UNIVERSIDADE FEDERAL DA BAHIA
UNIVERSIDADE ESTADUAL DE FEIRA DE SANTANA

DISSERTAÇÃO DE MESTRADO

An Assessment On Variability Implementation Techniques In Software Product Lines: A Replicated Case Study

LORENO FREITAS MATOS ALVIM

Programa Multi-Institucional de Pós-Graduação em Ciência da Computação

Salvador
Agosto/2016

LORENO FREITAS MATOS ALVIM

AN ASSESSMENT ON VARIABILITY IMPLEMENTATION TECHNIQUES IN SOFTWARE PRODUCT LINES: A REPLICATED CASE STUDY

M.Sc. Dissertation presented to the Multi-institutional Master Programme in Computer Science at Federal University of Bahia and Feira de Santana State University in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: EDUARDO SANTANA DE ALMEIDA

Salvador
Agosto/2016

I dedicate this dissertation to my parents, fiancée, friends and professors who gave me the necessary support to get here.

RESUMO

Computação Orientada a Serviços (COS) e Linhas de Produtos de Software (LPS) são abordagens destinadas ao desenvolvimento de sistemas de software que permitem às organizações reutilizarem, de forma sistemática, artefatos de software existentes ao invés de repetidamente desenvolvê-los a cada novo sistema implementado. Devido a essa característica, ambas as abordagens tem recebido uma crescente atenção de pesquisadores, principalmente, a sua combinação conhecida como Linha de Produtos Orientada a Serviços (LPOS). Isto pode ser justificada porque LPOS tem como objetivo alcançar os mesmos benefícios de ambas as abordagens e também resolver os problemas enfrentados por cada uma.

No entanto, mesmo com toda esta campanha, muitos desafios precisam ser superados, em particular, a falta de avaliações que considerem as diferentes técnicas de implementação de variabilidade. Este trabalho investiga este problema através de dois diferentes estudos de caso. Assim, buscou-se identificar qual técnica de implementação de variabilidade obteve melhores resultados com relação a complexidade, estabilidade e modularidade de software. Baseados nos resultados, um modelo de decisão inicial foi desenvolvido para auxiliar engenheiros de software a escolherem a mais adequada técnica de implementação de variabilidade com base nesses critérios observados.

Palavras-chave: Engenharia de Software, Linhas de Produtos de Software, Computação Orientada a Serviços, Linhas de Produto Orientada a Serviços, Estudo de Caso.

ABSTRACT

Service-Oriented Computing (SOC) and Software Product Lines (SPL) are approaches for developing software systems which enable organizations to reuse, in a systematic way, existing software assets rather than repeatedly developing them for new implemented systems. Due this characteristic, both approaches have received growing attention from researches, mainly, their combination known as Service-Oriented Product Lines (SOPL). It can be justified because SOPL has as objective achieving the same benefits from both approaches and also solving the problems facing by each one.

Nevertheless, ever with this hype, many challenges need to be overcome, in particular, the lack of assessments considering different variability implementation techniques. This work investigates this problem through two different case studies. Thus, we aim to identify which variability mechanism achieved better results with respect to complexity, stability and modularity of software. Based on the results, an initial decision model was developed to aid software engineers choose the most suitable variability implementation technique based on a set of parameters.

Keywords: Software Engineering, Software Product Lines, Service-Oriented Computing, Service-Oriented Product Lines, Case Study.

CONTENTS

List of Figures xiii

List of Tables xv

List of Acronyms xvii

Chapter 1—Introduction 1

1.1 Motivation 1
1.2 Goal of this Dissertation 2
1.3 Related Work 2
1.4 Out Of Scope 4
1.5 Statement of the Contributions 4
1.6 Research Design 5
1.7 Dissertation Structure 5

Chapter 2—An Overview on Software Product Lines and Service-Oriented Computing 7

2.1 Software Product Lines (SPL) 7
2.1.1 Advantages 8
2.1.2 The SPL Development Process 8
2.1.3 Variability Management 12
2.1.4 Product Derivation 13
2.2 Service-Oriented Computing (SOC) 18
2.2.1 Goals and Benefits of Service-Oriented Computing 19
2.2.2 The Principles 20
2.2.3 Service-Oriented Product Lines (SOPL) 22
2.2.3.1 Applying SPLE for Development of SOPL 25

2.3 Chapter Summary 28

Chapter 3—An Overview on Techniques to Implement Variabilities 29

3.1 Techniques for Implementing Variabilities 29
3.1.1 Conditional Compilation (CC2) 29
3.1.2 Aspect-oriented Programming (AOP) 30
3.1.3 Parameterization 31
3.1.4 Delegation 32
3.1.5 Inheritance 33
3.1.6 Libraries 34
3.1.7 Open Services Gateway Initiative (OSGi) 34
3.2 The Measurement Framework 36
3.3 Metrics For Code Quality Evaluation 37
3.4 Chapter Summary 41

Chapter 4—The Case Study 43

4.1 Case Study Protocol 43
4.1.1 Objective 43
4.1.2 The Case 45
4.1.3 Units of Analysis 50
4.1.4 Case Study Research Questions 50
4.1.5 Data Collection 51
4.1.6 Data Analysis 51
4.2 Results and Findings 52
4.2.1 Complexity Analysis 52
4.2.2 Stability Analysis 53
4.2.3 Modularity Analysis 54
4.2.4 Descriptive and Exploratory Analysis 55
4.2.5 Results Summary 58
4.3 Threats to Validity 58
4.4 Chapter Summary 59

Chapter 5—The Replicated Case Study 61

5.1 Case Study Protocol 61
5.1.1 Objective 62

5.1.2 The Case 63
5.1.3 Units of Analysis 68
5.1.4 Case Study Research Questions 68
5.1.5 Data Collection 68
5.1.6 Data Analysis 68
5.2 Results and Findings 69
5.2.1 Complexity Analysis 69
5.2.2 Stability Analysis 71
5.2.3 Modularity Analysis 71
5.2.4 Descriptive and Exploratory Analysis 72
5.2.5 Results Summary 75
5.3 Comparative Analysis 75
5.3.1 Decision Model 78
5.4 Threats to Validity 81
5.5 Chapter Summary 81

Chapter 6—Conclusions 83

6.1 Main Contributions 84
6.2 Concluding Remarks 84
6.3 Future Work 85

LIST OF FIGURES

1.1 Research Design. 5

2.1 Comparative among the accumulated cost to develop n kinds of systems as single system and SPL (Pohl *et al.*, 2005). 9

2.2 Time to market with and without product line engineering (Pohl *et al.*, 2005). 9

2.3 The tow-life-cycle model of software product line engineering (Linden *et al.*, 2007). 10

2.4 Essential product line activities (Clements and Northrop, 2001). 11

2.5 Hierarchy in product families (Sinnema *et al.*, 2006). 12

2.6 Example of feature model. 14

2.7 Product derivation process in SPL (Rabiser *et al.*, 2011). 15

2.8 Key activities for product derivation (Rabiser *et al.*, 2010). 15

2.9 Goals and benefits of service-oriented computing categorized in two groups: strategic goals and resulting benefits (Erl, 2007). 20

2.10 Services within a Product Line (Ribeiro, 2010). 24

2.11 Service-Domain Engineering of an SOPL (Mohabbati *et al.*, 2014). 25

2.12 Service-Application Engineering of a SOPL (Mohabbati *et al.*, 2014). . . 26

3.1 Example of the conditional compilation implementation technique. 30

3.2 Example of AOP implementation technique. 31

3.3 Example of the parameterization implementation technique. 32

3.4 File config.properties. 32

3.5 Variability by delegation technique. 33

3.6 Variability by Inheritance technique. 34

3.7 Service ativator. 35

3.8 Example of the OSGi implementation technique. 35

3.9 The Measurement Framework. 36

4.1 Screenshot a Warehouse SPL product implemented with conditional compilation. 46

4.2 Warehoure feature model. 48
4.3 Warehouse SPL implementation (Components). 49
4.4 Warehouse SPL implementation (Methods) 49
4.5 Warehouse Cyclomatic Complexity. 52
4.6 Warehouse Lines of Code. 53
4.7 Warehouse Weighted Operations. 54
4.8 Warehouse Instability. 54
4.9 Concerns Diffusion over Components for Warehouse 55
4.10 Concerns Diffusion over Operations for Warehouse 56
4.11 Boxplot comparing the variability mechanisms according to aspects complexity, stability and modularity. 57

5.1 Screenshot of RiSEEvent SPL product implemented with conditional compilation. 63
5.2 RiSEEvents feature model. 66
5.3 RiSEEvent SPL implementation (Components). 67
5.4 RiSEEvent SPL implementation (Methods). 67
5.5 RiSEEvent Cyclomatic Complexity. 69
5.6 RiSEEvent Lines of Code. 70
5.7 RiSEEvent Weighted Operations. 70
5.8 RiSEEvent Instability. 71
5.9 Concerns Diffusion over Components for RiSEEvent. 72
5.10 Concerns Diffusion over Operations for RiSEEvent 72
5.11 Boxplot comparing the variability mechanisms according to aspects complexity, stability and modularity. 74
5.12 Boxplot for the variables by program. 78
5.13 Decision model process. 79
5.14 Warehouse Decision model. 80
5.15 RiSEEvent Decision model. 80

LIST OF TABLES

4.1 Warehoure features for releases. 47
4.2 Results of the quantitative analyses for Warehoure. 56
4.3 Results for the quantitative analysis for Warehoure. 58

5.1 RiSEEvent features for releases. 64
5.2 Results for the quantitative analysis for RiSEEvent. 73
5.3 Results of the quantitative analysis for RiSEEvent. 75
5.4 Comparative table with the Warehouse and RiSEEvent SPLs. 76
5.5 Descriptive Statistics by Program. 77

LIST OF ACRONYMS

SOC	Service-Oriented Computing
SPL	Software Product Lines
SOPLE	Service-Oriented Product Line Engineering
AFM	Aspectual Feature Modules
SOA	Service-Oriented Architecture
SOPL	Service-Oriented Product Lines
CC1	Cyclomatic Complexity
LOC	Lines of Code
WOCS	Weighted Operations per Component or Service
IMSC	Instability Metric for Service or Component
CDC	Concerns Diffusion over Components
CDO	Concerns Diffusion over Operations
CC2	Conditional Compilation
AOP	Aspect-oriented Programming
OSGi	Open Services Gateway Initiative
GQM	Goal-Question-Metric

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

Software Product Lines (SPL) is a reuse-oriented approach that involves creating a set of related software systems from a reusable collection of software artifacts. Organizations are adopting this approach because it promises improvements in quality and productivity in software development. In this context, individual products are built through selection and customization of shared software artifacts developed during domain engineering. This process is named product derivation (Sinnema *et al.*, 2006), and occurs in the application engineering phase.

On the other hand, Service-Oriented Computing (SOC) is an emerging paradigm of software engineering. It allows decomposing software into services, providing a well-defined functionality and hiding systems implementation (Apel *et al.*, 2008). Services are autonomous, platform-independent computational units that can be described, published, discovered, and dynamically composed and assembled (Mohabbati *et al.*, 2014). Thus, programmers have conditions to integrate distributed services, even if they are written in different programming languages (Apel *et al.*, 2008). It makes SOC an alternative to solve integration and interoperability problems, and increase business flexibility. However, this paradigm alone does not offer strategies for high customization and systematic planned reuse (Medeiros *et al.*, 2010).

In order to achieve the benefits highlighted by these two approaches, the combination of SPL and SOC has received considerable attention of researchers in the last years. This interest is justified because they can address issues of each other. Regarding SPL, SOC can achieve dynamic reconfiguration, and for SOC, SPL can cover an approach to manage variations among multiple service-oriented systems. This combination of SPL and SOC is called Service-Oriented Product Line (SOPL) (Vale *et al.*, 2012).

Thus, this work investigates the combination of SPL and SOC with emphasis on variability implementation, more specifically, the choice of a specific variability mechanism (Conditional Compilation, Aspect-oriented Programming, or Open Services Gateway Initiative) according to aspects such as complexity, modularity or stability.

For this purpose, we implemented two SPLs and generated three releases for each SPL. Moreover, we developed two SOPL from these SPLs with features resolved by services. Finally, we defined a decision model that indicates the variability technique which produces the best components or services according to aspects observed. As in Sant'anna *et al.* (2003), for this work component is an abstraction that represents classes and aspects, and operations represent methods and advices.

1.2 GOAL OF THIS DISSERTATION

The goal of this dissertation can be stated as follows:

This work investigates the problem of variability implementation in software assets of software product lines and service-oriented product lines. Furthermore, a measurement framework and a decision model are presented in order to analyze implementations of SPLs, and to indicate the most suitable variability technique based on software quality.

1.3 RELATED WORK

Software variability management is a key activity to have success in the domain of software product lines. Gacek and Anastasopoules (2001) addressed issues of handling variability at the code level. They identified in the literature thirteen mechanisms for implementing variability (alone or combined with other) in product lines at the code level (*e.g.*, aspect-oriented programming, conditional compilation, delegation, parameterization, and so on). Furthermore, they proposed a framework for evaluating the implementation approaches regarding their use in a product line context. Our approach is similar, however, it also focuses on implementation mechanisms targeted to the SOPL.

Chang and Kim (2007) performed a technical comparison among variability on conventional applications and the variability on SOC. Based on the results, they identified four types of variability, which may occur on services. The main contribution was to build a framework that aids both in service modeling variability as well as designing of adaptable services, for each type of variability identified. Unlike our work, Chang and Kim (2007) did not quantitative analyses for identify what types of variability is most efficient for a particular metric.

Smith and Lewis (2009) defined an approach that uses Service-Oriented Architecture (SOA) and services as core assets in a software product line. This approach claims that before implementation the mapping of SOA concepts must be performed in the context of

SPL and defined an initial set of variation mechanisms to handle with variabilities issues. Ribeiro *et al.* (2011) proposed an approach for implementing core assets in service-oriented product lines providing guidelines and steps that enable the implementation in the SOPL context (components, services providers and consumers). Both studies contributed with our dissertation to indicate how features could be resolved by services. Moreover, Ribeiro *et al.* (2011) provided inputs for the development of our measurement framework that aims to collect data to evaluate the different implementations.

Apel *et al.* (2008) developed a feature-based approach, where a scenario that integrates the notions of services and feature-based product lines was created. This scenario provided conditions to discuss five benefits identified by the approach: improvements in modularity, variability, uniformity, specifiability, and typeability. This work provided for us the idea of one of the scenarios for implementing the SPL (the Warehouse). However, the focus of this approach was assess some variability techniques in the context of SPL and SOPL.

Zhang *et al.* (2016) characterized and compared seven variabilities mechanisms according with their techniques, binding time, granularity, and further aspects. Moreover, through the pratical usage they presented that benefits and challenges are determined by variability mechanisms characteristics. Although some techniques being the same of our study, the kind of comparison was different.

Matos Jr. (2008) compared some techniques used by the software industry that enable variability. Moreover, he classified these mechanisms according to their structure and location in the source code, and presented variability implementation patterns using different techniques in order to deal with each kind of variation. At the end, he analyzed the impact resulting from each pattern using qualitative and quantitative metrics. Ferreira *et al.* (2014) through a quantitative and qualitative analysis identified how feature modularity and change propagation behave in the context of two evolving SPLs. In order to collect the quantitative data, they developed each SPL using three different variability mechanisms. Moreover, the compositional mechanism available in Feature-oriented Programming (FOP) was evaluated by using other two variability techniques as baseline (conditional compilation and design patterns). Gaia *et al.* (2014) expanded this work and investigated whether the simultaneous use of aspects and features through the Aspectual Feature Modules approach (AFM) facilitates the evolution of SPLs. All these works compare some variability techniques through quantitative and qualitative analysis, however, neither of them in the context of SOPL.

Carvalho *et al.* (2016) performed an exploratory study to analyzing solutions used in dynamic variability implementation. This evaluation was performed with respect to

size, cohesion, coupling, and instability of the source code with the aid of a measurement framework. Although similar, our work focuses in context of SPL and SOPL and this work in dynamic software product lines context.

1.4 OUT OF SCOPE

Some SPL and SOC topics were not investigated in this work such as:

- **Other activities in SPL and a service-oriented product lines process.** This work focuses on variability implementation at the code level which enables the development of different products by means of the product derivation process. Other activities such as design, evolution and testing were not covered in this dissertation.

- **Quality Attributes.** Performance, availability, security, among others quality attributes were not considered in this work. It is an extensive area out of the scope of this dissertation.

- **Combination of Technologies.** The combination of services with variability techniques at SPL can provide good results for software development. However, because of the scope of this study, possible combination were not considered.

1.5 STATEMENT OF THE CONTRIBUTIONS

The main research contribution of this dissertation are the case studies presented. Through them were performed the assessment of variability techniques with the aid of framework developed which provide support for the development of decision model. These and other contributions can be highlighted as follows:

- An overview of techniques used for handling product lines variability at the code level.

- An assessment for comparing techniques used to implement components and services regarding complexity, stability, and modularity in the context of service-oriented product lines.

- An case study performed according with guidelines proposed by Runeson *et al.* (2012), which was applied the measurement framework on the first SPL (Warehouse).

- A replicated case study in another domain (RiSEEvent SPL) which allowed to compare the results and infer indicators about the evaluation performed.

- The definition of an initial decision model which provides guidelines for software engineers during the choice of techniques that support variability implementation at the code level.

1.6 RESEARCH DESIGN

Figure 1.1 shows the research design approach defined for this work which is composed of five key activities: literature review, measurement framework, SPLs development, techniques evaluation, and decision model construction.

The first activity involved a brief literature review where we identified proposals for implementing variability in SPL, and how to apply services to develop SOPL. Furthermore, previous work that provided guidelines, and metrics for evaluation of the techniques were also investigated.

Based on the results from the literature, we developed a framework (second activity) to evaluate the techniques used in the implementation of SPLs and SOPL, which were implemented during the third step of this work.

In order to apply the framework and evaluate the techniques, we conducted two exploratory case studies (fourth step): the first one was applied to an warehouse domain, and the second one focused on a SPL responsible for managing academic events.

Finally, we developed an initial decision model that indicates the most suitable variability technique for a given context. The tasks performed in the second and third steps have high relevance for this model, once they impact directly in its quality.

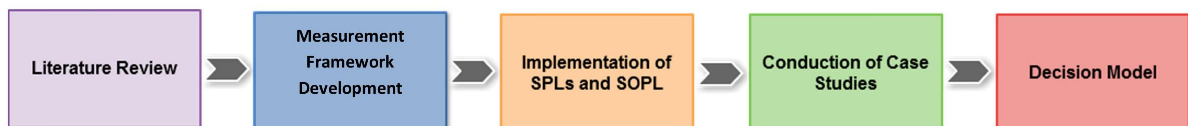


Figure 1.1: Research Design.

1.7 DISSERTATION STRUCTURE

The remainder of this dissertation is organized as follows:

- **Chapter 2** discusses essential aspects related to this work: Software Product Lines and Service-Oriented Computing.

- **Chapter 3** presents a brief assessment which evaluates techniques used to implement assets in SPL or SOPL.

- **Chapter 4** describes the case study which evaluates the different variability techniques. The case study protocol, research questions, data collection, data analysis, and outcomes are described in details.
- **Chapter 5** presents a replicated case study using another SPL, as well as a comparative analysis. Moreover, it presents the decision model constructed based on results from two case studies.
- **Chapter 6** provides the concluding remarks and directions for future work.

CHAPTER 2

AN OVERVIEW ON SOFTWARE PRODUCT LINES AND SERVICE-ORIENTED COMPUTING

This chapter presents some background information about Software Product Lines (SPL) and Service-Oriented Computing (SOC), which are the main topics of this work. This Chapter is organized as follows: in Section 2.1, we contextualize and introduce concepts related to Software Product Lines development processes (Clements and Northrop, 2001) (Pohl *et al.*, 2005), motivations and benefits for applying this approach. Section 2.2 explains the SOC aspects and the principles that guide the development process. Finally, we introduce the approach resultant of the combination between SPL and SOC.

2.1 SOFTWARE PRODUCT LINES (SPL)

Nowadays, reusing artifacts is a common practice in the software engineering field. This activity comprises sharing from pieces of code and service libraries to software components and frameworks. One of the systematic approaches for software reuse is the development of Software Product Lines (SPL) (Clements and Northrop, 2001).

The idea of SPL was proposed by Parnas (1976). He defines product family as a collection of solutions with high level of similarities. For him, this characteristic justifies studying first the common properties and subsequently the particularities of each software.

According to Clements and Northrop (2001), SPL can be defined as a set of related software-intensive systems sharing a collection of features to satisfy specific needs of customers or an particular market segment. The *particular market segment* refers to the domain and the business strategies of organization that can be changed to satisfy the stakeholders requirements (Mohabbati *et al.*, 2014). Software-intensive systems involve a set of core assets such as reusable software components, architecture, domain models and specifications.

Pohl *et al.* (2005) claim that software product lines is a paradigm to develop software using platforms and mass customization. According to them, developing applications based on platforms means building reusable code parts, which will be reused in several systems. On the other hand, mass customization implies employing the concept of

managed variability, *i.e.*, commonalities and variabilities among applications in a software product line (in terms of requirements, architecture, components, and test artifacts).

According to Klewerton and AssunÃ§Ã£o (2015), the most suitable SPL development strategy for organizations that have many products in their catalog is the extractive approach. The process of this strategy is based on a existent system as the starting point. Collecting relevant information such as traceability links among features is the first step. It is followed by identification of the similarities and variabilities among products, and finalized with the construction of corresponding artifacts for common and variable parts of an SPL. At the end this process, the main artifact created is the Product-Line Architecture. It provides a common design to all products of the SPL, in other words, through this artifact, software engineers can identify the mandatory and optional features from the considered domain.

2.1.1 Advantages

An extensive change in engineering practices is usually not initiated without an economic justification. Thus, one of the first reasons to adopt product line engineering is the cost reduction achieved with the change in the development process. Figure 2.1 shows the accumulated costs needed to develop n different systems (Pohl *et al.*, 2005).

The cost reduction is obtained by reuse of artifacts in several different kinds of systems, which implies in lower expenses for each system. However, before reusing artifacts, investments are required to plan and develop them. Therefore, only from the third system, the development costs in software product line engineering become smaller than those of individual systems. The artifacts in the platform are reviewed and tested in many products. These extensive tests imply in higher chance of detection and correction of faults, besides increasing the quality of all products (Pohl *et al.*, 2005).

Reduction of time to market is another advantage obtained with reuse. In opposite of single-product development, whose time can be roughly assumed as constant, product line engineering tends to present higher time-to-market. However, after having the resulting artifacts, this hurdle is considerably shortened to each new product. Figure 2.2 shows this scenery (Pohl *et al.*, 2005).

2.1.2 The SPL Development Process

The main difference between single system development and SPL engineering is the focus. Single system development aims at creating a unique system at a time, and product line

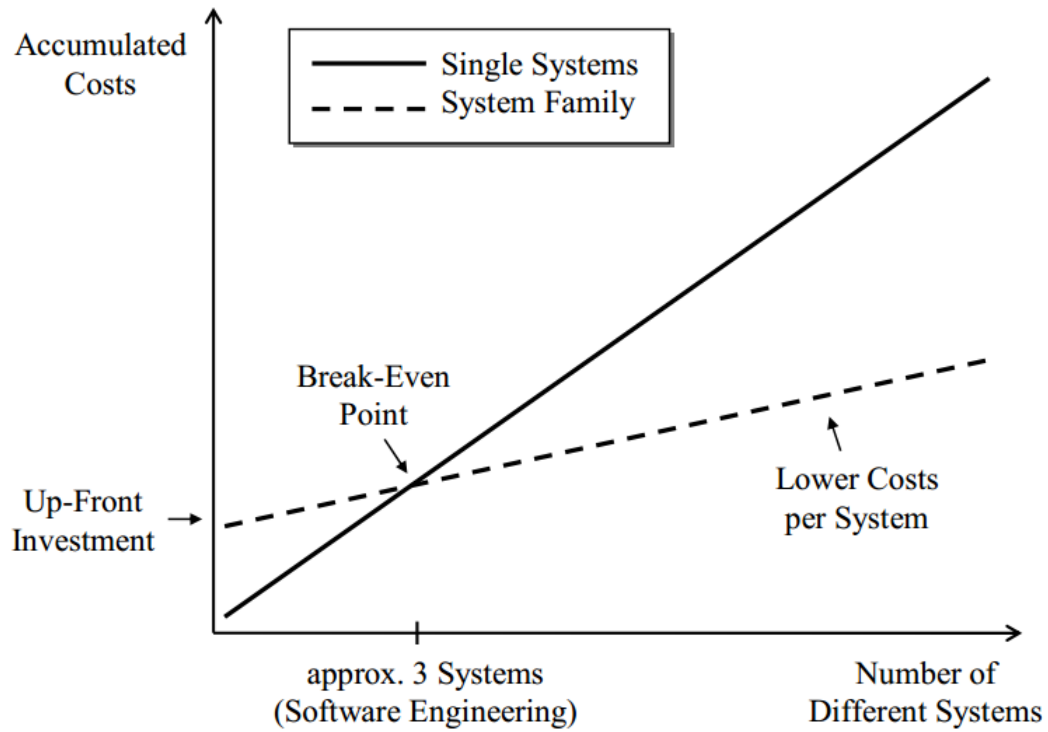


Figure 2.1: Comparative among the accumulated cost to develop n kinds of systems as single system and SPL (Pohl *et al.*, 2005).

