverse Engineering Software Architectures and Components from Object-Oriented Product Variants and APIs

Abstract: It is widely recognized that software quality and productivity can be significantly improved by applying a systematic reuse approach. In this context, Component-Based Software Engineering (CBSE) and Software Product Line Engineering (SPLE) are considered as two important systematic reuse paradigms. CBSE aims at composing software systems based on pre-built software components and SPLE aims at building new products by managing commonalty and variability of a family of similar software. However, building components and SPL artifacts from scratch is a costly task. In this context, our dissertation proposes three contributions to reduce this cost.

Firstly, we propose an approach that aims at mining reusable components from a set of similar object-oriented software product variants. The idea is to analyze the commonality and the variability of product variants, in order to identify pieces of code that may form reusable components. Our motivation behind the analysis of several existing product variants is that components mined from these variants are more reusable for the development of new software products than those mined from single ones. The experimental evaluation shows that the reusability of the components mined using our approach is better than those mined from single software.

Secondly, we propose an approach that aims at restructuring object-oriented APIs as component-based ones. This approach exploits specificity of API entities by statically analyzing the source code of both APIs and their software clients to identify groups of API classes that are able to form components. Our assumption is based on the probability of classes to be reused together by API clients on the one hand, and on the structural dependencies between classes on the other hand. The experimental evaluation shows that structuring object-oriented APIs as component-based ones improves the reusability and the understandability of these APIs.

Finally, we propose an approach that automatically recovers the component-based architecture of a set of object-oriented software product variants. Our contribution is twofold: the identification of the architectural component variability and the identification of the configuration variability. The configuration variability is based on the identification of dependencies between the architectural elements using formal concept analysis. The experimental evaluation shows that our approach is able to identify the architectural variability.

Keywords: software reuse, reverse engineering, restructuring, reengineering, object oriented, software component, software product line architecture, software architecture variability, API, product variants.