“An Approach to Implement Core Assets in Service-Oriented Product Lines”

By

Heberth Braga Gonçalves Ribeiro

M.Sc. Dissertation

RECIFE, AUGUST/2010
Heberth Braga Gonçalves Ribeiro

“An Approach to Implement Core Assets in Service-Oriented Product Lines”

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

A M.Sc. Dissertation presented to the Federal University of Pernambuco in partial fulfillment of the requirements for the degree of M.Sc. in Computer Science.

Advisor: Silvio Romero de Lemos Meira
Co-Advisor: Eduardo Santana de Almeida

RECIFE, AUGUST/2010
I dedicate this dissertation to myself and all my family, friends and professors, my lovely girlfriend and my parents-in-law, who gave me all necessary support to get here.
Acknowledgements

The music “The long and winding road” written by Sir Paul McCartney translates all my feeling at the right moment I finished this work and I may have no words to describe my gratitude for all the people that helped me during this journey. Without all them, I could not go to the entire Master path. My excuses for the ones I forgot, it does not mean you were not important.

Initially, I would like to dedicate and thank this work for the six most important human beings in my life: (i) my wonderful mom, for your infinite love, caressing and dedication that transcends any distance in the world; (ii) my father (in memorian); (iii) myself for always go after my goals, regardless of whether I will succeed or fail; (iv) my lovely brothers which encouraged me to face off all the narrow way during this Master; (v) my lovely girlfriend, Paty (sunshine flower), for such love, dedication, caressing and all the confidence in me. I appreciate greatly the vital shelter in sad moments and the beautiful smiles of happy times, leaving a joyful heart and ready to face life’s battles. I love you, sunshine flower. (vi) my parents-in-law for welcoming me into your home at times when I needed. I would like to thank, also, my aunt Graça and my cousins, Júnior and Karine, for encouraging me to achieve my goals.

Next, I would like to thank the Reuse in Software Engineering Group (RiSE) group to give me the support and the space to discuss the aspects that were useful to my work. In special, my co-advisor Eduardo Almeida, for his patience in giving me the directions for my Master course. Flávio Medeiros, Hernan Munoz, Ivonei da Silva, Paulo Silveira, Ivan Machado, and Vanilson Burégio for the friendship that arose during the Master path, their revisions, thoughts and comments. My advisor, Silvio Meira, for accepting me as his student. Furthermore, I would like to thank the Professor Doctor Augusto Sampaio, for his patience and the incredible programming languages lessons. Professor Doctor Vander Alves, who participated as a collaborator in some papers produced during the Master course, with his great ideas and opinions.

I would like to thank all my friends that live in Maceió, for their patience when I could not go out with them to party. In addition, there are people that I have special thanks, friends who helped me to feel a little bit like home during this almost two years living in Recife. Ícaro Medeiros and Hugo Estanislaw, who were together with me since my transition from Maceió until Recife, in the most difficult moments. Flávio Medeiros and Hernan Munoz, who lived together in the same room with me during the last year of the Master, and in the last moments for finishing this work.

And last but not least, I want to thank God for giving me strength and illuminating my entire path to the end of this dissertation.
Welcome my son, welcome to the machine.
What did you dream?
It’s alright we told you what to dream.

—WELCOME TO THE MACHINE  (Roger Waters - Musician)
Resumo

Linhas de Produto de Software (LPS) e Orientação à Serviços (OS) são duas estratégias que vem sendo exploradas, tanto em pesquisas acadêmicas, quanto na indústria de software. Os conceitos de LPS e OS focam em objetivos distintos, porém, ambas usam o reúso de software como seu principal princípio. No entanto, para o resultado dessa junção ser otimizado, é necessário utilizar um processo de desenvolvimento bem definido. Caso contrário, a equipe de desenvolvimento poderá produzir software de maneira não sistemática, aumentando as chances de falha, o tempo e o custo de desenvolvimento.

Embora existam algumas abordagens explorando estes conceitos, há uma certa carência em como lidar com alguns desafios, como complexas atividades em relação a implementação de artefatos e preocupações tecnológicas específicas, os quais são importantes para o gerenciamento de aspectos variáveis. Portanto, a fim de entender e reduzir os problemas citados anteriormente, este trabalho propôs uma abordagem para a implementação de artefatos em linhas de produto de software orientada a serviços, constituída de um conjunto de atividades, tarefas, passos e orientações com entradas e saídas específicas, sendo cada uma delas realizada por um conjunto predefinido de papéis com responsabilidades definidas.

Para garantir a qualidade da abordagem desenvolvida, um modelo de decisão foi construído através de um estudo de caso inicial realizado na universidade, a fim de guiar engenheiros de software na tarefa de escolher tecnologias para implementar aspectos variáveis com o suporte de mecanismos de implementação de variabilidade. Por fim, a abordagem definida foi validada, também, por um estudo de caso inicial.

**Palavras-chave:** Orientação à Serviços, Arquitetura Orientada à Serviços, Linhas de Produto de Software, Desenvolvimento de Software, e Processos de Desenvolvimento de Software.
Software product lines (SPL) and Service-Orientation (SO) are two strategies being explored, both in academic research, as in software industry. The SPL and SO concepts focus on different goals, but the both use software reuse as their main principle. However, for the outcome of this combination to be optimized, there is a need to use a well-defined development process. Otherwise, the development team will produce software in a non-systematic way, increasing the chances of failure, the time and cost of development.

Although there are some approaches exploring this, there is a lack on how to deal with some challenges, such as complex activities for the implementation of assets and technology specific concerns, which are useful to manage the assets life cycle. Therefore, in order to understand and reduce the aforementioned problems, this work proposed an approach for implementing core assets in service-oriented product lines, consisted of a set of activities, tasks, steps and guidelines with inputs and outputs specified, each of which is held by a predefined set of roles with defined responsibilities.