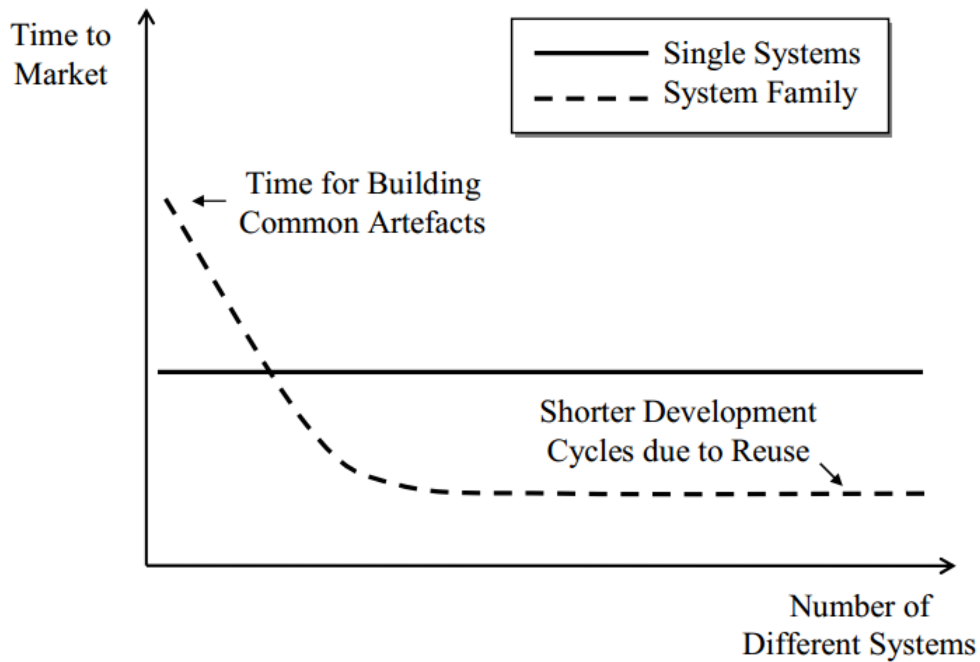


Figure 2.2: Time to market with and without product line engineering (Pohl *et al.*, 2005).

engineering can generate a set of systems. The change of focus is essentially of the business strategy. In other words, traditional development is driven by customer needs defined in a contract, in turn, SPL development is geared towards a niche market (Linden *et al.*, 2007).

Figure 2.3 shows the domain engineering and the application engineering processes, with a fundamental distinction of development *for* reuse and development *with* reuse respectively, in order to compose software product lines (Linden *et al.*, 2007).

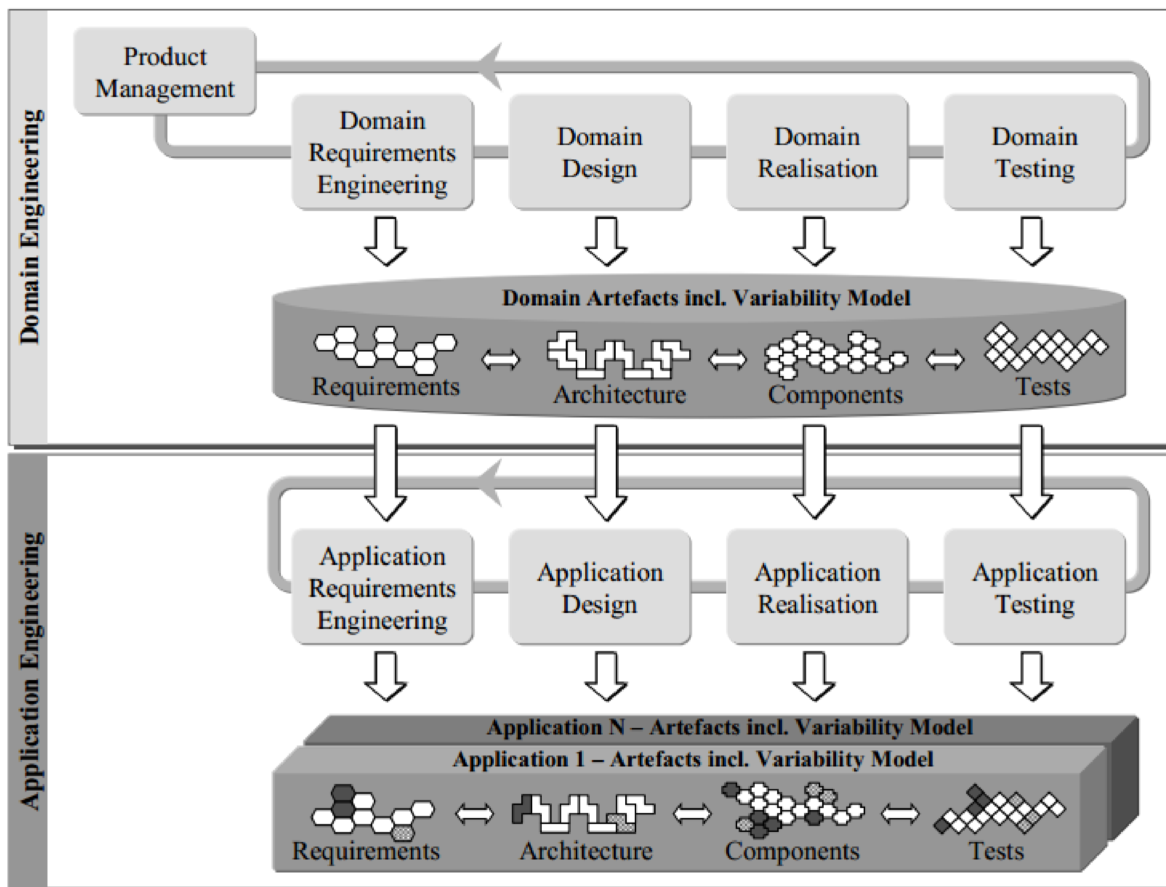


Figure 2.3: The tow-life-cycle model of software product line engineering (Linden *et al.*, 2007).

In *domain engineering*, a basis is provided for the development of specific products. As opposed to many traditional reuse approaches, not only code assets are encompassed in the infrastructure, but also all other software artifacts (Linden et al., 2007). In summary, domain engineering is the process of defining the commonality, variability, the scope of the SPL, and the development of reusable artifacts to accomplish the defined variabilities. Due to this, it is also known as Core asset Development (Pohl *et al.*, 2005).

Application engineering is considered Product Development because it is responsible for building the final products on top of the product line infrastructure established in domain engineering. This usually contains most of the functionality required for a new product (Linden *et al.*, 2007). The key goals of application engineering are: to achieve a comprehensive reuse of domain assets; to exploit the commonality and variability of the SPL; to document the application artifacts (*i.e.* application requirements, architecture, components and tests); and to estimate the impacts of the differences between application and domain requirements on architecture, components and tests (Pohl *et al.*, 2005).

Linden *et al.* (2007) also present several principles considered fundamental to success of software product line engineering. According to them, they can be described as *variability management* (made explicitly and must be systematically managed), *business-centric* (product line infrastructure is in the long term an adequate instrument to field new products onto the market), *architecture-centric* (the technical side must allow taking advantage of similarities among systems), and *low-life-cycle* approach (distinction of domain and application engineering is a key characteristic of SPL).

Clements and Northrop (2001) summarize all activities for SPL development in three essentials ones (core asset developing, product development, and management). Figure 2.4 shows the three essential activities. They consider this representation in the highest level of generality to combine technology and business practices.

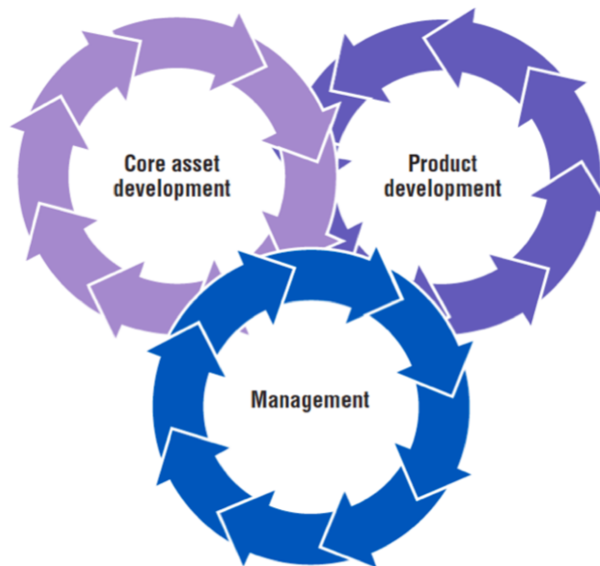


Figure 2.4: Essential product line activities (Clements and Northrop, 2001).

The rotating arrows indicate that companies review the existing core assets and new core assets can evolve during product development. There is also a strong feedback loop

between the core assets and the products. Core assets are updated as organizations develop new products. Next, they track asset use, and the results are sent to the asset development activity (Clements and Northrop, 2001).

2.1.3 Variability Management

Variability management is a key factor to differentiate conventional software engineering and SPL engineering. Differences that appear such as an evolution of programs over the time are known as variabilities in time. On the other hand, there are also variabilities in space. These occur when differences among software releases happen in a fixed point in time. Unlike traditional software development, SPL engineering deals with variability in both time and space, but for this study, variabilities in space are the main focus for allowing developers to deal with decisions of design, known as variation points (Matos Jr., 2008).

Variation points are places in a design or implementation used to identify where the variations occur. They are identified as central elements in the variability process because facilitates the systematic documentation, traceability, development, assessment and evolution of variability (Sinnema *et al.*, 2006).

Figure 2.5 shows that hierarchically product lines can be divided into three abstraction levels: features compose the layer with higher level of abstraction, the architecture makes the intermediate layer, and component implementations integrate the lowest level. Thus, as variation occurs in the entire development process, it is possible to identify variation points in all layers (Sinnema *et al.*, 2006).

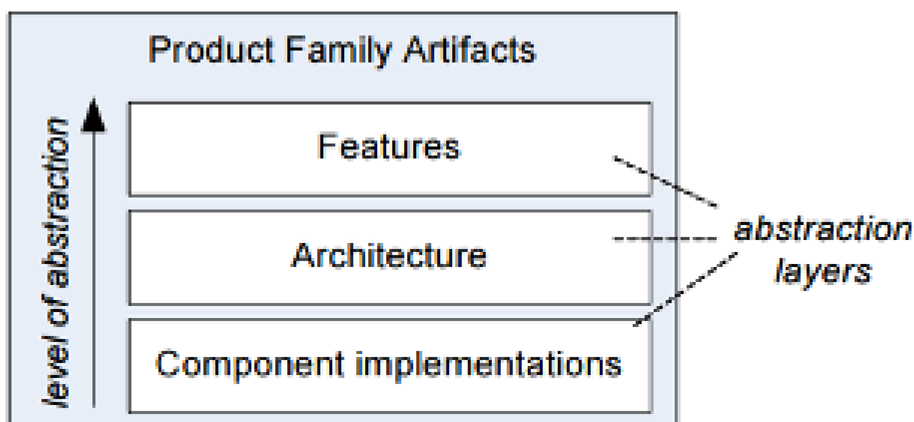


Figure 2.5: Hierarchy in product families (Sinnema *et al.*, 2006).

Some researchers consider as feature both variabilities and commonalities, which

commonly represents reusable requirements (Ribeiro, 2010). Kang *et al.* (1990) define a feature as *"a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"*.

Many approaches have a shared goal to model commonalities and variabilities in the SPL context, such as Clements and Northrop (2001), Gomaa (2004) and Pohl *et al.* (2005). Among these approaches, feature modeling, proposed by Kang *et al.* (1990), is the most used technique.

Basically, a feature model is a hierarchically arranged set of features that provides a graphical representation of variability relations, constraints and dependencies among features. It provides a good means of communication for the requirements analyst obtain knowledge about the SPL domain, helping him to deal with the user's problems (Kang *et al.*, 1990). Because of this, feature model has been widely used by researches and developers of SPL to specify what features are common or variable in the SPL (Gamez *et al.*, 2015). In this sense, Kang *et al.* (1990) classified features as:

Mandatory features: all features of this kind must be present in every product of the SPL. They represent the commonality of the product line.

Optional features: opposite of mandatory features, this kind of feature is a variability and can be present or not in products of the SPL.

Alternative features (Xor-features): only one feature of this group must be selected during the configuration of a specific product.

Or-features: one or more features can be selected to create a valid configuration of feature model.

Abstract Features: are used only for organizing the structure of a feature model. In other words, they do not have any impact at implementation.

Concrete Features: in turn it is responsible by providing the functionality to systems.

Figure 2.6 is an example of a feature model that represents a family of cars. The first feature of diagram (ABS) is a mandatory feature identified by a black circle. The optional feature is represented by the feature airbag.

In this example, every car must be painted by one color (Xor-feature) identified by a empty arc above the set of features red, black and silver. Moreover, the black arc indicates that the cars must support one or more kind of fuel (Or-feature).

2.1.4 Product Derivation

Product derivation is a process of selecting and customizing shared assets during application engineering. In this activity, customers, domain experts, business professionals and

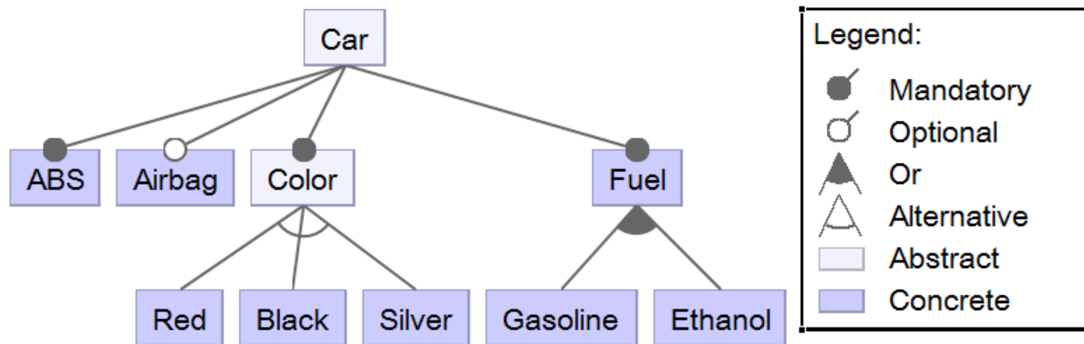


Figure 2.6: Example of feature model.

other stakeholders take knowledge about the variability provided by SPL. Based on this information and customers requirements, the derivation team selects a particular product and adapts it for a particular purpose. In practice, product derivation rarely is a sequential process and many interactions are necessary for bringing out customers requirements and solving variability (Rabiser *et al.*, 2011).

The aim of product derivation is to generate products to customers, and bring return on initial investment necessary to operate an SPL economically (Rabiser *et al.*, 2010). The costs for building up an SPL should be outweighed by the benefits of rapid derivation of customized products during application engineering. This is achieved by exploiting the commonality and variabilities established in domain engineering (Rabiser *et al.*, 2011).

Figure 2.7 shows a high-level view of application engineering. In this figure, the upper white vertical arrows depict the derivation process of selecting and customizing reusable assets. On the other hand, the deployment activities to release the product are denoted by lower white arrows.

In order to identify the key activities for product derivation, O'Leary *et al.* (2009) compared two approaches with particular goals, developed in different and independent projects: Process framework for Production Derivation (Pro-PD) and Decision-Oriented Product Line Engineering for effective Reuse: User-centered User (DOPLER). Pro-PD aims at defining a general process framework for product derivation. The DOPLER, in turn, aims at establishing a user-centered tool-supported for product derivation, with the goal of attending needs from industry.

Figure 2.8 shows the results of a mapping conducted in the chosen approaches Pro-PD and DOPLER. The key activities have been divided in three groups: (i) preparing for derivation, (ii) product derivation / configuration, and (iii) additional development /

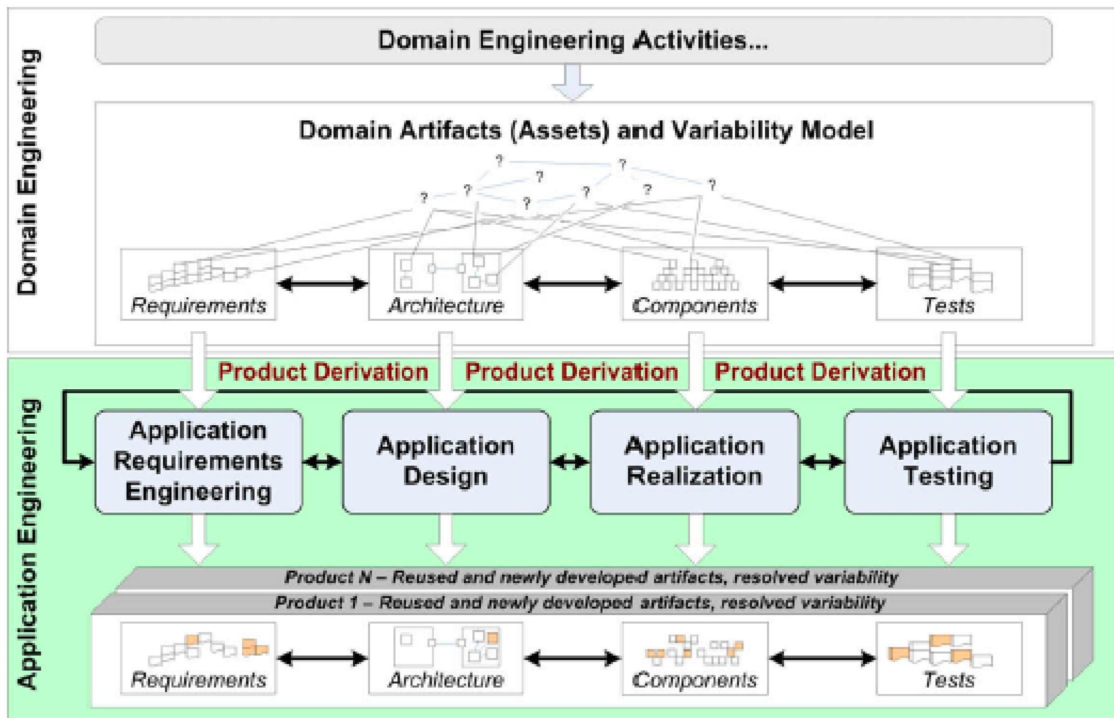


Figure 2.7: Product derivation process in SPL (Rabiser *et al.*, 2011).

testing.

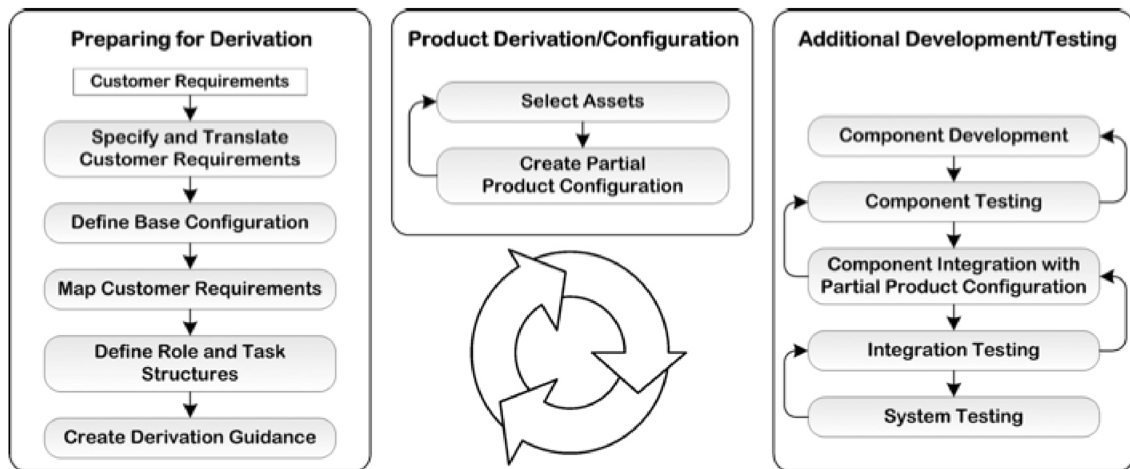


Figure 2.8: Key activities for product derivation (Rabiser *et al.*, 2010).

Preparing for derivation gathers the preparatory activities necessary for achieving derivation. No derivation (from the investigated approaches) starts from scratch, just

selecting features or making decisions, for example, as defined in the variability model (Rabiser *et al.*, 2010). Preparatory activities are listed as following:

- *Specify and translate customer requirements* - The start point of products derivation is a clear specification of customer demands. If needed, they should be translated to the language used in the organization. The purpose with these actions is to avoid a terminology confused and assets described in customer-specific language.

- *Define base configuration* - The set of existing platform configurations provides the options of choosing a base configuration. Experiences acquired in past projects are of great importance, because customers could have similar requirements providing a possibility of reuse. However, if no configuration is appropriate, other base configuration should be created.

- *Map customer requirements* - Base configuration is used to map the customers' needs. When some need cannot be satisfied with the assets by reuse or adaptation should be started a negotiation process with the customer. Questions as profitability of the platform assets for the whole product line must be taken into consideration.

- *Define role and task structures* - The goal is to define who is responsible for what of the remaining tasks in the derivation process. This is important because reduces the complexity of large decisions to achieve different views. Furthermore, it is important to know who made what and when.

- *Create derivation guidance* - Create ways to facilitate the decision making by domain experts is essential, since remaining variability must be explained. Moreover, while sales people need to understand variability from a high level, engineers need to know the details.

Product derivation / configuration starts with a selection and customizing of assets from platform, identifying if new developments are necessary. It should be an iterative process to ensure that all customer requirements have been fulfilled (O'Leary *et al.*, 2009):

- *Select assets* - Given the role and task structures defined previously, assets are selected from the product line. Tool support must be used for evaluating the dependencies and constraints between the assets.

- *Create partial product configuration* - Step-by-step and in an iterative way, a partial configuration that does not solve fully variability is created. In an ideal case, this first product would be enough for satisfying all customer requirements. However, in many cases, is necessary some additional development, that should have its activities defined and prioritized based on customer requirements. A good alternative for supporting the negotiations with customers is the use of simulations based on partial configuration.

Additional development / testing is the last set of activities performed during the derivation process. These tasks are responsibility of the product development team and have as goal to implement the required changes at the product level (Rabiser *et al.*, 2010):

- *Component development* - New source code is developed to implement new functionality or to adapt an existent at platform. It is important to take in consideration that new components must be developed with the possibility that they can later be updated to a platform asset.

- *Component testing* - After the process of building or adaptation, new parts of components should be tested rigorously, for example, by unit testing.

- *Component integration with partial product configuration* - The newly developed and adapted assets must be integrated with the partial product configuration. In order to do this, it will be necessary code to linking the product interfaces with them or even implementing architectural changes to facilitate integration.

- *Integration testing* - To verify whether the newly developed or adapted assets interact correctly with the existent architecture, it is essential the achievement of integration testing. During testing will be checked the consistency and correctness of the product.

- *System testing* - After all activities, the product suffers a check to confirm that it is in accordance with all product-specific requirements. If the product is not approved, new iterations will be required.

Product derivation faces many difficulties, and these challenges exist given the inherent size and complexity of the SPL. Moreover, communicating the variability for different stakeholders is not an easy activity, because people are not always available (Rabiser *et al.*, 2010).

Deelstra *et al.* (2004) say that derivation of individual products, from software assets shared in SPL, is an activity that demands a considerable time and effort. They performed a case study in two large and mature industrial organizations where identified a group of product derivation problems such as: (i) false positive to check the compatibility during component selection, (ii) large amount of human errors, (iii) consequences imposed by variant selection unclear, (iv) repetition of development caused for not finding the component between thousands in repository, and (v) differences among provided and required interfaces. According to them, these problems are created by lack of a methodology that supports to application engineering, apart from the underlying causes of complexity and implicit properties.

In addition, some tools were developed to support the SPL process. Lisboa *et al.* (2011) performed a systematic study and presented a list of tools that offer support to domain analysis phase, with available executable, and documentation describing their functionalities. Among the nineteen tools identified thirteen provide support for product derivation, such as: Gears (Krueger, 2007), Pure::variants (Schulze *et al.*, 2013), and FeatureIDE (Thüm *et al.*, 2014). Schmid and de Almeida (2013) also looked at technology for modeling and managing variation and thus facilitate product lines engineering. They presented others tools such as EASy-Producer (El-Sharkawy *et al.*, 2011) and (Dhungana *et al.*, 2013).

de Souza *et al.* (2015) performed a multiple-case study in two different software companies with aim to investigate how product derivation is performed in practice. They compared the case studies to find similarities and differences in order to identify patterns into concepts explored in the literature, such as, iterative and incremental product derivation.

2.2 SERVICE-ORIENTED COMPUTING (SOC)

Since 70s, the programming languages incorporated the concept of abstractions from details of software functionality. The functional decomposition was the first technique adopted. In this direction, with the increasing complexity, concepts of encapsulation were introduced in the programming languages. This increase of abstraction is an important step towards service-orientation, because it is one of the key principles of Service-Oriented Computing (SOC) (Vale *et al.*, 2012).

SOC is the computing paradigm that utilizes services as essential element for developing applications that address the needs of customers (Papazoglou, 2003). Three areas compose the basis of service-orientation: programming paradigms, distributed computing, and

business computing (Krafzig *et al.*, 2004).

Based on services, SOC can support a process of quick development, with low-cost, interoperable, evolvable, and massively distributed applications. Services are autonomous (an autonomous piece of software has the freedom and control to make its own decisions without external involvement), platform-independent (can be accessed independent of hardware or software platform on which was implemented), that can be described, published, and discovered. They perform the most varied functions that range from answering simple requests to sophisticated business processes (Papazoglou *et al.*, 2007). Moreover, when developers make use of the full potential of semantic for services, the results are easily customizable modules, which simplify the future software development (Huhns and Singh, 2005).

In order to achieve all goals and consequently satisfy these requirements, services should be technology neutral (invocation mechanisms must use protocols, descriptions and discovery with widely accepted standards), loosely coupled (it is not necessary to know the internal functionality of service neither have any internal structures), and support location transparency (definitions and locations of services must be stored in repository available for the most kinds of customers) (Papazoglou, 2003).

2.2.1 Goals and Benefits of Service-Oriented Computing

Service-oriented computing has a extremely wide vision, which makes SOC very interesting to companies willing to improve the efficiency of their IT department. Erl (2007) identified seven goals that form the vision and established a target for enterprises that adopt SOC.

Figure 2.9 shows and categorizes these goals in two groups (strategic goals and strategic benefits) because three of them (increased organization agility, increased return on investment, and reduced IT burden) are concrete benefits resulting from other four (increased intrinsic interoperability, increased federation, increased vendor diversification options, and increased business and technology domain alignment) (Erl, 2007):

Software interoperability occurs when the exchange information between programs is allowed. Programs with this feature are desirable because avoid the necessity of integration. Service-orientation has as goal to establish native interoperability within services thereby defining, a foundation for the realization of other strategic goals and benefits.

Increased federation is the second goal of software-oriented computing. Resources and application united, providing autonomy and self-governance characterize federated IT environments. In order to achieved it, the services are developed in a standardized way and supporting composability.

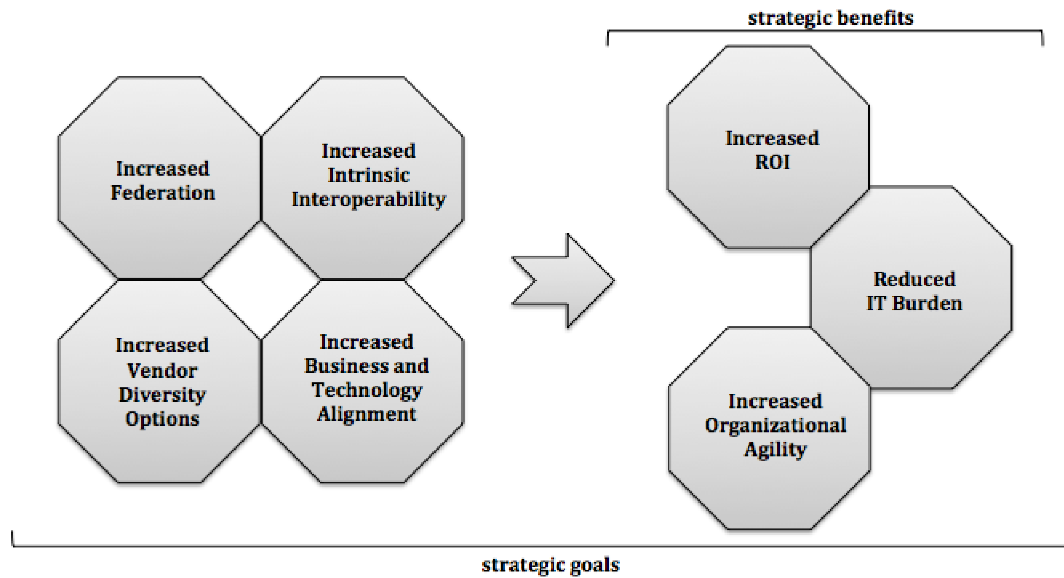


Figure 2.9: Goals and benefits of service-oriented computing categorized in two groups: strategic goals and resulting benefits (Erl, 2007).

Another target that determines guidelines for service-oriented is the *variety of vendors*. This objective is based on the principle that it is beneficial for organizations having many choices of vendors. This way, enterprises can perform changes without affecting the overall architecture of the produced solutions.

The last goal classified as strategic is the *increased business and technology domain alignment*. SOC introduces a design paradigm that promotes abstraction on many levels. The establishment of service layers is one of the most efficient means where functional abstraction is applied. It can also encapsulate and represent business models.

2.2.2 The Principles

Service-orientation aims at building distributed solutions. What differentiates service-orientation from other paradigms is the way used to conduct the separation of concerns and how the solution logic is shaped. Erl (2007) describes the principles that rule the service-oriented computing:

- **Service Contracts:** service contracts are a focal point of service design because they define everything that a service does. They are responsible for establishing the constraints and technical requirements. Moreover, this principle describes semantic information that the service owner wishes to make public.

Contracts aim at enabling services to have a meaningful level of interoperability, thus reducing the need for data transformation and allowing that services be more easily and intuitively understood.

- **Service Coupling:** represents the level of dependency among parts that compose the service. This principle advocates a specific type of relationship inside and outside of service, which emphasize always in reduction of dependency between the service contract, its implementation, and its service consumers. The aim is to get the max of independency among services and thus having the minimal impact when they evolve.

- **Service Abstraction:** emphasizes the need to hide all possible details of a service, avoiding unnecessary access to additional service details. As a result, this principle provides the conditions to get a loosely coupled relationship, reducing costs and effort of management the service.

- **Service Reusability:** reaffirm services as business resources with functional context not defined. This ensures that services may be repeatedly leveraged over time, providing return on the initial investment of delivering the service. Moreover, the reuse of services increases the agility of companies, enabling a rapid fulfillment of future business automation requirements through wide-scale service composition.

- **Service Autonomy:** services are inherently autonomous, and it means that they have the freedom and control to make its own decisions without the need for external approval. A fundamental objective of the service autonomy is to expand the amount of control that a service has over its runtime environment. To achieve this goal, autonomy requires a more isolated software implementation that increases the independence among pieces of software.

- **Service Statelessness:** requires minimal consumption of resources with management of state information. This concern relates with the availability of service, because managing state information in excess can compromise it. Thus, it applies the principle in order to increase service scalability and improve the potential for service reuse.

- **Service Discoverability:** in an organizational environment, the term discovery means more that only the need for something to be effectively searched and located. In these situations, encompasses for example, the means of consistently communicating information about resources (meta information), or still meta information

to be clearly documented by those who have the appropriate communication skills. The aim of principle service discoverability is to express the purpose and capabilities of the services, so that humans and software programs have capability to interpret them.

- **Service Composability:** the sophistication adopted in each new service-oriented solution suffers an increase. As consequence, the same happens with the complexity of underlying service composition configurations. Thus, the ability to effectively compose services becomes a critical requirement for achieving the goals established for service-oriented computing. The service composability operates to ensure that services are designed in order to participate as effective members of multiple compositions.

2.2.3 Service-Oriented Product Lines (SOPL)

Using the service orientation idea, the software development process changed. A system is no longer developed, integrated, and released in a sequential way. Such as SPL, this approach supports continuous change in expectations and customer requirements. Thus, the system evolution occurs by means of the addition and integration of new services (Lee *et al.*, 2008).

Both approaches (SPL and SOC) encourage reusing the existing software assets and capabilities rather than to develop them from scratch. It implies in productivity gains, decreased development costs, improved time-to-market and competitive advantage. However, both have also distinct goals that may be stated as (Cohen and Krut, 2007):

- Software product lines systematically capture and exploit commonality among a set of related systems while managing variations for specific customers or market segments.

- Service-oriented computing enables assembly, orchestration and maintenance of enterprise solutions to quickly react to changing business requirements.

Service-Oriented Product Line (SOPL) is an approach that combines concepts and techniques of SPL and SOC to achieve high customization, systematic reuse and other benefits related to software product lines (Medeiros *et al.*, 2009). On the other hand, services can aid to achieve flexibility and dynamic reconfiguration, since in most of the SPL approaches, is necessary to instantiate a product before delivering for customers, making difficult to make any changes to the software (Vale *et al.*, 2012).

The combination of these approaches has been a subject of considerable research interest in recent years. Mohabbati *et al.* (2014) conducted a mapping study, which indicates the combination of SOC and SPL as a promising way to development of architectures for adaptive systems, with an effective response to dynamic functional and non-functional requirements. Moreover, this integration enables reusing of architectures in different instances.

Some researchers have presented work where SOPL is the alternative chosen for the development of systems. Cohen and Krut (2007) presented two possibilities of connection among SOC and SPL. In the first one, services are designed as an SPL. In this scenario, services themselves would be configurable according to architecture variations or specific features. Thus, the product line encompasses its model, design and development in the context of service-oriented.

In the second one, services will be included within the SPL architecture. To achieve this, developers should include a variation point in the architecture implemented as a component or service. Depending on the features needed by the application, a specific configuration selects the component or the service in accordance with specificities by each alternative. Services in this context address possible selection of features such as: a need for dynamic variation, exploitation of the availability of existing services or quick construction of SPL systems (Cohen and Krut, 2007).

Kotonya *et al.* (2009) proposed an approach that integrates SPL and SOC through two steps. They declared that to achieve success in promoting reusability in service-oriented product-line development it is essential the correct specification and identification of services with the right features. Thus, during feature analysis, the approach supports service identification into a family of business processes.

Lanman *et al.* (2013) addressed the challenges to adopt SOA into an existing SPL. They described that SOA is appropriate to solve problems faced by Live Training Transformation (the product line strategy put in place by the United States Army Program Executive Office for Simulation, Training and Instrumentation), more specifically in the following three areas:

1 Lack of the ability to interoperate among systems;

2 Necessity of adaptation on-demand to user requirements; and

3 Capacity of processing and storing of a massive volume of data coming from a variety of unmanaged clients and servers.

According to them, the process of migrating to SOA is not away from SPL. Since, SPL process is useful for managing directly the development of SOA services. Almost all changes in the process were additions with the aim of covering particularities of service development. Thus, as the mentality of reusable services and shared development of product lines has many similarities, the impact on the development in a shared service oriented infrastructure is not as significant change in the developers ideal.

Gamez *et al.* (2015) used concepts from SPL to address the managing of variability in transaction services selection. Based on an SPL approach, they developed a strategy which considers a set of services that provide the same functionality as being part of a service family. They believed to reduce the complexity of the service selection step from repositories. This activity is part of a process of recursively constructing a value-add web service, which becomes more challenging with increasing number and diversity of services available on Internet (Mohabbati *et al.*, 2014).

Figure 2.10 shows the situation where service technologies are responsible for implementing variability mechanisms to customize specific and variable features. The core services are the main reusable assets that can be configured to build customized products allowing new dimensions of customizations as supporting for dynamic variation and exploitation of existing services.

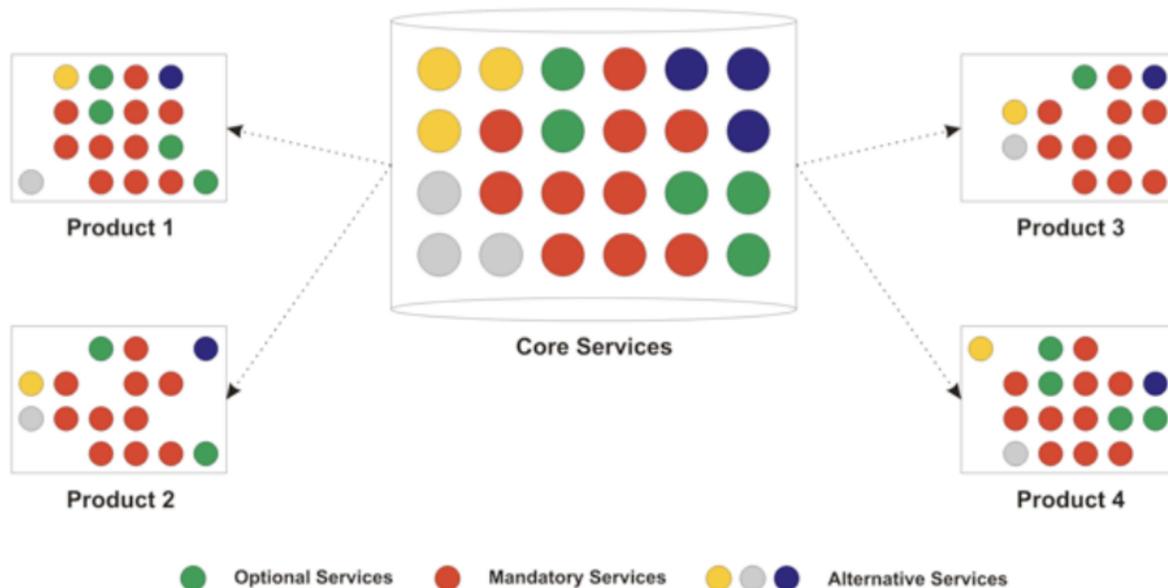


Figure 2.10: Services within a Product Line (Ribeiro, 2010).

2.2.3.1 Applying SPLE for Development of SOPL .

Mohabbati *et al.* (2014) proposed a method to support the development and customization of a family of SOC-based applications. It is an extension of the traditional SPL life-cycle widely used in the SPL development. In this way, as in the previous method, the proposal also defines two core activities: Service-Domain Engineering and Service-Application Engineering, as showed in Figure 2.11 and 2.12 respectively.

Service-domain engineering is responsible for making and evolving the reuse infrastructure. It analyzes the requirements and the scope of an SPL as a whole. The output of this process are common reusable business processes and services. In turn, service-application engineering uses the reference architecture to derive individual services (or to customize them). In the following, we describe the three main activities of service-domain engineering (Mohabbati *et al.*, 2014):

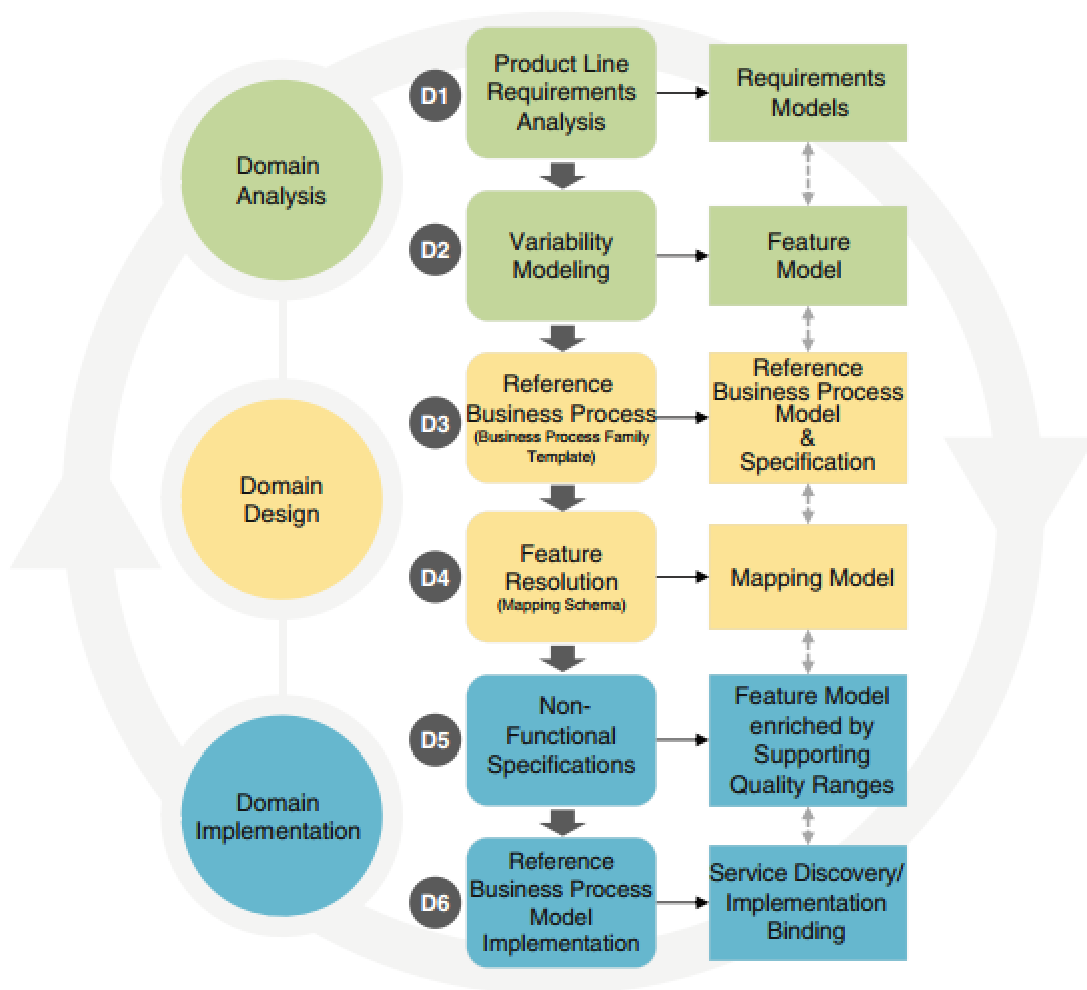


Figure 2.11: Service-Domain Engineering of an SOPL (Mohabbati *et al.*, 2014).

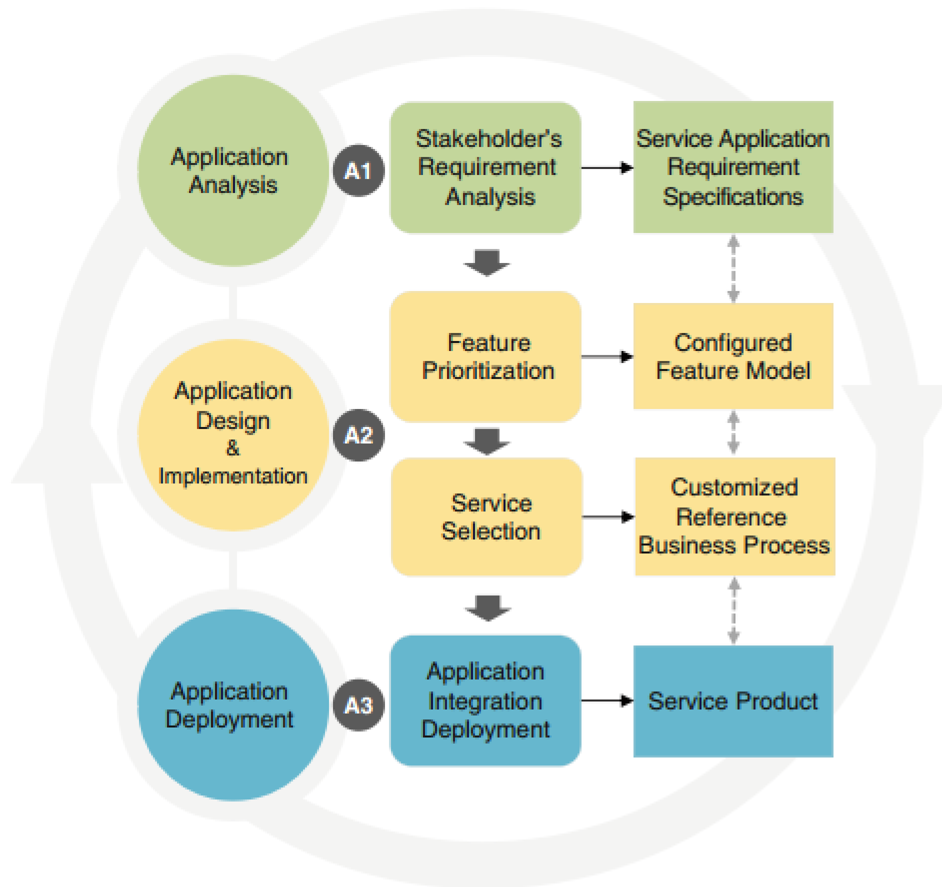


Figure 2.12: Service-Application Engineering of a SOPL (Mohabbati *et al.*, 2014).

- **Domain analysis** in SOPL covers the steps of SPL requirements engineering (D1) and the analysis of variability using feature modeling (D2). As the traditional SPL process, the first step performs activities such as elicitation of SPL business goals, requirements specification, and verification if requirements can be managed in terms of changes or refinements. Moreover, it is identified the commonality and variability among the requirements of several stakeholders. The outcome of this phase are requirements models that define the SPL scope and decide on the boundary of the SPL.

  Based on the requirements models, the second step performs its analysis and makes a feature model as artifact. This includes knowledge of variability associated to functional and non-functional requirements. Moreover, it can be used to indicate the permissible configuration space and serves as guideline in the customization process. In SOC, services are the basis of development, encapsulate functionality, and

provides individual non-functional properties. Thus, this proposal defines a feature as an increment in the service functionality for the context of SOPL. Therefore, based on the granularity levels, a feature can be implemented or associated to a composite service.

- **Domain design** phase (D3), a reference architecture and reusable components are developed based on the feature model of the requirement engineerings phase. Moreover, the model mapping (D4) is in charge to establish the mapping relationships between the features within the feature model and the corresponding activities specified within the reference business processes model. Services are responsible for activities of the reference business process. In other words, the activities of this phase produce a reference architecture as the behavioral model of features for the entire family and specify how features are composed at run-time.

- **Domain implementation** captures the non-functional requirements, since that they also can vary for different stakeholders. Thus, the D5 step is responsible by capturing and specify non-functional quality during the development of an SOPL. Thus, some features are annotated with quality ranges in the feature model. These annotations allow that engineers and developers evaluate progressively the impact of variant features selected according to the quality characteristics that the services provide. In the end, the reference business processes model is update to cover these changes.

Next, the main activities from the application engineering life-cycle are described. Basically, the steps that compose this life-cycle are related to customization and derivation. Thus, the service-application engineering uses the variability defined in domain engineering by selecting shared assets also developed in domain engineering (Mohabbati *et al.*, 2014):

- **Application Analysis** initially discovers and captures the functional and non-functional requirements of a stakeholder for deriving process variants, which can be deployed as the final product. Moreover, the preferences of the stakeholder also are stored to perform feature prioritization and selection. Others activities performed are the validation and verification of application requirements model according to stakeholder needs and SPL constraints.

- **Application Design and Implementation** uses feature model for managing and selecting variants that compose service product instances. Moreover, it is in this phase that occurs the prioritization of features and selection of sub-processes. All of

this process is performed in order to satisfy stakeholders requirements. Even at this stage, the feature model is specialized with the selection of optional features. Finally, the reference business process model is configured and corresponding services are selected.

- **Application Deployment** is the last phase of the process. It is responsible by the deployment of customized services. After deployment, the execution of the personalized services is monitored to ensure the compliance of the service with the stakeholders requirements.

2.3 CHAPTER SUMMARY

This chapter presented the main concepts about Software Product Lines, Service-Oriented Computing and Service-Oriented Product Lines. Regarding SPL, it discussed SPL development, motivations and benefits for applying this approach. Moreover, we highlighted the process of selecting and customizing shared assets during Application Engineering named as Product Derivation. This process is responsible to bring return on initial investment necessary to operate an SPL economically (Rabiser *et al.*, 2010).

In addition, this chapter also presented the basic concepts related to SOPL that is the combination of service-orientation and software product lines, and some important directions in the field. Next chapter presents the measurement framework used in this dissertation to evaluate the SPL implementations.

CHAPTER 3

AN OVERVIEW ON TECHNIQUES TO IMPLEMENT VARIABILITIES

In the previous Chapter, we described that the combination of SOC and SPL, called of Service-Oriented Product Line (SOPL), has started to receive growing attention by researchers and practitioners, mainly, the way how variability can be implemented in service-oriented product lines. Apel *et al.* (2008) and Smith and Lewis (2009) investigated the most suitable ways to develop SPL where services are the core assets.

This dissertation focuses on the derivation process of products more specifically on variability implementation in software product lines and service-oriented product line. In order to identify how efficient some techniques used in this activity are, we studied the main variability implementation techniques found in the literature. Moreover, this Chapter presents a measurement framework used in the case studies for performing a quantitative assessment of the chosen techniques with the support of variability mechanisms.

The remainder of this Chapter is organized as follows. Section 3.2 describes the framework used in the assessment. Section 3.3 discusses the metrics that compose the framework. Section 3.1 describes some techniques for implementing variabilities. Finally, Section 3.4 summarizes this Chapter.

3.1 TECHNIQUES FOR IMPLEMENTING VARIABILITIES

This section presents some techniques used in the literature for implementing variability, such as Conditional Compilation (CC2), Aspect-Oriented Programming (AOP), and Open Services Gateway initiative (OSGi).

3.1.1 Conditional Compilation (CC2)

Conditional Compilation is a well-known annotation-based technique that enables control over the code segments (Gacek and Anastasopoules, 2001). This technique is one the most elementary, yet most powerful, approach to enable variabilities in implementation artifacts (Matos Jr., 2008). Basically, for handling software variability, the preprocessor

directives indicate pieces of code that should be compiled or not, based on the value of preprocessor variables (Gaia *et al.*, 2014).

One advantage of this technique is to allow that the code be marked at different granularities from a single line of code to a whole file (Gaia *et al.*, 2014). As consequence, it is possible to encapsulate multiple implementations in a single module. To select the desired functionality should be used an appropriate conditional sentence. Moreover, conditional compilation enables the separation of variabilities by the use of directives, in other words, the insertion completes source files into code (Gacek and Anastasopoules, 2001).

Figure 3.1 shows a piece of code using conditional compilation to select lines of code that compose the feature "Bugs".

```
154     //#if ${Bugs} == "T"
155     public String sendBugtrackEmail(String nome, String assunto, String mensagem) throws EmailException{
156         SimpleEmail email = new SimpleEmail();
157         String msg;
158         email.setHostName("smtp.gmail.com");
159 
160         email.addTo("riseeventemail@gmail.com", "Bugtrack Event");
161         email.setFrom("riseeventemail@gmail.com", nome);
162         email.setSubject(assunto);
163         email.setMsg(mensagem);
164 
165         email.setAuthentication("riseeventemail@gmail.com", "senhasecreta");
166         email.setSslSmtpPort( "465" );
167         email.setSSLOnConnect(true);
168         email.setStartTLSEnabled(true);
169         email.setStartTLSRequired(true);
170 
171         email.send();
172         msg = "Email enviado com Sucesso";
173         return msg;
174     }
175     //#endif
```

Figure 3.1: Example of the conditional compilation implementation technique.

In line 154, there is a directive `//#if $Bugs == "T"` that indicates the beginning of the code belonging to the 'Bugs' feature. In line 175, there is a `#endif` directive that determines the end of the code associated to this feature. The identifiers used in the construction of these directives, in this case 'Bugs', are defined in a configuration file and are always associated with a boolean value. This value indicates the presence of the feature in the product.

3.1.2 Aspect-oriented Programming (AOP)

Aspect-Oriented Programming (AOP) has been proposed as a technique for improving separation of concerns, increasing reusability and software evolution. This technique

supports modularization of crosscutting concerns by providing abstractions that enable to separate and compose them to produce the overall system (Sant'anna *et al.*, 2003).

Aspects are the main mechanism of modularization that provide a new abstraction composing components (classes, methods, and so on.) at specific join points. They encapsulate a concern code that would be tangled with and scattered across the code of other concerns (Gaia *et al.*, 2014).

AspectJ is an extension of Java for aspect-oriented programming. For this extension, aspects are described in terms of pointcuts and advices: pointcuts are responsible for describing the 'join points' (well-defined points in the program flow, for example, method calls) and their values; advice is a method-like abstraction that defines code to be executed when a join point is reached.

Figure 3.2 shows an aspect implementation and how it can modularize a feature. Lines 9 to 11 show an intercept of execution before the method initialization. In turn, lines 15 to 35 present what will replace.

```
9      pointcut sendBugtrackEmail_aj(String nome, String assunto, String mensagem):
10         execution(String Email.sendBugtrackEmail(String     , String, String)) &&
11         args(nome, assunto, mensagem);
12
13
14     //util
15     String around(String nome, String assunto, String mensagem) throws EmailException:
16         sendBugtrackEmail_aj(nome, assunto, mensagem){
17         SimpleEmail email = new SimpleEmail();
18         String msg;
19         email.setHostName("smtp.gmail.com");
20
21         email.addTo("riseeventemail@gmail.com", "Bugtrack Event");
22         email.setFrom("riseeventemail@gmail.com", nome);
23         email.setSubject(assunto);
24         email.setMsg(mensagem);
25
26         email.setAuthentication("riseeventemail@gmail.com", "senhasecreta");
27         email.setSslSmtpPort( "465" );
28         email.setSSLOnConnect(true);
29         email.setStartTLSEnabled(true);
30         email.setStartTLSRequired(true);
31
32         email.send();
33         msg = "Email enviado com Sucesso";
34         return msg;
35     }
```

Figure 3.2: Example of AOP implementation technique.

3.1.3 Parameterization

The idea of parameterization as mechanism for implementing variability is to represent reusable software as a library of parameterized components, which has the behavior

determined by the values of the parameters.

Parameterization avoids code replication through centralized design around a set of variables. In other words, an instantiation determines the types that parameterize a class (or function) resulting in data types or components of software more flexible. However, despite the advantages (promote reuse, avoid replication of code and facilitate traceability), the centralizing of code by defining parameters is often a very complex task (Gacek and Anastasopoules, 2001).

```
14 public class Email {
15
16     public void sendNotification(User user, Review review) throws EmailException{
17
18         if(VariantsConfiguration.REVIEW_ROUND_OF_REVIEW == true){
19             sendRoundNotification(review, user);
20         }
21         if(VariantsConfiguration.BUGS == true){
22             sendBugtrackEmail(user.getNome(), review.getAssunto(), review.getMensagem());
23         }
24     }
25
26     //{ReviewRoundofReview}
27     private void sendRoundNotification(Review review, User user) throws EmailException{
67
68
69     //{Bugs}
70     private String sendBugtrackEmail(String nome, String assunto, String mensagem) throws EmailException{
```

Figure 3.3: Example of the parameterization implementation technique.

```
 8 ReviewRoundofReview=false
 9 ReviewSimpleReview=true
10 Bugs=true
11 EventProgram=true
12 EventImportantDates=true
```

Figure 3.4: File config.properties.

A common use of this technique is the parameterization of a data structure, for example, a stack, which contains a set of elements with the type that can be set through a parameter.

Figure 3.3 shows part of the code that implements by parameterization the feature ”Bugs”. It is possible to identify in line 21, the verification used to enable or not the feature code. Figure 3.4 shows part of file that contains the parameters values used in verification.

3.1.4 Delegation

Delegation is an object oriented technique that removes variabilities from the classes which implement commonality and place them in specific classes. In this way, objects extend

their functionality by forwarding requests to other objects. For this, the delegating objects hold references to objects which are able to deal with specific variations (Matos Jr., 2008).

In this context, when a delegating object requires an execution of code related to some variation (optional or alternative), delegation objects are responsible to provide these functionalities. Figure 3.5 shows this situation (Matos Jr., 2008).

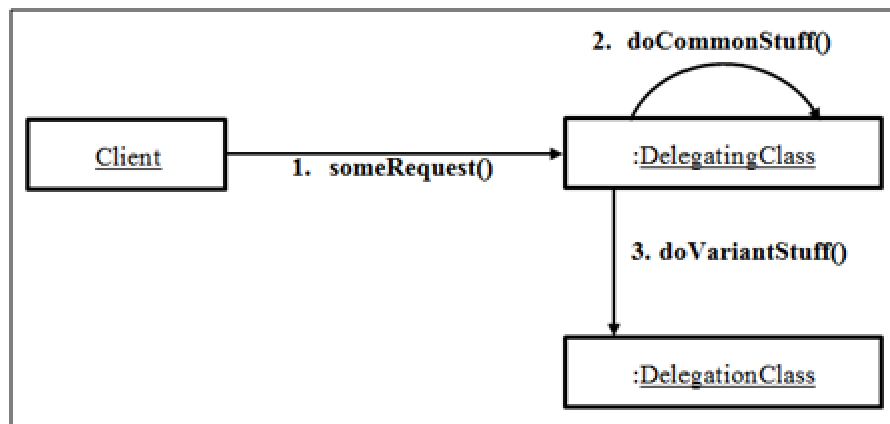


Figure 3.5: Variability by delegation technique.

Delegation works will way when deals with optional features. However, it has difficulty with alternative ones. In the first situation, there is only one indirection while in the last one, many indirections are necessary at the variation point. Moreover, other problem emerge in cases where the number of variants (*e.g.*, of an object function) starts to grow significantly. In many cases, additional delegation and probably source files are required (Gacek and Anastasopoules, 2001).

3.1.5 Inheritance

Object-oriented languages provide inheritance as a mechanism of classes specialization. With this technique, specialized classes are able to refine the structure and behavior of more general classes (parent classes). Thus, new attributes or operations can be defined and also redefine the behavior of some operations present in parent class (known as method override) (Matos Jr., 2008).

In SPL context, positive variabilities (features implemented separately in distinct modules: classes, packages, files) can be implemented as refinements of classes, adding the necessary behavior and structure to deal with each variability (Figure 3.6). However, the growth of the amount of different variabilities leads also to a growth of the number of subclasses that in many cases increase the complexity producing an unclear inheritance

tree (Gacek and Anastasopoules, 2001).

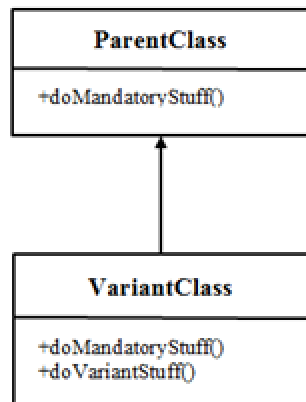


Figure 3.6: Variability by Inheritance technique.

3.1.6 Libraries

Programmers use software libraries during the software development. These libraries are a suite of code and data that become more easy the development process. In the context of software development, the libraries are divided into two subgroups (static libraries and dynamic link libraries), which can be used to provide variability to SPLs.

Static libraries are a simple and limited alternative to allow software variability. They contain a set of external functions that can be linked to the code after it has been compiled. Each these libraries providing some kind of variability support. For this purpose, it is important to ensure that the signatures of the functions are known to the compiled code (and will remain unchanged). Configuration management tools are useful during the libraries selection process (Gacek and Anastasopoules, 2001).

Dynamic link libraries (DLL) are more robust than static libraries. They are loaded in dynamic way (in the same process space or in separated like independent EXE's) whenever the applications require. Separation of variability is reached by developing distinct controls where the operating system is responsible linking the DLLs to the application and resolve the addresses for all methods (Gacek and Anastasopoules, 2001).

3.1.7 Open Services Gateway Initiative (OSGi)

OSGi technology is a set of specifications that defines an open, common architecture to develop, deploy and manage services in a coordinated way for Java language. These specifications aim to facilitate the development, reduce the complexity and increase the

reuse of services developed. Another important characteristic is allow that applications discover and use services provided by other applications running inside the same OSGi platform (Ribeiro, 2010).

OSGi is a suitable technology for implementing variation points because offers an easy way to include new components and services without being required to recompile the whole system. In other words, it is possible to perform changes between different implementations in a dynamic way (Almeida et al., 2008).

Figures 3.7 and 3.8 show OSGi implementation for the same functionality presented in previous subsection. As the features are implemented as services, it is required a class for starting or stopping them. Figure 3.7 shows this part of the code. In the lines 15 and 20 can be observed the methods responsible for starting and stopping the service respectively. Figure 3.8 shows the class that implements the feature.

```
1   package bundle_bugs;
2
3   import org.osgi.framework.BundleActivator;
4   import org.osgi.framework.BundleContext;
5
6   public class Activator implements BundleActivator {
7
8       private static BundleContext context;
9
10      static BundleContext getContext() {
11          return context;
12      }
13
14
15      public void start(BundleContext bundleContext) throws Exception {
16          Activator.context = bundleContext;
17      }
18
19
20      public void stop(BundleContext bundleContext) throws Exception {
21          Activator.context = null;
22      }
```

Figure 3.7: Service ativator.

```
1   package bugs_util;
2
3   import org.apache.commons.mail.EmailException;
4   import org.apache.commons.mail.SimpleEmail;
5
6   public class EmailBugs {
7
8       //{Bugs}
9       public String sendBugtrackEmail(String nome, String assunto, String mensagem) throws EmailException{
10          SimpleEmail email = new SimpleEmail();
11          String msg;
```

Figure 3.8: Example of the OSGi implementation technique.

3.2 THE MEASUREMENT FRAMEWORK

According to Sant'anna *et al.* (2003), the measure of internal attributes such as size and instability have no meaning when observed in isolation. However, the metrics are more effective when combined to produce a measurement framework, which enables software engineers to understand and interpret the meanings of the measured data.

For this reason, and following the idea of relationship among metrics proposed by Ribeiro *et al.* (2011), a measurement framework was developed. It provides support for assessing the code quality in context of SPL and SOPL projects. The metrics reused in this framework for data collect were defined or used in different studies such as McCabe (1976), Sant'anna *et al.* (2003), Almeida *et al.* (2008) and Gaia *et al.* (2014). Figure 3.9 shows the measurement framework.

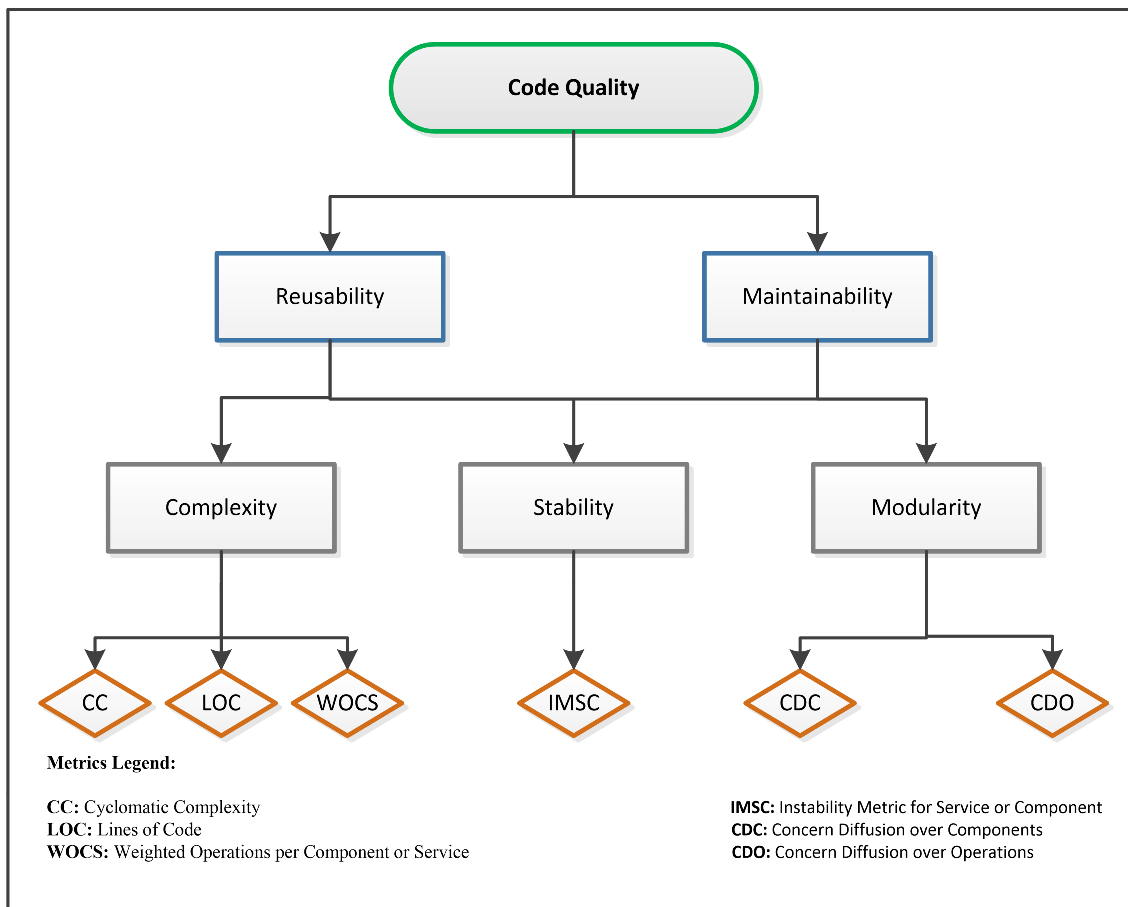


Figure 3.9: The Measurement Framework.

As it can be seen in Figure 3.9, the suite developed is composed of six metrics. These metrics grouped provide the necessary conditions to analyze code quality, and thus meets

the requirements of measurement.

Moreover, three different levels compose the framework: code quality is the response variable studied in this work. For this purpose, reusability and maintainability are the qualities observed in the system. These are influenced by the factors level (complexity, stability and modularity) categorized and quantified using the defined metrics.

3.3 METRICS FOR CODE QUALITY EVALUATION

In this section, we discuss the set of metrics used to compose the framework that will be responsible by quantitatively evaluate each implementation technique. These metrics are grouped according to three criteria: metrics to evaluate the complexity of the implementation artifacts; stability metrics used to measure instability of source code; finally metrics related to modularity of the implementation.

1. **M1. Complexity Metrics**

   A critical issue in SPLs domain is how to perform an efficient separation of concerns and modularize the systems in order to have modules and components well defined, testable and more reusable (Sant'anna *et al.*, 2003). In this scenario, structural dependencies between services and components have a significant influence on system complexity (Ribeiro, 2010). In this context, the following metrics can be used to assess the complexity of source code artifacts:

   Cyclomatic Complexity (CC1) - Cyclomatic complexity is measured in terms of the number of paths through a program. Values obtained with aid of this metric are useful to keep the size of components or services manageable and allow the testing of all independent paths. In this sense, the cyclomatic complexity metric can be used to analyze the complexity of each component developed (Almeida *et al.*, 2008). The cyclomatic complexity is calculated from a connected graph of the service or component:

$$\boldsymbol{CC1} = E - N + p, \textit{ where:} \tag{3.1}$$

   E = the number of edges of the graph.

   N = the number of nodes of the graph.

   p = the number of connected services or components.

According to McCabe (1976), modules with CC1 among 1 and 10 are simple programs. Values between 11 and 20 represent programs more complex with moderate risk. In turn, values ranging between 21 and 50 represent a complex program with high risk. Values of CC1 above 50 indicate untestable programs.

Lines of Code (LOC) - Some metrics were proposed for measuring size. One of the simplest one is lines of code that counts all lines for each implementation, excluding comments and blank lines. This measure provides grounds for comparison between systems, but different programming styles can bias the results of this metric application. Because of this, it is necessary to ensure the same programming style in all projects developed (Sant'anna *et al.*, 2003).

Weighted Operations per Component or Service (WOCS) - The quantity of time and effort required for developing and maintaining a service or component can be measured by the number of methods and their complexity. Hence, the Weighted Operations per Component or Service (WOCS) determines the complexity of service or component based on its operations (methods) that will be required by other services or components (Ribeiro, 2010). Consider a component or service C with operations *O1,..., On*. Let *c1, ..., cn* be the complexity of the operation, then:

$$\boldsymbol{WOCSC} = c1 + \ldots + cn \tag{3.2}$$

For this metric, the number of parameters of an operation is the key of complexity. Thus, operations with many parameters are more likely to be complex than other which requires few parameters. In this way, the complexity of operation Ok is defined as: ck = αk + 1, where αk denotes the number of formal parameters of Ok (Matos Jr., 2008).

2. **M2. Stability Metrics**

   A single change can begin a cascade of changes of independent modules or services when there is a fragile design and difficulties in reusing. Thus, both the extent of change and its impact cannot be predicted by the designer (Sant'anna *et al.*, 2003). In this context, the metric to measure instability of source code artifacts used was:

   Instability Metric for Service or Component (IMSC) - IMSC is supported

by *fan.in* and *fan.out*, where *fan.in* (for function 'A') is calculated by the number of functions that call function 'A'. In turn, *fan.out* is the reverse procedure, that is, the number of functions that the function 'A' performs requests. The value measured by IMSC reflects interaction between services or components through sending and receiving messages (Ribeiro, 2010). Thus, this metric is defined as following:

$$\boldsymbol{IMCS} = \frac{fan.out}{fan.in + fan.out} \tag{3.3}$$

The instability metric has range [0, 1], where I = 0 indicates a maximally stable category and I = 1 indicates a maximally instable category, which means that the service or component is very instable and most difficult the maintenance.

3. **M3. Modularity Metrics**

Decomposition mechanisms used both in design and implementation are closely related to Separation of Concerns (SoC). Concerns are an alternative for decomposing software in smaller parts and at the same time more manageable and comprehensible. Moreover, they are regarded equivalent to features for some researchers (Gaia *et al.*, 2014). Thus, the following metrics were chosen to support the modularity analysis of source code artifacts:

Concerns Diffusion over Components (CDC) - This metric quantifies the degree of feature scattering considering the granularity level of components. *CDC* counts the number of components whose purpose is to contribute to the implementation of a concern. A high value for *CDC* indicates that a feature implementation can be scattered (Gaia *et al.*, 2014).

Concerns Diffusion over Operations (CDO) - Similar to *CDC*, Concern Diffusion over Operations also quantifies the degree of feature scattering, but to the level of granularity of methods. It counts the number of methods and constructors performing a feature (Gaia *et al.*, 2014).

These two last metrics were adapted (as in Gaia *et al.* (2014)) to consider the ratio of the measured value to the total value on that release, it means, for example, that the relative CDC calculated represents the percentage of classes that are used to implement the feature. Thus, these relative metrics allow to analyze together the

set of metric values for all features. For both metrics, a lower value implies a better result.

3.4 CHAPTER SUMMARY

In this Chapter, we presented the measurement framework which will be used to compare SPLs implementations, as well as the set of metrics for code evaluation. The metrics are related to complexity, stability and modularity. We also discussed some of the most relevant techniques to handle variabilities at the implementation level.

Next chapter presents a case study to investigate how efficient are some techniques used to implement variability in software product lines and service-oriented product line.

CHAPTER 4

THE CASE STUDY

As stated in Chapter 2, through variability management, SPL engineering provides required conditions to differentiate products in the same software domain. In order to do it, software engineers have to define which techniques are the most suitable to implement the variabilities defined in the management process. Aiming to aid this process, we analyzed some techniques through a case study.

Case study is a commonly used research strategy in several areas including software engineering that has as objective to improve the software engineering process and the resultant software products (Runeson *et al.*, 2012). Yin (2009) says that case study is an empirical investigation of a contemporary phenomenon within its real-life context, mainly when the boundaries between phenomenon and context are unclear.

Runeson *et al.* (2012) claim that the strategies adopted during the execution of the research must be defined based on the purposes of the case study: exploratory, descriptive, explanatory, and improving. This Chapter presents an exploratory case study aiming at investigating different variability mechanisms to implement software product lines.

The remainder of this Chapter is organized as follows. Section 4.1 presents the research design with the most important definitions related to the case study research. The results and findings are described in Section 4.2. Section 4.3 discusses the threats to validity; and finally, Section 4.4 presents the Chapter summary.

4.1 CASE STUDY PROTOCOL

Runeson *et al.* (2012) define a structure for an exploratory case study as means to overcome existing limitation resources and time limits. This research follows the protocol definition proposed by them in the next sub-sections.

4.1.1 Objective

One of the goals of this case study is to make a contribution for the body of knowledge on the development process of SPL and SOPL, mainly, in the variability implementation. Thus, the overall goal of the case study is to investigate the code quality inside this context. To do

that, we analyzed some implementations performed through some variability techniques most used for developing SPL systems (conditional compilation and aspect-oriented programming). Moreover, we also used one mechanism (Open services gateway initiative) for implementing SOPL and identifying its behavior in reuse.

Although conditional compilation is not a new variability techniques, we decided to choose it because it is still a state-of-the practice option adopted in SPL industry (Gaia *et al.*, 2014). Moreover, many large open-source systems (including the Linux kernel) have also used CC for variability realization (Zhang *et al.*, 2016). In turn, aspect-oriented programming is mainly used in code tracing, logging and exception handling. Due to this, there is still not enough evidence of its successful practical usage in variability realization, what justifying its study (Zhang *et al.*, 2016). Finally, OSGi was chosen because of its support to implementing components and managing their interaction and lifecycle, unlike of Web Services that has some limitations in terms of the degree of interaction between the components (services, in this case) Almeida *et al.* (2008).

This objective was refined through the GQM (Goal/Question/Metric) paradigm (Basili *et al.*, 1994) to obtain the specification of a measurement system that has as targeting a particular set of issues and of rules for the interpretation of the measurement data. In other words, the case study must be conduced in accordance with a specific goal, a set of questions that represent the operational definition of the goal, and the related metrics that must be collected to aid answering the questions (Ribeiro, 2010). Thus, following the GQM template, it aims at:

Goal. *Analyze the* variability implementation techniques (conditional compilation, OSGi, and aspects) *for the purpose of* evaluation *with respect to* reusability, maintainability, complexity, stability and modularity *from the point of view of* software engineers and researchers *in the context of* a software product line and service-oriented product line project.

Hypotheses. According to Kitchenham *et al.* (1995), it is important to define the expected effects for the object being analyzed. Moreover, they also state that formally, we can never prove hypotheses, we can only disprove them. Thus, we should also define what is not expected. Because of this, it was established hypotheses which the experiment will accept or reject. For this study, it means that there is no difference among the techniques for implementing variabilities:

$$
\begin{aligned}
H_0 \quad &: \quad CC2_{CC1} = AOP_{CC1} = OSGi_{CC1} \\
&: \quad CC2_{LOC} = AOP_{LOC} = OSGi_{LOC} \\
&: \quad CC2_{WOCS} = AOP_{WOCS} = OSGi_{WOCS} \\
&: \quad CC2_{IMSC} = AOP_{IMSC} = OSGi_{IMSC} \\
&: \quad CC2_{CDC} = AOP_{CDC} = OSGi_{CDC} \\
&: \quad CC2_{CDO} = AOP_{CDO} = OSGi_{CDO} \qquad (4.1)
\end{aligned}
$$

In turn, the alternatives hypotheses are valid when the null hypotheses are rejected. In this scenario, there are differences among the technologies for implementing variabilities studied:

$$
\begin{aligned}
H_a \quad &: \quad CC2_{CC1} \neq AOP_{CC1} \text{ or } CC2_{CC1} \neq OSGi_{CC1} \text{ or } AOP_{CC1} \neq OSGi_{CC1} \\
&: \quad CC2_{LOC} \neq AOP_{LOC} \text{ or } CC2_{LOC} \neq OSGi_{LOC} \text{ or } AOP_{LOC} \neq OSGi_{LOC} \\
&: \quad CC2_{WOCS} \neq AOP_{WOCS} \text{ or } CC2_{WOCS} \neq OSGi_{WOCS} \text{ or } AOP_{WOCS} \neq OSGi_{WOCS} \\
&: \quad CC2_{IMSC} \neq AOP_{IMSC} \text{ or } CC2_{IMSC} \neq OSGi_{IMSC} \text{ or } AOP_{IMSC} \neq OSGi_{IMSC} \\
&: \quad CC2_{CDC} \neq AOP_{CDC} \text{ or } CC2_{CDC} \neq OSGi_{CDC} \text{ or } AOP_{CDC} \neq OSGi_{CDC} \\
&: \quad CC2_{CDO} \neq AOP_{CDO} \text{ or } CC2_{CDO} \neq OSGi_{CDO} \text{ or } AOP_{CDO} \neq OSGi_{CDO} \qquad (4.2)
\end{aligned}
$$

4.1.2 The Case

The independent variable of this study is the variability mechanism used to implement the SPL: compilation conditional, Aspect-Oriented Programming and Open Services Gateway Initiative. In order to evaluate these different variability implementation techniques, this case study was developed.

The SPL under study is an academic simulator of a warehouse SPL implemented by the RiSE Labs group. The development team was composed of one Ph.D. student and one M.Sc. student. This SPL was inspired on the scenario proposed by Apel *et al.* (2008) which provides conditions to investigate the source code related to modularity and software variability. Figure 4.1 shows a screenshot of the most complete product of the Warehouse SPL developed by RiSE Labs group using conditional compilation as variability implementation technique.

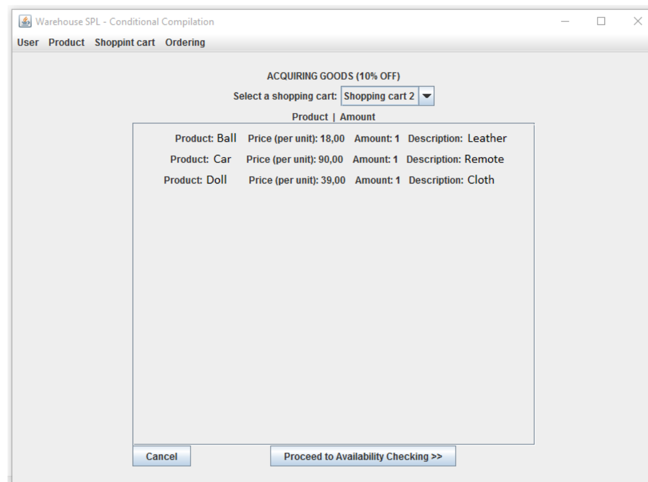


Figure 4.1: Screenshot a Warehouse SPL product implemented with conditional compilation.

The Warehouse SPL is composed of 23 features that allow the simulation of basic functionalities such as get customer requests, checking the availability of ordered goods, ordering the goods from an inventory and billing.

In the first phase, the students developed the SPL from the scratch using conditional compilation as variability mechanism. Three releases were developed where the release 1 contains the core asset of SPL and the subsequent ones incorporate changes to implement others features. The other implementations using AOP and OSGi were developed based on this first release. For OSGi, the services were used to resolve features, it means that services are the main core assets, where for each feature, a corresponding service was developed in the implemented SPL. Table 4.1 shows all developed scenarios that were achieved by including, changing, or removing features. Figure 4.2 shows the feature model of the SPL.

As aforementioned, a set of scenarios that represent the changes performed among the releases were implemented. Two change scenarios for each mechanism resulted in a total of nine releases. Figure 4.3 and 4.4 show some measures about the size of the Warehouse SPL in terms of the number of components (class and class refinements and aspects), and methods.

Columns R.1 to R.3 represent the different releases of the SPL and the number of components varies from 41 (in release 1 for conditional compilation) to 72 (in release 3 for OSGi). The number of components required in all releases by CC2 implementation is smaller than AOP and OSGi (which requires more components).

As the number of components, the OSGi implementation is the one that requires more

Table 4.1: Warehoure features for releases.

	Release 1	Release 2	Release 3
Access_Control	x	x	x
User	x	x	x
Add_User	x	x	x
Edit_User	x	x	x
Product	x	x	x
Add_Product	x	x	x
Edit_Product	x	x	x
Acquisition	x	x	x
Shopping_Cart	x	x	x
New_Shopping_Cart	x	x	x
Edit_Shopping_Cart	x	x	x
Ordering	x	x	x
Availability_Checking	x	x	x
Payment	x	x	x
Shipping	x	x	x
Normal_Billing	x		
Billing_withDiscounting		x	x
Debit_Card	x	x	x
Credit_Card		x	x
Remove_User			x
Remove_Product			x
Remove_Shopping_Cart			x
Status_Monitoring			x

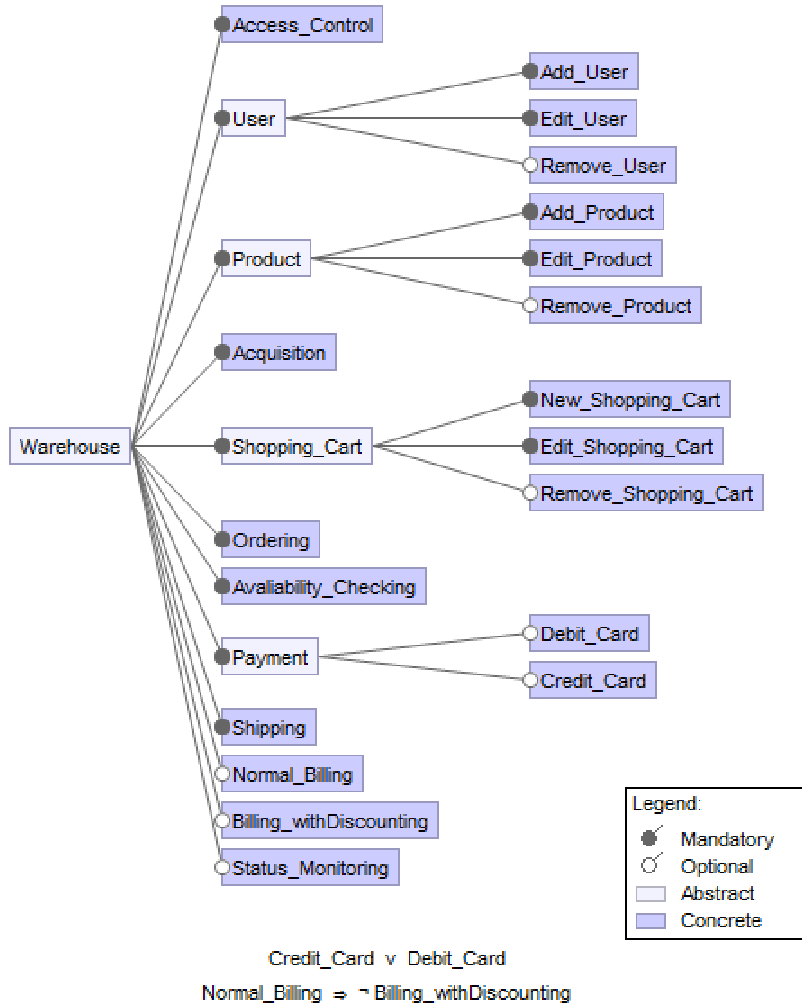


Figure 4.2: Warehoure feature model.

methods (278) followed by AOP and CC2. It occurs because both AOP as OSGi need to add methods that are used as part of configuration of these variability mechanisms. Thus, this characteristic makes the number of classes and methods increase in relation to conditional compilation.

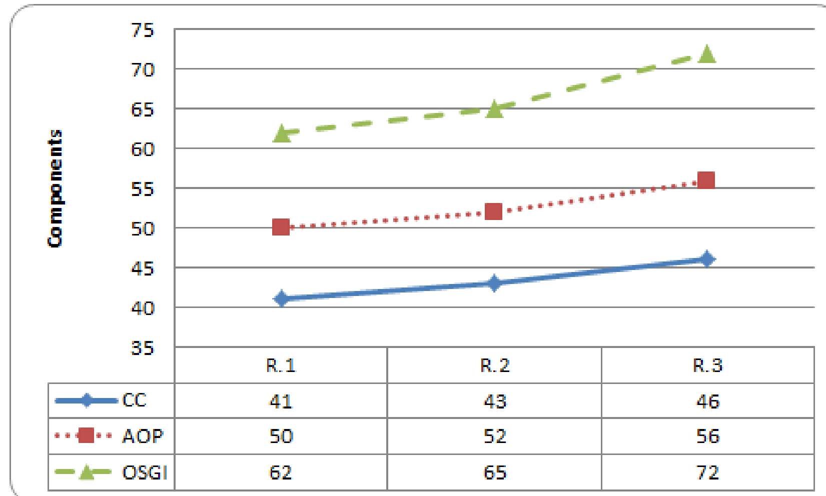


Figure 4.3: Warehouse SPL implementation (Components).

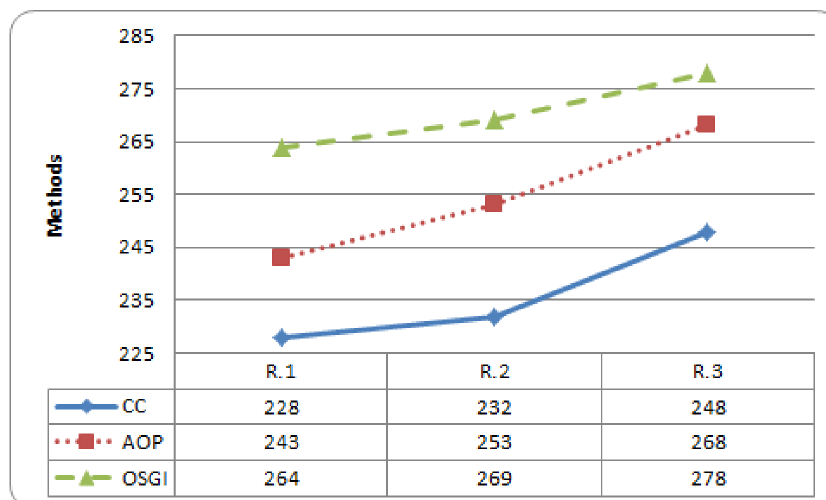


Figure 4.4: Warehouse SPL implementation (Methods)

4.1.3 Units of Analysis

The unit of analysis in this case of study is the evaluation of the variability implementation techniques. The execution process of the case study was divided in these steps. Initially, we developed and collected the data for analysis from the SPL implemented using conditional compilation. Then, we performed this same process using AOP and finalized with the SOPL developed with OSGi.

4.1.4 Case Study Research Questions

According to Runeson *et al.* (2012), the desired knowledge during the case study must be based on the research questions. Furthermore, the resolution of these questions means that the case study achieved its objectives.

Thus, we defined the following research questions (RQ) in this case study:

RQ 1: How complex are the services and components developed using CC2, AOP or OSGi?

Rationale: this question investigates structural dependencies between components or services. This kind of dependency has influence on the system complexity, thus, the goal is to identify which mechanism produces best results.

RQ 2: How stable are the services and components developed using CC2, AOP or OSGi?

Rationale: this research question identifies which mechanism produces better results for stability. With a fragile design and development may be triggered an independent cascade of changes when a single change is performed in a component. Thus, this aspect is very important.

RQ 3: How modular are the services and components developed using CC2, AOP or OSGi?

Rationale: developers often are aiming to implement modular software, and at the same time, more manageable and comprehensible. In this context, RQ 3 aims at supporting our analysis about the modularity level of services and components.

4.1.5 Data Collection

Runeson *et al.* (2012) state that the main decisions about methods for data collection are defined at design time for the case study. Moreover, they present three principles for data collection: (i) use multiple sources of data; (ii) create a case study database; and (iii) validate data and maintain a chain of evidence.

They also claim that available sources of expected data influence the methods of data collection. An example of this occurs when researchers intend to obtain information from software engineers with interviews, focus groups, and questionnaire surveys which are clearly methods of data collection indicated (Runeson *et al.*, 2012).

Data collection can be classified in regard to degree of involvement of software engineers. In the first degree (inquisitive techniques), researchers have direct contact with the interviewees and collect data in real time (*e.g.* interviews). In the second degree (observational techniques), the engineers involvement occurs in indirect way, in other words, the researcher collects the data without interaction with the interviewees (*e.g.* instrumenting systems). Finally in the third degree, there is only the study of work artifacts (Lethbridge *et al.*, 2005).

For this case study, we used the method of third degree once that researchers had contact only with the software artifacts as data source. Thus, we adopted the static analysis (generates findings without executing the program) of a software as data collection method. This technique is indicated when the program behavior is not the focus of the study.

In this technique, the code analysis extracts a very large amount of information. However, to get useful information, it is necessary parsers and other analysis tools, which cannot always be mature enough (Lethbridge *et al.*, 2005). We overcome this obstacle using the plugin Metrics[1] for the Eclipse platform, which provides conditions to measure several metrics.

4.1.6 Data Analysis

We define the strategy for data analysis based on the theoretical propositions of (Runeson *et al.*, 2012). They gave support in decisions about what kind of data should be object of study or ignored. These propositions also assist in the development of a suitable data collection plan to literature review and research questions.

In order to perform this study, a quantitative analysis was applied in the data collected.

[1]http://sourceforge.net/projects/metrics/

This analysis was performed in accordance with the literature review and the framework previously presented (Section 3.2). It was divided in complexity, modularity, and stability.

Each subsection of analysis describes the results obtained with the application of the metrics on the implemented releases. It must be highlighted that the values measured for complexity, modularity, and stability represent the level of influence which these quality factors have on code reusability and maintainability.

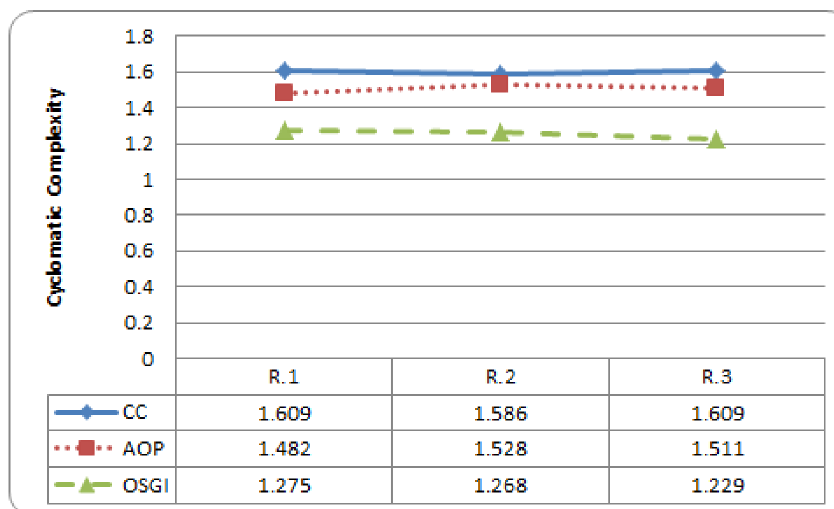
4.2 RESULTS AND FINDINGS

This section presents the data collected after implementation. Moreover, it is performed a quantitative analysis related to complexity, stability and modularity.

4.2.1 Complexity Analysis

Three metrics were used in the process of analysis such as presented in Figure 3.2 : cyclomatic complexity, lines of code, and weighted operations per component or Service.

Figure 4.5 shows a graphic with mean values measured of cyclomatic complexity for the releases of Warehouse. As it can be seen, OSGi mechanism has the lowest cyclomatic complexity compared to the other techniques. Among its releases, the variation practically does not exist, remaining at roughly 1.2. CC2 presents the higher mean however, in release 2, the AOP mechanism had mean values were very close, with values close to 1.5.



	R.1	R.2	R.3
CC	1.609	1.586	1.609
AOP	1.482	1.528	1.511
OSGI	1.275	1.268	1.229

Figure 4.5: Warehouse Cyclomatic Complexity.

Unlike what happened with CC2 analysis, OSGi is the variability mechanism that needs more lines of code among the techniques studied. When conditional compilation

is used in the implementation, the range was 2487 to 3309 lines among the releases. The variation between products for AOP was lower between 2649 to 2933. Finally, the range for OSGi was of 2992 to 3309. The differences of lines of code used among the mechanism is explained in part by the fact that CC2 inserts its preconditions in pieces of code commented only in the base product. In this way, when the others releases are derived the preconditions are removed. Moreover, the own characteristics of the other techniques make them need more lines of code. Figure 4.6 shows the number of lines of code for each release discussed.

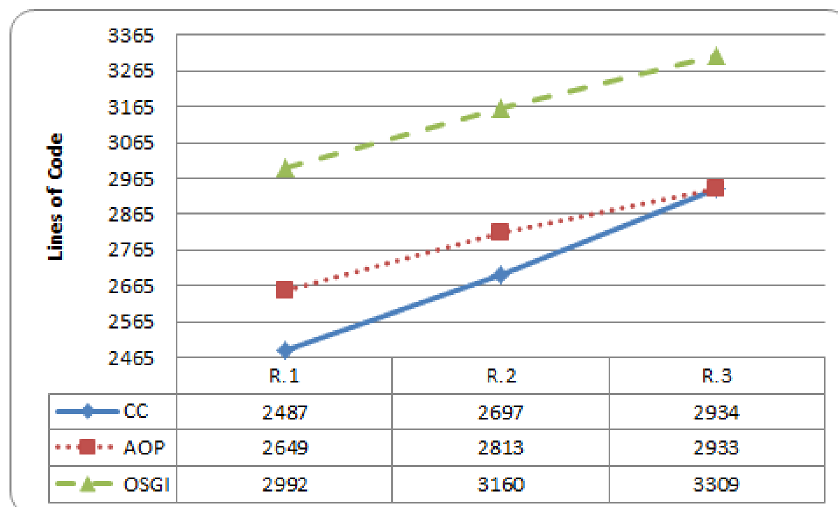


Figure 4.6: Warehouse Lines of Code.

The last metric used to measure the Warehouse code complexity was weighted operations per components or service. For this metric, OSGi had values lower than others. The difference among the values measured is considerable, once that OSGi average is half when compared to AOP. Conditional compilation had the higher results, it is explained by the fact that it had the implementation with fewer number of methods and classes among the techniques studied. Thus, the number of parameters (key to calculate this metric) is distributed in less methods, which leads to an increase of the mean for CC2. Figure 4.7 shows the results discussed for this metric.

4.2.2 Stability Analysis

The results were computed based on the instability metric presented in Section 3.2. Observing the graphic showed in Figure 4.8 is possible to identify that OSGi is the mechanism with components or services more unstable in all releases analyzed. In turn, AOP and CC2 had results very similar.

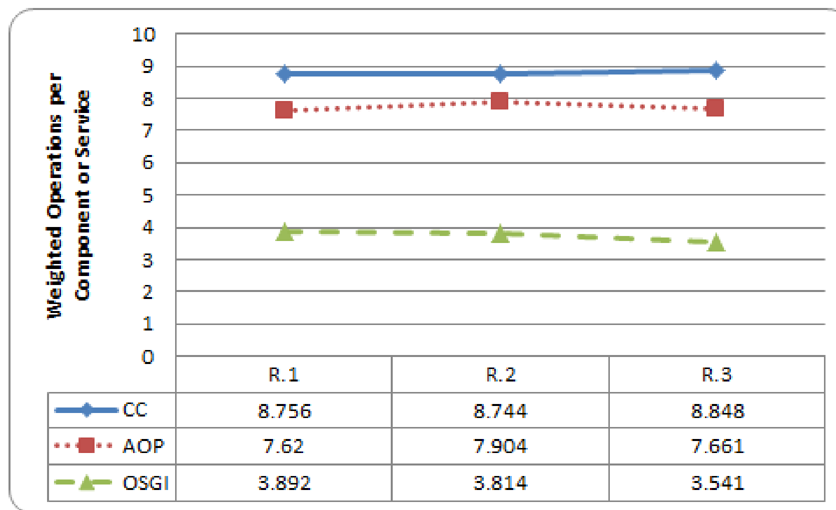


Figure 4.7: Warehouse Weighted Operations.

The values related to OSGi implementation can be explained through the way that are defined the bundle context and additional code for providing discoverability for the services. Thus, due to dynamic characteristic of this technique, it is not possible to separate the persistence from the Bundle, resulting in a high coupling among services that produces instability values more elevated.

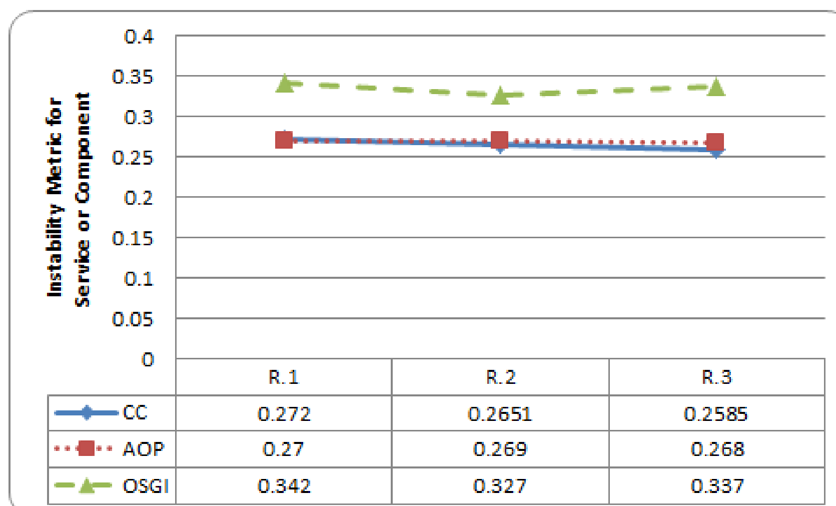


Figure 4.8: Warehouse Instability.

4.2.3 Modularity Analysis

The metrics chosen to perform this analysis were concerns diffusion over components and concerns diffusion over operations (as shown in Figure 3.2). As previously discussed, these

metrics were adapted and we considered the ratio of the measured value to the total value on each release.

Regarding to CDC, Figure 4.9 shows that OSGi implementation is the most modular when the criterion observed refers to the degree of feature scattering on the level of components. AOP had mean with values intermediate, and CC2 implementation had the worse result for this metric with mean always higher than the other variability mechanisms.

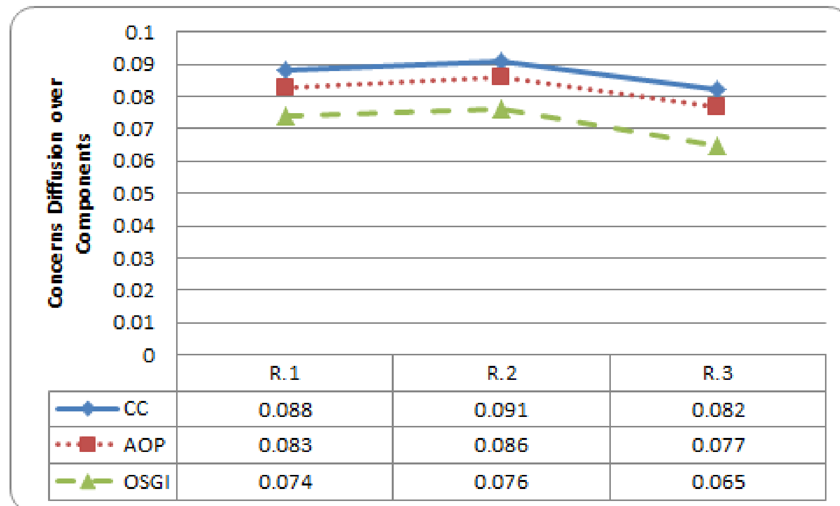


Figure 4.9: Concerns Diffusion over Components for Warehouse

This result was expected due the intrinsic features of the techniques analyzed. During the development process using OSGi each feature was isolated in an service what by itself provides some modularity. In an analogous way, features were inserted in aspects when the development used AOP mechanism. Nevertheless, CC2 implementation does not have a native way to isolate features.

Finally, CDO was the last metric applied in this case study. In contrast with the results of CDC, the values achieved do not clarify which mechanism got better results. They are very similar and sometimes a technique has slight superiority. Figure 4.10 shows this situation.

4.2.4 Descriptive and Exploratory Analysis

We used the nonparametric Kruskal-Wallis test (KW) to compare the different mechanisms of variability because in most of the samples, the normality hypothesis was rejected by the Shapiro-Wilk test (SW) which prevents use of variance analysis.

The Kruskal-Wallis (KW) is an extension of the Wilcoxon-Mann-Whitney test (WMW). It is a nonparametric test used to compare three or more populations. The underlying

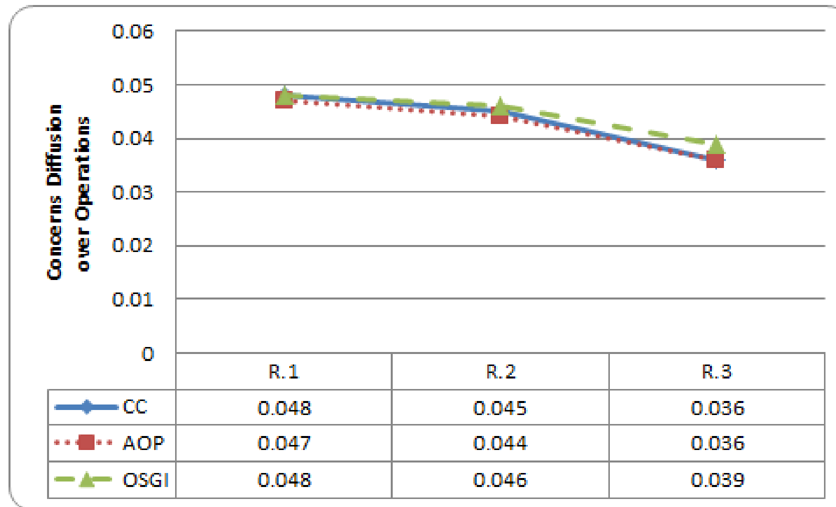


Figure 4.10: Concerns Diffusion over Operations for Warehouse

hypothesis test is:

H_0: All populations have identical distributions.

H_1: At least two populations have different distributions.

The results are shown in Table 4.2. The comparison of variability mechanisms indicates that LOC and CDO did not present statistically significant difference at the level of 5%. The variables Instability had a borderline significance. It means that there is a difference only in one of the mechanisms compared to others, as shown in the Boxplot of Figure 4.11, where only OSGi had higher values.

Table 4.2: Results of the quantitative analyses for Warehoure.

Variable	Kruskal-Wallis Chi-squared	H-value	df
Cyclomatic Complexity	7.2000	5.6	2
Weighted Operations	7.2000	5.6	2
Lines of Code	5.4222	5.6	2
Instability	5.6000	5.6	2
Concerns Diffusion over Components	5.9556	5.6	2
Concerns Diffusion over Operations	0.5650	5.6	2

Thus, these results rejected the null hypotheses defined for the case study except for CDO metric that did not present significant results in this study.

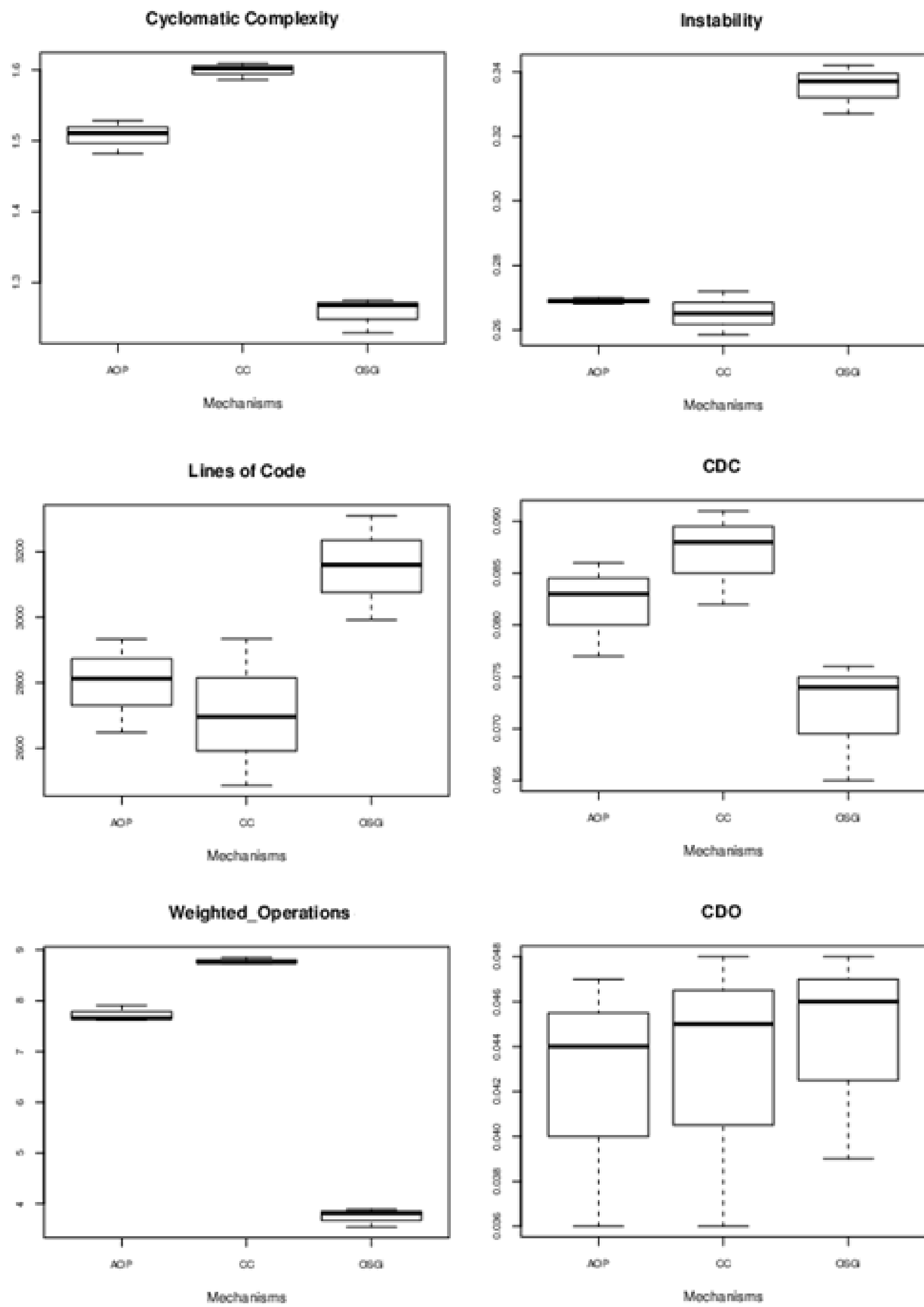


Figure 4.11: Boxplot comparing the variability mechanisms according to aspects complexity, stability and modularity.

4.2.5 Results Summary

Table 4.3 summarizes the results achieved in this case study. They showed that there are some aspects that must be considered when using different techniques to implement variability in SPL or SOPL systems. In this sense, it was analyzed the differences between each mechanism regarding to complexity, stability and modularity.

Moreover, statistical analysis was applied to test the hypotheses. The test rejected five null hypotheses, except one related to CDO. These results can offer initial guidance to classify in different priority levels the variability mechanisms.

Table 4.3: Results for the quantitative analysis for Warehoure.

Variability Mechanism	Metrics	Release 1	Release 2	Release 3
Compilation Conditional	CC1	1.609	1.586	1.609
	LOC	2487	2697	2934
	WOCS	8.756	8.744	8.848
	IMSC	0.272	0.2651	0.2585
	CDC	0.088	0.091	0.082
	CDO	0.048	0.045	0.036
AOP	CC1	1.482	1.528	1.511
	LOC	2649	2813	2933
	WOCS	7.62	7.904	7.661
	IMSC	0.27	0.269	0.268
	CDC	0.083	0.086	0.077
	CDO	0.047	0.044	0.036
OSGi	CC1	1.275	1.268	1.229
	LOC	2992	3160	3309
	WOCS	3.892	3.814	3.541
	IMSC	0.342	0.327	0.337
	CDC	0.074	0.076	0.065
	CDO	0.048	0.046	0.039

4.3 THREATS TO VALIDITY

According to Runeson *et al.* (2012), the validity of a study expresses the the trustworthiness of the results. Moreover, it is used to indicate in what extent the results are not biased by the researchers subjective point of view. They also claim that the validity should be addressed during all steps of the case study.

There are some ways to categorize the aspects of validity and threats to validity in the

literature. This work used as basis the scheme proposed by Yin (2009), with the following aspects: construct validity, internal validity, and external validity. Thus, the threats to validity of this work are described next.

- **Construct Validity:** The fact that the main researcher is also involved in the development process is a threat to construct validity for this case study, since he can have influence on the results and conclusions of the work. In order to mitigate this influence, another software engineer was involved in the development process of the Warehouse. Moreover, for each mechanism studied, we developed three releases, so it made possible to compare the results and identify any incoherent result.

- **Internal Validity:** This case study addressed different areas, such as, SOC, mechanisms for implementing variability, and the combination between SOC and SPL known as SOPL. Understanding all of these areas would take a long time and effort. Thus, it is possible some concepts may have been poorly understood. Trying to solve these possible issues, we investigated related work during the since literature review that could provide metrics to be used and how proceed during the evaluation process.

- **External Validity:** (Runeson *et al.*, 2012) claim that this aspect of validity is concerned with to what extent it is possible to generalize the findings and to what extent the findings are of interest to other people outside the investigated case. Thus, as this study aims at identifying the most suitable variability mechanism, it may be used as baseline for comparison in further studies developed in the same context. However, the small domain analyzed is a threat the Warehouse is an academic small project that may not have functional value with software product lines and service-oriented product lines concepts and that makes it harder generalizations. However, in order to solve these possible issues, we will replicate this study in another domain.

4.4 CHAPTER SUMMARY

This Chapter presented a case study describing in details its objective, the case, units of analysis, data collection and analysis and the main findings.

The case study aimed at comparing different variability mechanism and identifying the applicability of them on the development of SPL and SOPL. In order to set up this environment, we evaluated complexity, modularity, and stability.

Next Chapter presents a replicated case study applied in another domain. The goal is to highlight the aspects identified in this study, and try to identify new evidence in the area.

CHAPTER 5

THE REPLICATED CASE STUDY

Replications are useful since they allow understanding how many results were influenced by the context. Moreover, when a replicated case study has the same results as the previous one, it means that generalizations regarding the particular research questions can be made (Runeson *et al.*, 2012).

Juristo and Gómez (2012) present some definitions for replication in the scientific context, including one that says it is a methodological tool based on a repetition procedure that is involved in establishing a fact, truth, or piece of knowledge. Replications also are classified into three groups based on the types analyzed: (i) replications that vary little or not at all with respect to the reference experiment; (ii) replications that do vary but still follow the same method as the reference experiment; (iii) replications that use different methods to verify the reference experiment results (Gómez *et al.*, 2010). Thus, this Chapter presents a replicated case study that has as a basis all the steps defined in the last case study presented in the previous Chapter. It is important to highlight that this study was performed on another domain with new SPLs and SOPL.

The replicated case study presented in this Chapter aims at analyzing the variability mechanisms used for implementing SPL and SOPL and comparing the results with the last case study. The remainder of this Chapter is organized as follows. Section 5.1 presents the replicated research design. The results and findings are described in Section 5.2 and Section 5.3 presents the comparative analysis. Section 5.4 discusses the threats to validity, and finally, Section 5.5 presents the Chapter summary.

5.1 CASE STUDY PROTOCOL

When generalizing results, the protocol used in a replicated case study must be identical to the that applied in the original case. Thus, it is possible to demonstrate quality assurance or to support a replication by other researchers. For this reason, the protocol and the guidelines applied in the previous study were used in this replication to guarantee that this case study will be as similar as possible to the original one.

Thus, the protocol definitions proposed are detailed in the next sub-sections: objective

(Section 5.1.1), the bounded system or case (Section 5.1.2), units of analysis (Section 5.1.3), case study research questions (Section 5.1.4), data collection instruments (Section 5.1.5) and data analysis procedures (Section 5.1.6).

5.1.1 Objective

As well as in the original case study, we replicated the case study aiming at making a contribution to the body of knowledge in the development process of SPL and SOPL. Thus, the overall goal of our replicate case study was to investigate the code quality inside this context, hence, increasing the generalizations of the initial results.

For this study, we selected an more complex SPL (and consequently the SOPL developed with OSGi) that has its domain related to management of academic scientific events. Thus, following the GQM template developed for the original case study, the goal of this replication is:

Goal. *Analyze the* variability implementation techniques (conditional compilation, OSGi, and aspects) *for the purpose of* evaluation *with respect to* reusability, maintainability, complexity, stability and modularity *from the point of view of* software engineers and researchers *in the context of* a service-oriented product line project.

Hypotheses. In a similar way, the hypotheses defined for the previous case study will also be tested in this study. Thus, the null hypotheses which we would like to reject are those that claim that there is no difference among the techniques for implementing variabilities:

$$
\begin{aligned}
H_0 \quad &: \quad CC2_{CC1} = AOP_{CC1} = OSGi_{CC1} \\
&: \quad CC2_{LOC} = AOP_{LOC} = OSGi_{LOC} \\
&: \quad CC2_{WOCS} = AOP_{WOCS} = OSGi_{WOCS} \\
&: \quad CC2_{IMSC} = AOP_{IMSC} = OSGi_{IMSC} \\
&: \quad CC2_{CDC} = AOP_{CDC} = OSGi_{CDC} \\
&: \quad CC2_{CDO} = AOP_{CDO} = OSGi_{CDO} \qquad (5.1)
\end{aligned}
$$

On the other hand, the alternatives hypotheses claim that there are differences among the techniques for implementing variabilities:

$$
\begin{aligned}
H_a \quad & : \quad CC2_{CC1} \neq AOP_{CC1} \text{ or } CC2_{CC1} \neq OSGi_{CC1} \text{ or } AOP_{CC1} \neq OSGi_{CC1} \\
& : \quad CC2_{LOC} \neq AOP_{LOC} \text{ or } CC2_{LOC} \neq OSGi_{LOC} \text{ or } AOP_{LOC} \neq OSGi_{LOC} \\
& : \quad CC2_{WOCS} \neq AOP_{WOCS} \text{ or } CC2_{WOCS} \neq OSGi_{WOCS} \text{ or } AOP_{WOCS} \neq OSGi_{WOCS} \\
& : \quad CC2_{IMSC} \neq AOP_{IMSC} \text{ or } CC2_{IMSC} \neq OSGi_{IMSC} \text{ or } AOP_{IMSC} \neq OSGi_{IMSC} \\
& : \quad CC2_{CDC} \neq AOP_{CDC} \text{ or } CC2_{CDC} \neq OSGi_{CDC} \text{ or } AOP_{CDC} \neq OSGi_{CDC} \\
& : \quad CC2_{CDO} \neq AOP_{CDO} \text{ or } CC2_{CDO} \neq OSGi_{CDO} \text{ or } AOP_{CDO} \neq OSGi_{CDO} \qquad (5.2)
\end{aligned}
$$

5.1.2 The Case

The SPL under study is an SPL for desktop applications that assists users in the management of academic scientific events. The development team for the base version of the SPL used in this study was composed of one Ph.D. student and two M.Sc. students. However, different from the previous case study, the main researcher did not participate in all development phases. He worked on the development of the SPL through AOP and the SOPL using OSGi based on conditional compilation version implemented by other developers. Figure 5.1 shows a screenshot of the most complete product of the RISEEvent SPL developed by RiSE Labs using conditional compilation as variability implementation technique.

Figure 5.1: Screenshot of RiSEEvent SPL product implemented with conditional compilation.

RiSEEvent is composed by 28 features which provide the required functionalities to

enable overall management of the entire academic scientific events, such as, submission, registration and evaluation of the work, payment, event program, besides generating some documentation such as reports, and certificates.

Table 5.1: RiSEEvent features for releases.

	Release 1	Release 2	Release 3
EventProgram	x	x	x
EventImportantDates	x	x	x
Activity	x	x	x
RegistrationUserActivity	x	x	x
ReportsFrequencyperEvent	x	x	x
ReportsFrequencyperActivity	x	x	x
ReportsListofAuthors	x	x	x
Bugs	x		x
Speaker		x	x
Organizer		x	x
RegistrationSpeakerActivity		x	x
RegistrationOrganizerActivity		x	x
CheckingCopyAtestado		x	x
CheckingCopyCertificado		x	x
PaymentAvista		x	x
PaymentDeposito		x	x
PaymentCartao		x	x
SubmissionParcial		x	x
InsertAuthors		x	x
Receipt		x	x
Reviewer			x
ReviewRoundofReview			x
ReviewSimpleReview			x
SubmissionCompleta			x
AssignmentChairindication			x
Assignmentautomatic			x
ConflictofinterestAutomatic			x
Notification			x

Based on the initial version implemented with conditional compilation (by other developers), the main researcher decided to develop two other releases aiming to complete the number of products for this variability mechanism to be evaluated. As the original case study, the Release 1 contains the core assets of the application. Subsequently, changes were incorporated to implement all features designed for this SPL. At the end of this process, three scenarios for each variability technique through inclusion, changing, or

removing classes were implemented. These scenarios are illustrated by Table 5.1 and Figure 5.2 which present the feature model of RiSEEvent SPL.

Figures 5.3 and 5.4 present an overview of the set of scenarios implemented for RiSEEvent SPL. As explained in the previous case study, the columns R1, R2, and R3 represent the different releases of the SPL. In RiSEEvent case study, the number of components required in all releases by CC2 implementation is also smaller than AOP and OSGi (which requires more components). In general, this result is similar when the point of view observed is the number of methods, which had measured an interval ranging from 529 to 1567. Only in the release 1 there was the occurrence of the implementation developed with AOP to have the greatest number of methods among the releases analyzed. The range for that was from 529 to 1567.

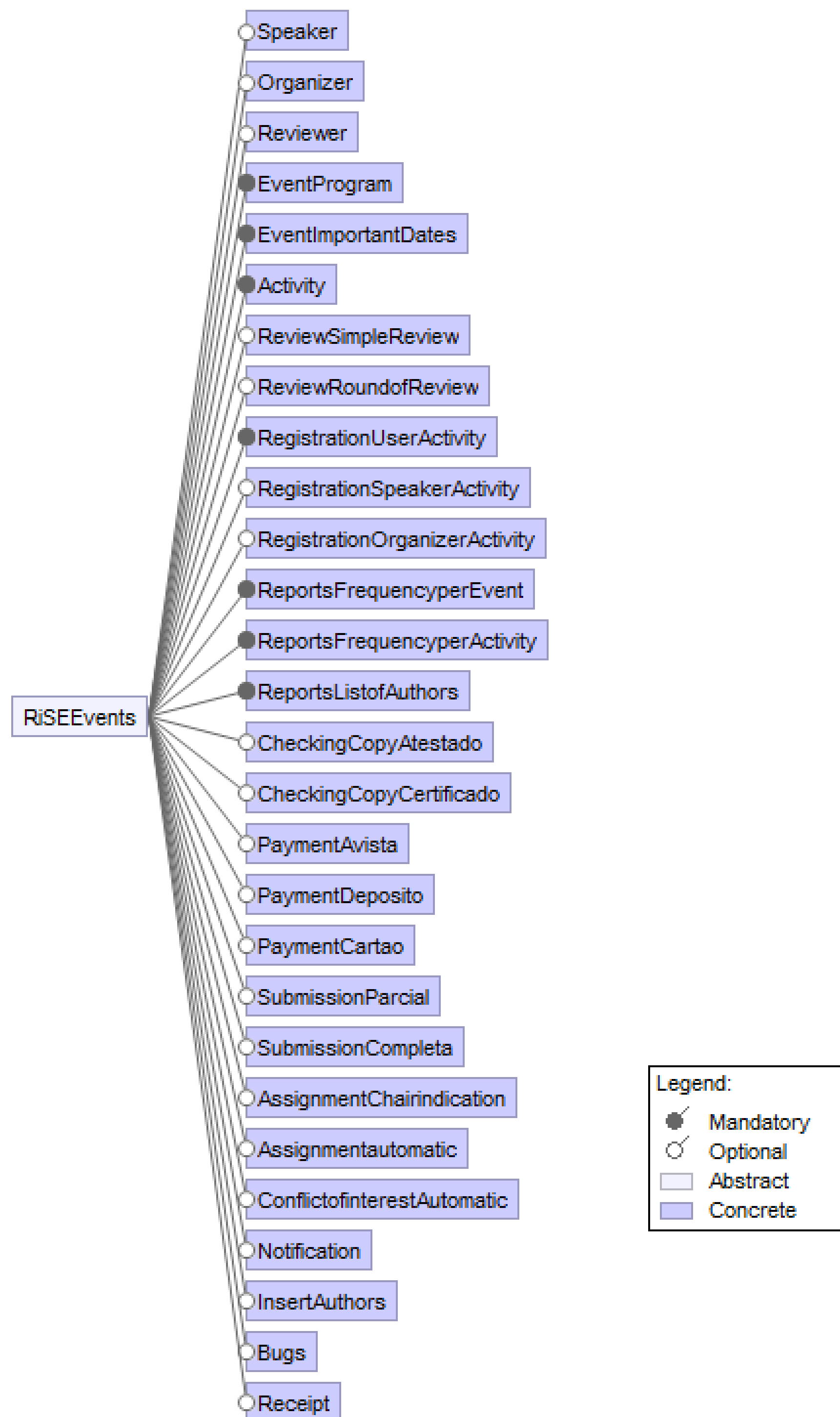


Figure 5.2: RiSEEvents feature model.

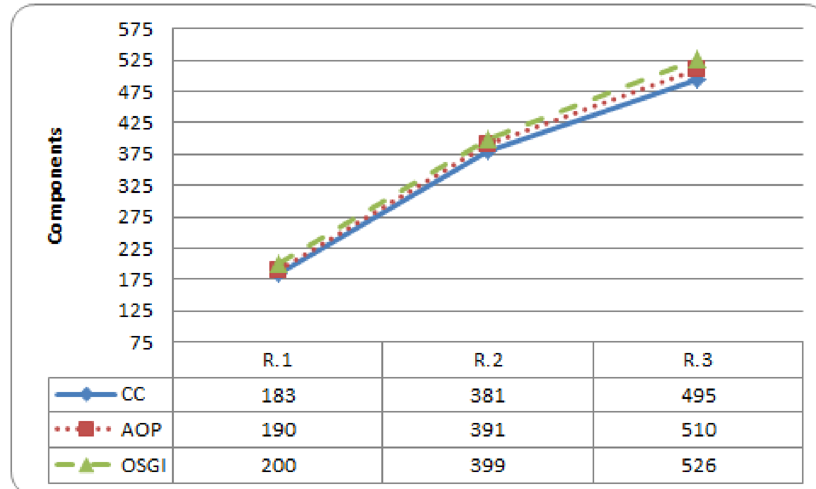


Figure 5.3: RiSEEvent SPL implementation (Components).

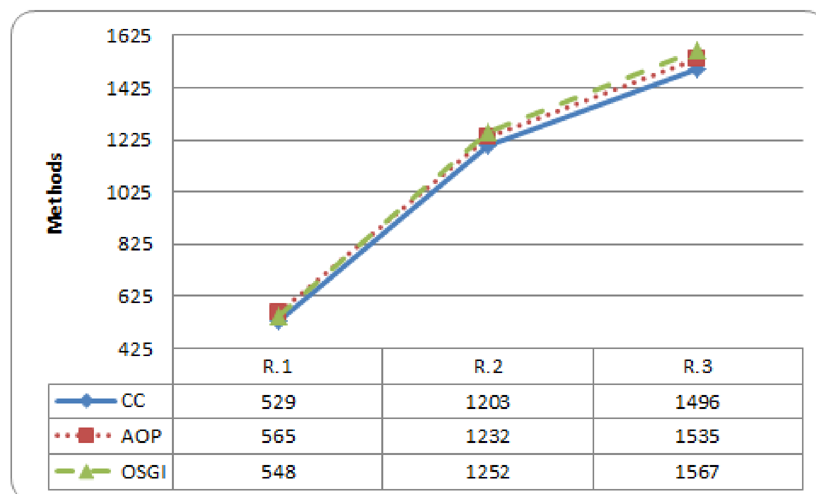


Figure 5.4: RiSEEvent SPL implementation (Methods).

5.1.3 Units of Analysis

For this case study, the unit of analysis is similar to the previous one. First, we collected the data from each SPL implementation in an individual way, and then we performed their comparative analysis.

5.1.4 Case Study Research Questions

The main objective of the original study was to investigate the code quality in the context of SPL and SOPL development applying the framework of the impacts of each mechanism. This replication had the same purpose, but it also aimed to reinforce the results achieved in the first case study and to enable the generalization of conclusions. Thus, we reused the same research questions:

Question 1: How complex are the services and components developed using CC2, AOP or OSGi?

Question 2: How stable are the services and components developed using CC2, AOP or OSGi?

Question 3: How modular are the services and components developed using CC2, AOP or OSGi?

More detail can be seen in GQM template of the previous study Subsection 4.1.

5.1.5 Data Collection

The case studies have many similarities. Both analyzed only the source code of SPLs and SOPL. In this way, the replicated case also used the method of the third degree, of which there is only the study of work artifacts (Lethbridge *et al.*, 2005), in the data collect aimed to perform static analysis.

5.1.6 Data Analysis

The strategy used in this case study was composed by a quantitative analysis of the data obtained after the metrics application (Section 3.3) in the source code of RiSEEvent SPL and SOPL. Next sections will present the analysis of complexity, modularity, and stability.

5.2 RESULTS AND FINDINGS

This section discusses the findings of the protocol proposed in the previous case study when applied on the second SPL.

5.2.1 Complexity Analysis

This section presents the results obtained for the complexity metrics (cyclomatic complexity, line of code, and weighted operations per component or Service). In Figure 5.5, it is possible to observe the cyclomatic complexity collected from the implemented scenarios.

The releases implemented through conditional compilation had a more elevated cyclomatic complexity among the studied techniques. Another aspect identified is the difference of values measured by each technique. CC2 and AOP values are closer to each other than the OSGi mean. The first two had mean values measured at around 2.1 while OSGi presented values to releases 1 and 2 of about 1.8 and to release 3 of 1.745

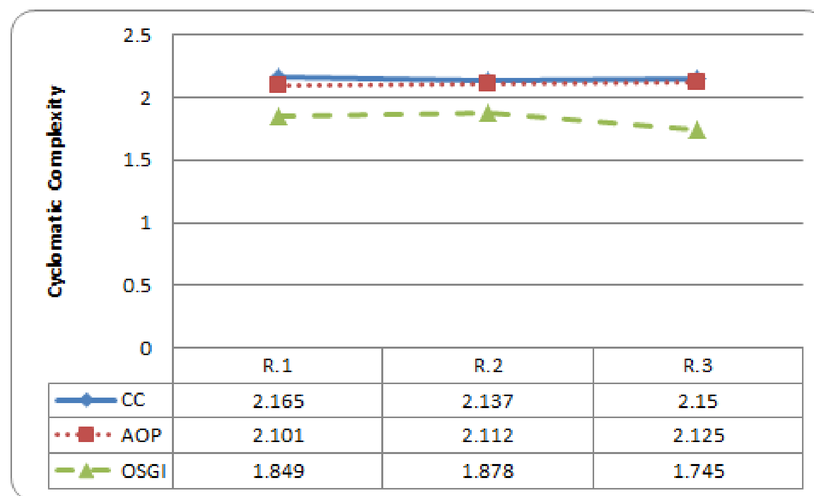
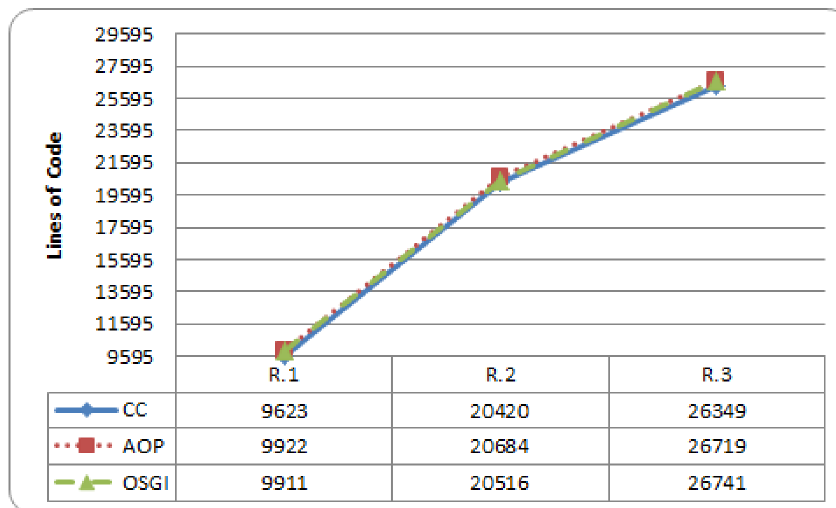


Figure 5.5: RiSEEvent Cyclomatic Complexity.

An opposite situation occurs when the observed criterion is lines of code. In this context, AOP becomes the mechanism with the highest values measured and conditional compilation required the lowest amount of lines of code. The exception happened in release 3, which OSGi was the technique that used more lines of code. This scenario can be seen in Figure 5.6.

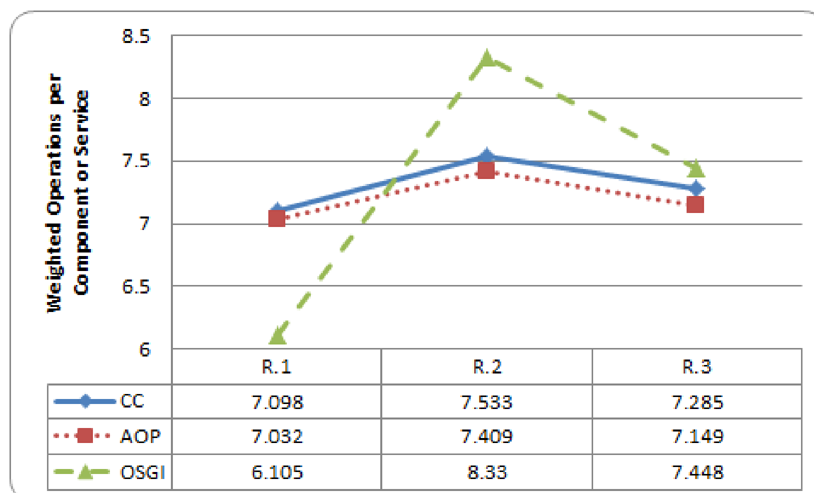
Finally, the last complexity metric used was weighted operations per components or service. It had a peculiarity, a wide variation in the results for OSGi. In the first release,



	R.1	R.2	R.3
CC	9623	20420	26349
AOP	9922	20684	26719
OSGI	9911	20516	26741

Figure 5.6: RiSEEvent Lines of Code.

OSGi had (with a large difference from other mechanisms) the smallest measured value. However, in release 2, the situation changed and it made the technique more complex. In release 3, this situation was repeated, but with the mean closer to other mechanisms. Except the first, AOP was the technique with the best results. Figure 5.7 shows these results.



	R.1	R.2	R.3
CC	7.098	7.533	7.285
AOP	7.032	7.409	7.149
OSGI	6.105	8.33	7.448

Figure 5.7: RiSEEvent Weighted Operations.

5.2.2 Stability Analysis

In Figure 5.8 it can be observed that the components built with OSGi are the most unstable among the ones studied. The other mechanisms had similar levels of instability, with a slight tendency for the implementations that use conditional compilation to be more stable.

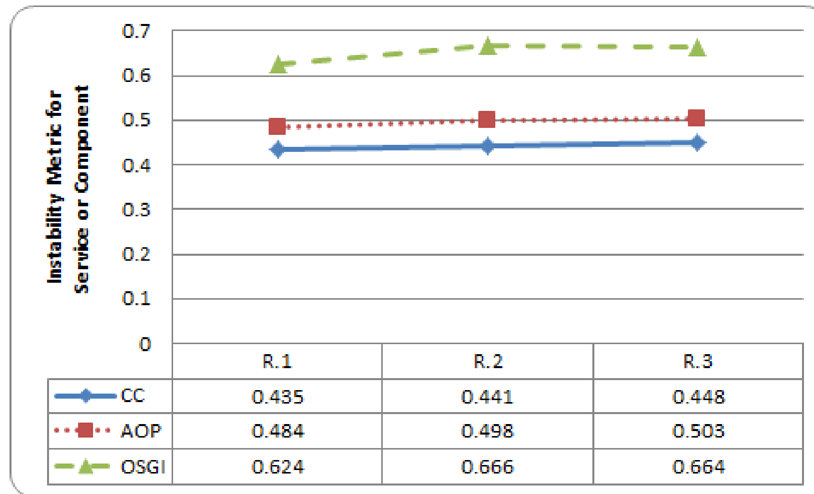


Figure 5.8: RiSEEvent Instability.

5.2.3 Modularity Analysis

The last group of metrics used in these studies is related to the modularity of programs. Figure 5.9 shows the results for CDC. Based on the results, we can state that features implemented with OSGi are a little less diffuse among the components than the others developed with AOP and conditional compilation. These had very similar results, which were not expected, since it did not occur in the first case study.

Figure 5.10 displays the results achieved with the concern diffusion metrics over operations. The collected data are again inconclusive regarding which mechanism had the best performance. OSGi presented a large variation and the others had identical results in both releases. It is possible that during migration from conditional compilation to AOP some external factor had influenced this process.

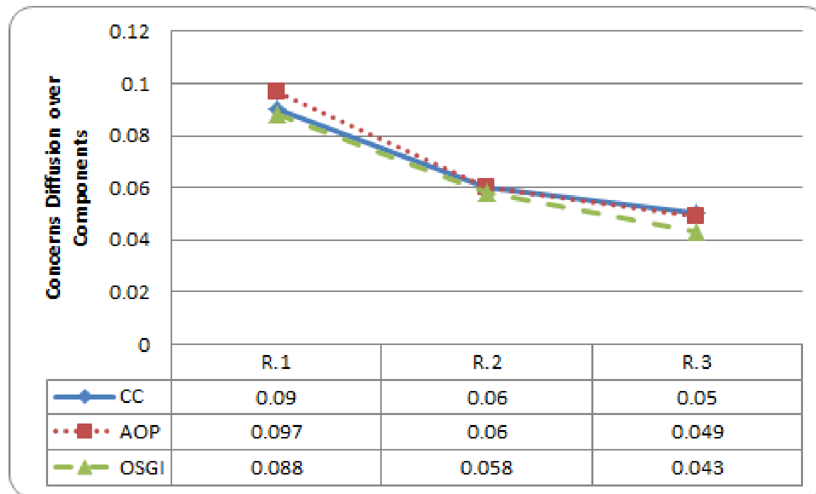


Figure 5.9: Concerns Diffusion over Components for RiSEEvent.

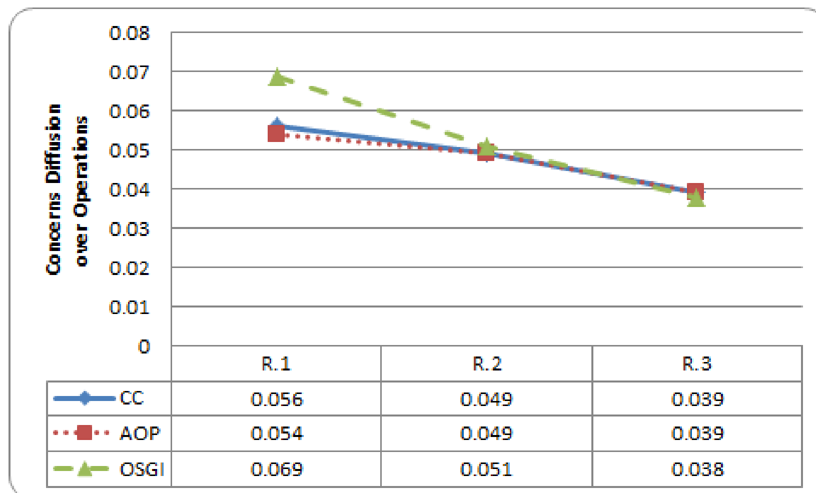


Figure 5.10: Concerns Diffusion over Operations for RiSEEvent

5.2.4 Descriptive and Exploratory Analysis

As in the first case study, we used the nonparametric Kruskal-Wallis test to compare the different mechanisms of variability and the results are presented in Table 5.2. They indicate that (for RiSEEvent) only the results from Cyclomatic Complexity and Instability which presented significant statistic difference at the level of 5% in comparison among the variability mechanisms should be considered. As a result, most mechanisms presented similar behavior in the Boxplot of Figure 5.11.

With these results, only the two null hypotheses related to cyclomatic complexity and instability defined for this case study can be rejected. Consequently, the other metrics did not produce significant results in this study.

Table 5.2: Results for the quantitative analysis for RiSEEvent.

Variable	Kruskal-Wallis Chi-squared	H-value	df
Cyclomatic Complexity	7.2000	5.6	2
Weighted Operations	0.6222	5.6	2
Lines of Code	0.6222	5.6	2
Instability	72.000	5.6	2
Concerns Diffusion over Components	0.6050	5.6	2
Concerns Diffusion over Operations	0.0904	5.6	2

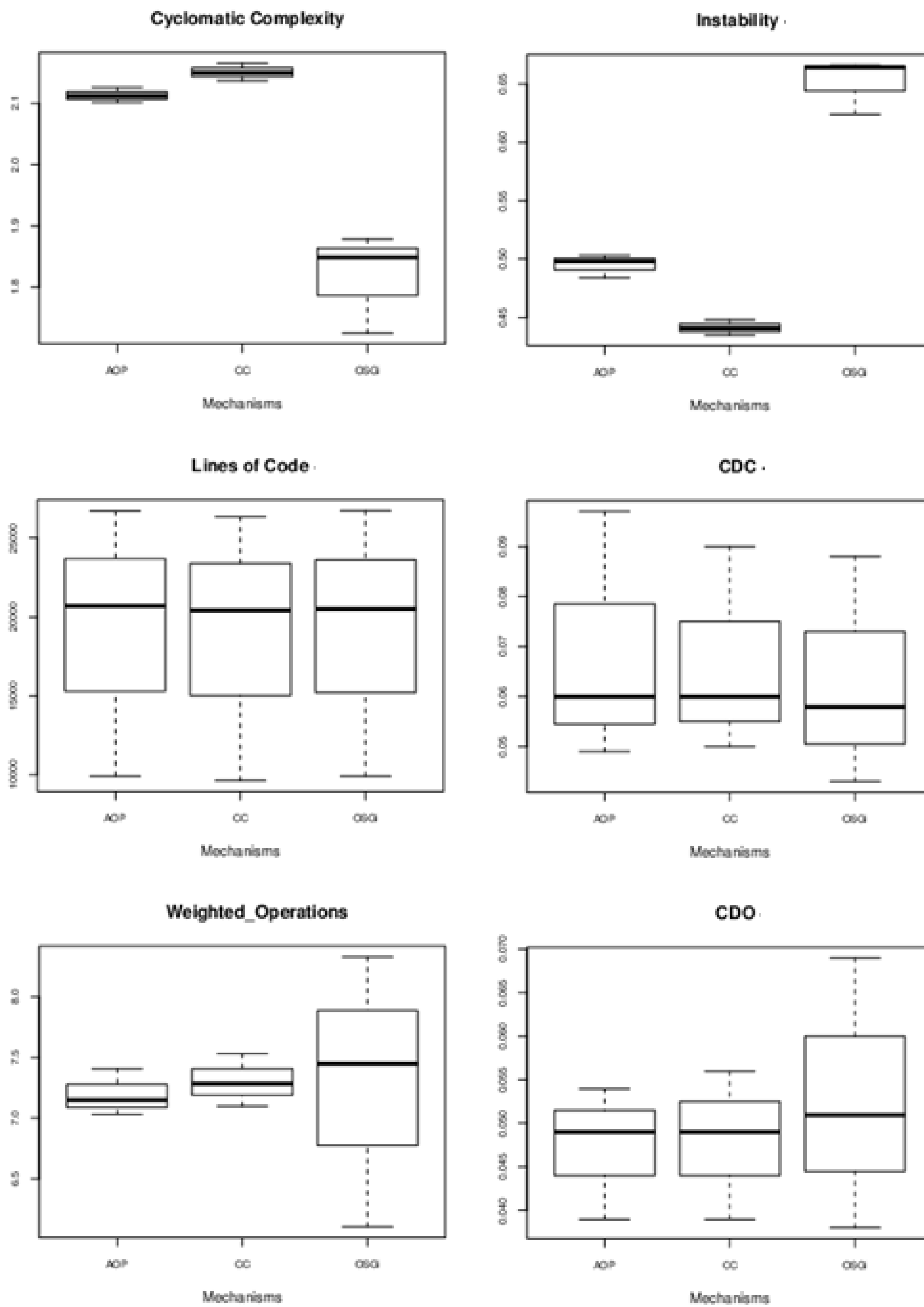


Figure 5.11: Boxplot comparing the variability mechanisms according to aspects complexity, stability and modularity.

5.2.5 Results Summary

Table 5.3 summarizes the results found in this replicated case study where were analyzed the differences between each mechanism related to complexity, stability and modularity. As in the first case study, statistical analyses were applied to test the hypotheses. In this case, were rejected two null hypotheses, indicating that only the alternative hypotheses related to cyclomatic complexity and instability may be true. In this sense, these two results can be combined with the results achieved in the previous case study to classify into different priority levels the variability mechanisms during the decision model development. Same as in the previous study, it is necessary to perform more detailed studies for verifying the hypotheses that were not rejected.

Table 5.3: Results of the quantitative analysis for RiSEEvent.

Variability Mechanism	Metrics	Release 1	Release 2	Release 3
Compilation Conditional	CC1	2.165	2.137	2.15
	LOC	9623	20420	26349
	WOCS	7.098	7.533	7.285
	IMSC	0.435	0.441	0.448
	CDC	0.09	0.06	0.05
	CDO	0.056	0.049	0.039
AOP	CC1	2.101	2.112	2.125
	LOC	9922	20684	26719
	WOCS	7.032	7.409	7.149
	IMSC	0.484	0.498	0.503
	CDC	0.097	0.06	0.049
	CDO	0.054	0.049	0.039
OSGi	CC1	1.849	1.878	1.745
	LOC	9911	20516	26741
	WOCS	6.105	8.33	7.448
	IMSC	0.624	0.666	0.664
	CDC	0.088	0.058	0.043
	CDO	0.069	0.051	0.038

5.3 COMPARATIVE ANALYSIS

In this section, we performed the comparison among the results obtained in both domains.

General Comparison. The SPL was implemented in JAVA language using the Model-View-Controller (MVC) as the architectural pattern in the desktop domain. The

conditional compilation (CC2), Aspect-Oriented Programming (AOP), and Open Services Gateway initiative (OSGi) were the techniques used to implement 23 features or services and perform the variability among the products. These different products have in their full releases 46, 56, and 72 Java classes for each technique used respectively.

As Warehouse, RiSEEvents had three main products implemented using conditional compilation, aspect-oriented programming, and Open services gateway initiative with 495, 510, and 526 Java classes for each technique used. Table 5.4 compares the general information for both systems.

Table 5.4: Comparative table with the Warehouse and RiSEEvent SPLs.

SPL	Techniques	Number of Classes	LOC	Number of Features
Warehouse	CC2	46	2934	23
	AOP	50	2933	
	OSGi	62	3309	
RiSEEvent	CC2	495	26349	28
	AOP	510	26719	
	OSGi	526	26741	

Complexity Comparison. The individual analysis allows us to observe that both case studies had similar data related to the first complexity criterion (cyclomatic complexity). OSGi was the technique with the best results in all products followed by AOP with intermediary results. However, in the RiSEEvent the values for releases implemented with conditional compilation were very close to AOP releases.

For the second criterion (lines of code), the results from both studies converged again, but in RiSEEvent the measured values were closer. They presented conditional compilation as the technique that needs fewer lines of code for implementing the SPLs. OSGi had more lines of code in its implementation for Warehouse, however, AOP had more lines of code in its implementation for two products of RiSEEvent (releases 1 and 2).

Finally, considering weighted operations per components or service there was an unusual fluctuation of results obtained. OSGi had WOCS values in Warehouse and in the first release of RiSEEvent smaller than other mechanisms had. However, for the second release from RiSEEvent, the measured results had a considerable increase, which made OSGi to have the worst results among the mechanisms in releases 2 and 3 of RiSEEvent. These values indicate that the services responsible for implementation of features are composed of possible god-classes, it means, classes that control too many other objects in the system and so it becomes a class that does everything. The same variation did not

occur in Warehouse results, where OSGi had best results followed by AOP and CC.

Stability Comparison. Both case studies had the components developed with conditional compilation more stable that the other ones. This result can be observed more clearly in the replicated case that had the greatest difference between the data from CC2 and AOP. These statements can be verified in Figures 4.11 and 5.11.

Modularity Comparison. The results obtained indicate that OSGi had the best components regarding to CDC metric . This can be observed only in the first case study where the mean of OSGi had, in general, 0.01 of difference from another mechanism. In the replicated case study, the results did not allow to infer any meaning to the measured values.

Concerning CDO, we could not identify significant differences among the three approaches in both systems. Thus, based on the results presented in Tables 4.3 and 5.3 we could not discard the null hypotheses with the use of concerns diffusion over operations metric.

Table 5.5: Descriptive Statistics by Program.

Variable	P.	Min	1Â° Qu.	Median	3Â° Qu.	Max	Mean	SD	CV	SV
CC1	P1	1.229	1.275	1.511	1.586	1.609	1.454	0.154	10.60	-0.42
	P2	1.725	1.878	2.112	2.137	2.165	2.027	0.163	8.06	-0.73
LOC	P1	2487	2697	2933	2992	3309	2886	256	8.89	0.09
	P2	9625	9922	20516	26349	26741	18987	7362	38.77	-0.27
WOCS	P1	3.541	3.892	7.661	8.744	8.848	6.753	2.302	34.09	-0.49
	P2	1.725	1.878	2.112	2.137	2.165	2.027	0.163	8.06	-0.73
IMSC	P1	0.259	0.268	0.270	0.327	0.342	0.290	0.035	11.92	0.59
	P2	0.435	0.448	0.498	0.624	0.666	0.529	0.095	18.03	0.47
CDC	P1	0.065	0.076	0.082	0.086	0.091	0.080	0.008	10.05	-0.41
	P2	0.043	0.050	0.060	0.088	0.097	0.066	0.020	30.40	0.43
CDO	P1	0.036	0.039	0.045	0.047	0.048	0.043	0.005	11.38	-0.49
	P2	0.038	0.039	0.049	0.054	0.069	0.049	0.010	20.24	0.48

Figure 5.12 shows the boxplot comparing the results of the variables for each program. It is observed, for example, that the minimum values of cyclomatic complexity, lines of code and instability for the RiSEEvents program (P2) are higher than the maximum values for the Warehouse program (P1). Table 5.5 presents the results of descriptive statistics (minimum, 1^0 quartile, median, 3^0 quartile, mean, standard deviation, coefficient of variation and skewness coefficient) from each variable for the programs. In this sense, the variables with higher dispersion values (considered moderate in the context of statistics) were lines of code (CV = 38.77%) and CDO (CV = 30.40%) for RiSEEvent and weighted

operations (CV = 34.09%) for Warehouse.

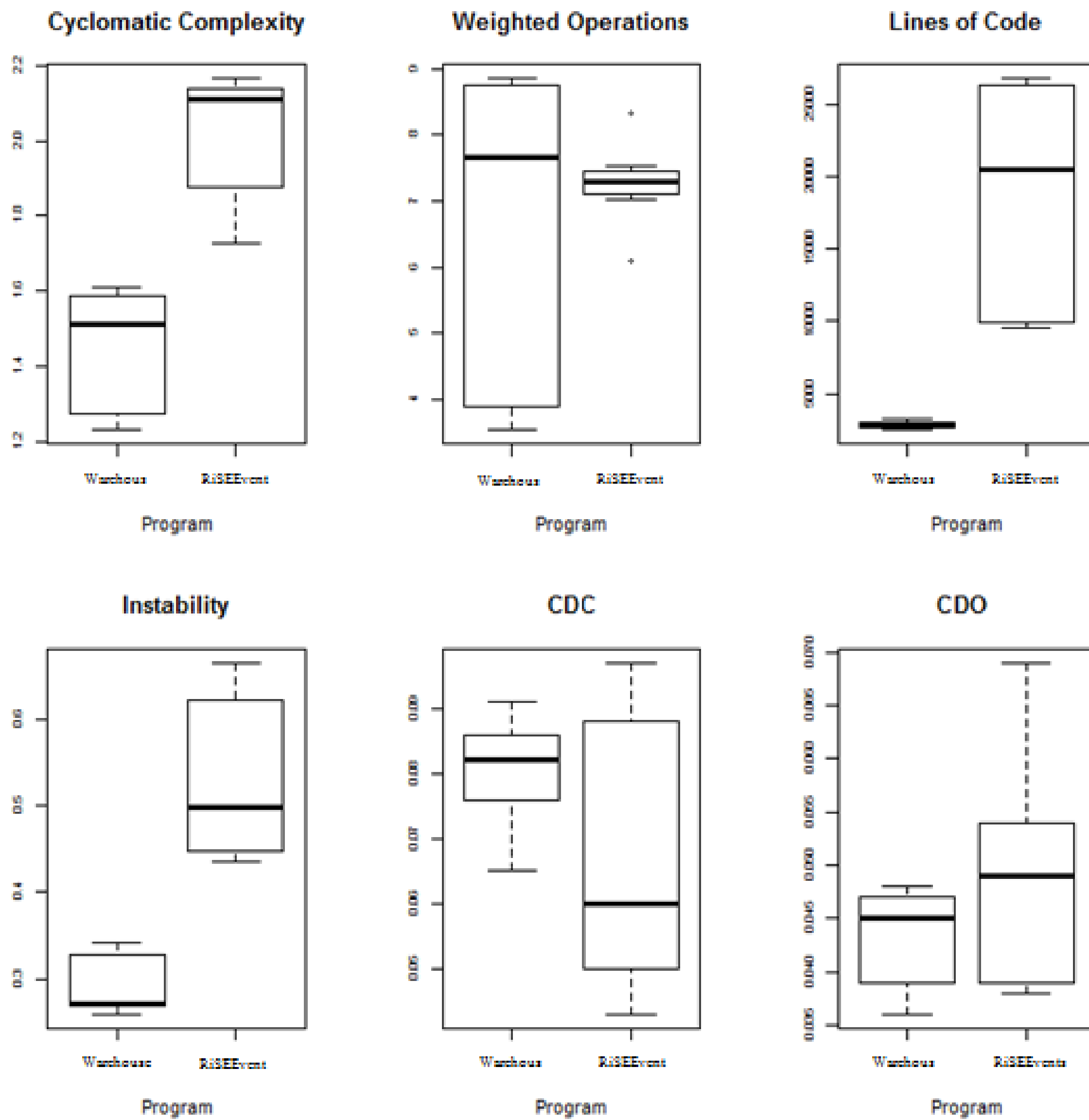


Figure 5.12: Boxplot for the variables by program.

5.3.1 Decision Model

The descriptive and exploratory analysis indicated that we should consider some factors when using the techniques to implement variability. Each criterion considered in this study (complexity, stability, and modularity) was detailed previously. In this sense, we verified some differences among the techniques studied which allowed us to define an initial

Figure 5.13: Decision model process.

decision model. It was influenced by the work of Ribeiro (2010) where a decision model for recommending variability mechanisms for three criteria was built. Moreover, the decision model was based on the quantitative and statistical analyses previously presented.

In the development process, we considered only the data that rejected the null hypotheses. Because of this, there is no indication for the most suitable techniques when software modularity is assessed, for example, by CDO metric in the decision model. Moreover, as a result of the lack of significative differences among the techniques two mechanisms can be as indicated as first options for a particular metric observed.

The decision model is centered on variability mechanisms and has three inputs that produce the most suitable indication for the observed criteria. The first input is the variability mechanisms target of the study (conditional compilation, AOP, and OSGi); the second one consists in the criteria chosen to compare the techniques (complexity, stability, and modularity); and the last one are the metrics used to quantify the criteria observed in the case studies.

Figure 5.13 shows this process that produces as output the recommended variability mechanism for the criterion chosen. It is important to highlight that the decision model makes a suggestion, but the user has the final decision.

Thus, based on the achieved results in first case studies, the decision model showed in Figure 5.14 was elaborated. In this model, there are priorities (for each metric) that indicate the most suitable mechanism for implementing variability in the context analyzed. For example, according to the decision model, OSGi is the technique that produces the best components considering cyclomatic complexity.

Figure 5.14: Warehouse Decision model.

Similarly, we developed other decision model based on the replicated case study results. In this case, only the results from cyclomatic complexity and instability which presented significant statistic difference at the level of 5%. Figure 5.15 shows RiSEEvent decision model.

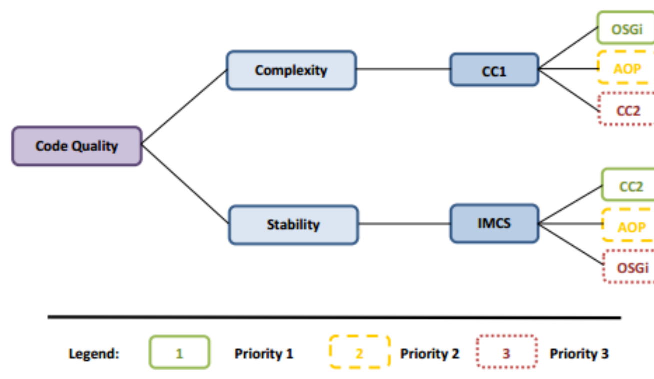


Figure 5.15: RiSEEvent Decision model.

Moreover, comparing the results we can assume this one as the model produced with the results of both studies. Since the cyclomatic complexity and stability results were evaluated in the original case study and reinforced with new assessments in the

replicated case study. Thus, this second assessment allowed understanding what results were influenced by the context, and provided support for making these generalizations regarding the cyclomatic complexity and instability.

5.4 THREATS TO VALIDITY

As the original case study, this replicated study also presented threats to validity that are described as follows:

- **Construct Validity:** This case study also had the main researcher as the developer of the source code. However, different from the first study, there were more software engineers involved in the development process. Moreover, the main researcher participated in a more advanced stage, only in the migration from conditional compilation to other mechanisms. Thus, this study can have an influence on the results and conclusions of the work performed by the researcher. In the same way as the first case study and aiming to mitigate this influence, three releases were developed for each technique.

- **Internal Validity:** The wide knowledge areas addressed in this study together with the context of RiSEEvent SPL (academic product line) are also threats for the study as happened in the first case study. In order to mitigate these issues, a compilation of the results through a comparison with data from both studies was performed.

5.5 CHAPTER SUMMARY

This Chapter presented a replicated case study based on the same protocol of the study discussed in the previous Chapter. We aimed to extend the original study replicating it in a different SPL context following the same original design. Thus, it was applied the same measurement framework in order to identify the level of complexity, modularity, and stability of RiSEEvent SPL.

It was also detailed the comparison of results obtained with Warehouse SPL and RiSEEvent SPL. Moreover, based on these results an initial decision model for recommending the most suitable variability mechanisms for each criterion evaluated in the studies was created.

Next Chapter presents the research contributions, future work and concluding remarks of this dissertation.

CHAPTER 6

CONCLUSIONS

Software Product Lines can be used in different contexts. Nowadays, the implementation of SPL with software assets developed as services has received the attention of researchers due to its significant potential. In this context, this dissertation investigated the combination of SPL and SOPL with emphasis on variability implementation, more specifically, the choice of a specific variability mechanism according to a particular context. These concepts achieve desired benefits such as improving reuse, decreasing development costs and time-to-market and making flexible applications which fulfill the requirements of users or market segment needs.

Initially, a brief literature review identified proposals of how variability implementation is being addressed by researchers in SPL and SOPL context. Moreover, this activity was also responsible for searching metrics and a guidelines for performing the evaluation of the techniques used in SPL and SOPL implementations.

Regarding evaluation, three variability techniques were considered (conditional compilation, aspect-oriented programming, and Open services gateway initiative), for evaluation based on a measurement framework used as the basis to perform the assessment. This framework was composed of metrics related to desired software qualities, which ensure reusability and maintainability of the source code.

A case study was conducted in the warehouse context with the goal of comparing the variability implementation techniques. The analysis provided data concerning to a set of metrics. Next, a replicated case study was also performed in a different domain. This replicated study was applied on a product line of academic events management and its main goal was to obtain more evidence allowing a comparison among the studies.

Based on the quantitative and statistical analyses of the case studies, we defined an initial decision model responsible for guiding software engineers during the choice of techniques to implement variability in SPL and SOPL projects. This decision model is able to indicate the suitable mechanism according to provided inputs, such as the criteria chosen to compare the techniques (complexity, stability, and modularity) and metrics used as the basis for comparison.

6.1 MAIN CONTRIBUTIONS

The main contributions of this work can be summarized into two aspects *(i) an assessment for comparing techniques used to implement variability in software product lines and service-oriented product lines; (ii) two case studies in which were performed the evaluations of variability techniques.* These contributions are described as follows:

An assessment for comparing techniques used to implement variability in software product lines and service-oriented product lines. Through this assessment, three variability techniques (conditional compilation, aspect-oriented programming, and Open services gateway initiative) were evaluated with respect to complexity, stability, and modularity. Besides, the evaluation results were used to produce an initial decision model for aiding software engineers on the task of choosing variability technique based on a set of parameters.

Two case studies in which were performed the evaluations of variability techniques. They were applied in an academic context in order to perform the assessments proposed in this work. The original case study allowed comparing the variability mechanism and identifying their applicability on the development of SPL and SOPL. In turn, the replicated case study aimed to highlight the aspects identified in the previous case study by performing the same study in a new SPL.

6.2 CONCLUDING REMARKS

Software Product Lines and Service-Oriented Computing encourage reusing the existing software assets and capabilities rather than developing them from scratch. In order to perform reusing, it is necessary to use techniques that support the variability management activity.

Unfortunately, it is still not well defined among the variability techniques, which one is most suitable for a particular case. Thus, the goal of this dissertation was to investigate the variability implementation in software assets of software product lines and service-oriented product lines.

Our findings from this work come from the evaluation of SPLs and SOPL in two case studies through the application of a measurement framework. They helped to construct a decision model that indicates the most suitable technique according to a set of criteria observed.

6.3 FUTURE WORK

Due to the time constraints imposed on a M.Sc. degree, this work can be seen as an initial step towards an efficient and effective development of core assets in software product lines and service-oriented product lines. In this way, more research needs to be conducted in order to improve what was initiated, and new paths should be explored. Thus, the following issues should be investigated as future work:

- **Apply this evaluation in other scenarios.** We believe that others case studies in different domains should be performed in order to collect more evidence to support generalizations. It is important that the new domain for this new study comes from an industrial project. Thus, we can perform a cross analysis among the industrial and academic SPLs.

- **Improve the measurement framework.** The metrics used to compose this framework were developed in another context, such as traditional software development. Thus, it is required that they can be refined for SPL and SOPL context. Moreover, new criteria and metrics can be added, as lack of cohesion of methods, in order to increase the coverage of the framework.

- **Evaluate other variability mechanisms.** OSGi was the unique technique really related to implement services analyzed in this dissertation. Hence, new techniques should be investigated and used, for example Java Web Services.

- **Combination of techniques.** Ribeiro *et al.* (2011) performed techniques combination aiming to provide variability to the services. Much in the same way, we can investigate the applicability of this approach to our study context.

- **Decision Model.** The decision model presented in this dissertation is an initial step to take some decisions through a model representation. Thus, it is important to use new variability mechanisms, criteria, and domains in order to enlarge and refine the decision model.

REFERENCES

Almeida, E., Santos, E., Alvaro, A., Garcia, V. C., Meira, S., Lucredio, D., and de Fortes, R. (2008). Domain implementation in software product lines using osgi. In *Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on*, pages 72–81.

Apel, S., Kaestner, C., and Lengauer, C. (2008). Research challenges in the tension between features and services. In *Proceedings of the 2Nd International Workshop on Systems Development in SOA Environments*, SDSOA '08, pages 53–58, New York, NY, USA. ACM.

Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley.

Carvalho, M. L. L., da Silva Gomes, G. S., da Silva, M. L. G., do Carmo Machado, I., and de Almeida, E. S. (2016). On the implementation of dynamic software product lines: A preliminary study. *IEEE Software*.

Chang, S. H. and Kim, S. D. (2007). A variability modeling method for adaptable services in service-oriented computing. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 261–268.

Clements, P. C. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley.

Cohen, S. and Krut, R. (2007). In *Proceedings of the First Workshop on Service- Oriented Architectures and Software Product Lines, 11th International Software Product Line Conference*.

de Souza, L. O., O'Leary, P., de Almeida, E. S., and de Lemos Meira, S. R. (2015). Product derivation in practice. *Information and Software Technology*, **58**, 319 – 337.

Deelstra, S., Sinnema, M., and Bosch, J. (2004). Experiences in software product families: Problems and issues during product derivation. In *In SPLC*, pages 165 – 182. Springer.

Dhungana, D., Seichter, D., Botterweck, G., Rabiser, R., Grünbacher, P., Benavides, D., and Galindo, J. A. (2013). Integrating heterogeneous variability modeling approaches with invar. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 8:1–8:5, New York, NY, USA. ACM.

El-Sharkawy, S., Kröher, C., and Schmid, K. (2011). Supporting heterogeneous compositional multi software product lines. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 25:1–25:4, New York, NY, USA. ACM.

Erl, T. (2007). *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Ferreira, G. C. S., Gaia, F. N., Figueiredo, E., and de Almeida Maia, M. (2014). On the use of feature-oriented programming for evolving software product lines - a comparative study. *Science of Computer Programming*, **93, Part A**, 65 – 85. Special Issue with Selected Papers from the Brazilian Symposium on Programming Languages (SBLP 2011).

Gacek, C. and Anastasopoules, M. (2001). Implementing product line variabilities. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, SSR '01, pages 109–117, New York, NY, USA. ACM.

Gaia, F. N., Ferreira, G. C. S., Figueiredo, E., and de Almeida Maia, M. (2014). A quantitative and qualitative assessment of aspectual feature modules for evolving software product lines. *Science of Computer Programming*, **96, Part 2**, 230 – 253. Selected and extended papers of the Brazilian Symposium on Programming Languages 2012 (SBLP 2012).

Gamez, N., El Haddad, J., and Fuentes, L. (2015). Managing the variability in the transactional services selection. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '15, pages 88:88–88:95, New York, NY, USA. ACM.

Gomaa, H. (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Gómez, O. S., Juristo, N., and Vegas, S. (2010). Replications types in experimental disciplines. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 3:1–3:10, New York, NY, USA. ACM.

Huhns, M. and Singh, M. (2005). Service-oriented computing: key concepts and principles. *Internet Computing, IEEE*, **9**(1), 75–81.

Juristo, N. and Gómez, O. S. (2012). Empirical software engineering and verification. chapter Replication of Software Engineering Experiments, pages 60–88. Springer-Verlag, Berlin, Heidelberg.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute.

Kitchenham, B., Pickard, L., and Pfleeger, S. L. (1995). Case studies for method and tool evaluation. *Software, IEEE*.

Klewerton, W. and AssunÃ§Ã£o, G. (2015). Search-based migration of model variants to software product line architectures. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 895–898, Piscataway, NJ, USA. IEEE Press.

Kotonya, G., Lee, J., and Robinson, D. (2009). A consumer-centred approach for service-oriented product line development. In *Software Architecture, 2009 European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on*, pages 211–220.

Krafzig, D., Banke, K., and Slama, D. (2004). *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Krueger, C. W. (2007). BigLever Software Gears and the 3-tiered SPL Methodology. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 844 – 845, New York, NY, USA. ACM.

Lanman, J., Darbin, R., Rivera, J., Clements, P., and Krueger, C. (2013). The challenges of applying service orientation to the u.s. army's live training software product line. In

Proceedings of the 17th International Software Product Line Conference, SPLC '13, pages 244–253.

Lee, J., Muthig, D., and Naab, M. (2008). An approach for developing service oriented product lines. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 275–284.

Lethbridge, T. C., Sim, S. E., and Singer, J. (2005). Studying software engineers: Data collection techniques for software field studies. *Empirical Softw. Engg.*, **10**(3), 311–341.

Linden, F. J. v. d., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Lisboa, L. B., Garcia, V. C., de Almeida, E. S., and de Lemos, S. R. M. (2011). Toolday: A tool for domain analysis. *Int. J. Softw. Tools Technol. Transf.*, **13**(4).

Matos Jr., P. O. A. (2008). *Analysis of Techniques For Implementing Software Product Lines Variabilities*. Master's thesis, Recife.

McCabe, T. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*.

Medeiros, F., de Almeida, E., and Meira, S. (2010). Sople-de: An approach to design service-oriented product line architectures. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 456–460. Springer Berlin Heidelberg.

Medeiros, F. M., de Almeida, E. S., and Meira, S. R. L. (2009). Towards an approach for service-oriented product line architectures. In *Software Product Line Conference, 2008. SPLC '08. 12th International*.

Mohabbati, B., Asadi, M., Gasevic, D., and Lee, J. (2014). Software product line engineering to develop variant-rich web services. In *Web Services Foundations*, pages 535–562.

O'Leary, P., Richardson, I., McCaffery, F., and Thiel, S. (2009). Preparing for Product Derivation - Activities and Issues. In *ICSOFT 2009 - Proceedings of the 4th International Conference on Software and Data Technologies, Volume 1, Sofia, Bulgaria, July 26-29, 2009*, pages 121 – 126.

Papazoglou, M. (2003). Service-oriented computing: concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3 – 12.

Papazoglou, M., Traverso, P., Dustdar, S., and Leymann, F. (2007). Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, **40**(11), 38 – 45.

Parnas, D. (1976). On the design and development of program families. *Software Engineering, IEEE Transactions on*, **SE-2**(1), 1 – 9.

Pohl, K., BÃ¶ckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Rabiser, R., Grunbacher, P., and Dhungana, D. (2010). Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Inf. Softw. Technol.*, pages 324 – 346.

Rabiser, R., O'Leary, P., and Richardson, I. (2011). Key activities for product derivation in software product lines. *J. Syst. Softw.*, **84**, 285 – 300.

Ribeiro, H. B. G. (2010). *An Approach to Implement Core Assets in Service-Oriented Product Lines*. Master's thesis, Federal University of Pernambuco, Recife.

Ribeiro, H. B. G., de Almeida, E. S., and de Lemos Meira, S. R. (2011). An approach for implementing core assets in service-oriented product lines. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 17:1 – 17:4, New York, NY, USA. ACM.

Runeson, P., HÃ¶st, M., Rainer, A., and Regnell, B. (2012). *Case Study Research in Software Engineering - Guidelines and Examples*. John Wiley & Sons, Inc.

Sant'anna, C., Garcia, A., Chavez, C., Lucena, C., and v. von Staa, A. (2003). On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*.

Schmid, K. and de Almeida, E. S. (2013). Product line engineering. *IEEE Software*, **30**(4), 24–30.

Schulze, M., Mauersberger, J., and Beuche, D. (2013). Functional Safety and Variability: Can It Be Brought Together? In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 236 – 243, New York, NY, USA. ACM.

Sinnema, M., Deelstra, S., and Hoekstra, P. (2006). The covamof derivation process. In *Proceedings of the 9th International Conference on Reuse of Off-the-Shelf Components*, ICSR'06, pages 101 – 114, Berlin, Heidelberg. Springer-Verlag.

Smith, D. and Lewis, G. (2009). Service-oriented architecture (soa) and software product lines: Pre-implementation decisions. In *Proceedings of the 13th International SoftwareProduct Line Conference (SPLC'09) - 3rd International Workshop on Service OrientedArchitectures and Product Lines (SOAPL'09)*, San Francisco, California, USA.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014). Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, **79**, 70 – 85.

Vale, T., Figueiredo, G. B., de Almeida, E. S., and de Lemos Meira, S. R. (2012). A Study on Service Identification Methods for Software Product Lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 156 – 163, New York, NY, USA. ACM.

Yin, R. (2009). *Case Study Research: Design and Methods*. Applied Social Research Methods. SAGE Publications.

Zhang, B., Duszynski, S., and Becker, M. (2016). Variability mechanisms and lessons learned in practice. In *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design*, VACE '16, pages 14–20, New York, NY, USA. ACM.