In order to ensure the quality of the approach developed, a decision model was defined and validated through an initial case study conducted at the university, for guiding software engineers on the task of choosing technologies for implementing variable aspects with the support of variability implementation mechanisms. Finally, the approach defined was validated, also, by an initial case study.

**Keywords:** Service-Orientation, Service-Oriented Architectures, Software Product Lines, Software Development, Software Development Process.
## Contents

Acknowledgements ................................................. iii

Resumo ..................................................................... vi

Abstract ................................................................... vii

List of Figures ......................................................... xiii

List of Tables ......................................................... xiv

List of Acronyms ..................................................... 1

1 Introduction ......................................................... 3
   1.1 Motivation ...................................................... 4
   1.2 Problem Statement .......................................... 5
   1.3 Overview of the Proposed Solution ...................... 5
      1.3.1 Context ................................................... 6
      1.3.2 Outline of the Proposal ............................... 7
   1.4 Out of Scope ................................................... 8
   1.5 Statement of the Contributions ............................ 9
   1.6 Organization of the Dissertation ....................... 10

2 Software Product Lines ........................................... 11
   2.1 Software Reuse .............................................. 11
   2.2 Software Product Lines Background .................... 12
      2.2.1 Activities and Motivation ............................. 12
      2.2.2 Commonalities and Variabilities .................... 13
   2.3 Software Product Lines Implementation ............... 15
      2.3.1 Implementation Activities ............................ 15
      2.3.2 Implementing Variability ............................. 16
         Binding variants within services ...................... 17
         Parameters ................................................ 19
         Design Patterns: Strategy Design Pattern .......... 20
   2.4 Chapter Summary ............................................ 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.2 Domain Feature Model</td>
<td>133</td>
</tr>
<tr>
<td>A.3 Use Cases</td>
<td>134</td>
</tr>
<tr>
<td>A.4 R4ll Specification Architecture</td>
<td>134</td>
</tr>
<tr>
<td>B Travel Reservation Product Line</td>
<td>138</td>
</tr>
<tr>
<td>B.1 Project Context</td>
<td>138</td>
</tr>
<tr>
<td>B.2 Domain Feature Model</td>
<td>139</td>
</tr>
<tr>
<td>B.3 Domain Use Cases</td>
<td>140</td>
</tr>
<tr>
<td>B.4 Domain Design Architecture Reference</td>
<td>140</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>RiSE Labs Influences</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>RiSE Labs Projects</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Essential Activities for Software Product Lines [Clements and Northrop, 2001]</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Configuration Files Example</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Services encapsulating logic [Erl, 2005]</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Service-Oriented Architecture Roles and Operations [Papazoglou, 2003]</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>SOA services within a Product Line</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Measurement Framework</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>CDCS Metric Example</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>Complexity Analysis</td>
<td>50</td>
</tr>
<tr>
<td>4.4</td>
<td>Stability Analysis</td>
<td>53</td>
</tr>
<tr>
<td>4.5</td>
<td>Modularity Analysis</td>
<td>55</td>
</tr>
<tr>
<td>4.6</td>
<td>Understandability Analysis</td>
<td>59</td>
</tr>
<tr>
<td>4.7</td>
<td>An application of the decision model</td>
<td>61</td>
</tr>
<tr>
<td>4.8</td>
<td>Binding variants within services mechanism model</td>
<td>62</td>
</tr>
<tr>
<td>4.9</td>
<td>Parameters mechanism model</td>
<td>62</td>
</tr>
<tr>
<td>4.10</td>
<td>Strategy Pattern mechanism model</td>
<td>63</td>
</tr>
<tr>
<td>5.1</td>
<td>Bundle life cycle (Alliance, 2007)</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of a business process model</td>
<td>69</td>
</tr>
<tr>
<td>5.3</td>
<td>Example of a feature model</td>
<td>70</td>
</tr>
<tr>
<td>5.4</td>
<td>Approach Activities</td>
<td>72</td>
</tr>
<tr>
<td>5.5</td>
<td>Conference Management Domain Feature Model</td>
<td>74</td>
</tr>
<tr>
<td>5.6</td>
<td>Bundles Diagram</td>
<td>74</td>
</tr>
<tr>
<td>5.7</td>
<td>Implement Services and Components Activity</td>
<td>75</td>
</tr>
<tr>
<td>5.8</td>
<td>Notification Service Diagram</td>
<td>79</td>
</tr>
<tr>
<td>5.9</td>
<td>Submission Business Process Model</td>
<td>80</td>
</tr>
<tr>
<td>5.10</td>
<td>Submission Service Bundle Diagram</td>
<td>83</td>
</tr>
<tr>
<td>5.11</td>
<td>Assess Services and Components Activity</td>
<td>89</td>
</tr>
<tr>
<td>5.12</td>
<td>Refine Services and Components Specification Activity</td>
<td>90</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>6.1 Measurement Framework</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>6.2 Cyclomatic Complexity Graphic</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>6.3 Cyclomatic Complexity Box Plot Graphic</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>6.4 Outlier Analysis for Cyclomatic Complexity</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>6.5 Weighted Operations per Component or Service Graphic</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>6.6 Weighted Operations per Component or Service Box Plot Graphic</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>6.7 Outlier Analysis for Weighted Operations per Component or Service</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>6.8 Coupling between Components and Services Graphic</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>6.9 Coupling between Components and Services Box Plot Graphic</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>6.10 Outlier Analysis for Coupling between Components and Services</td>
<td>107</td>
<td></td>
</tr>
<tr>
<td>6.11 Lack of Cohesion over Operations Graphic</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>6.12 Lack of Cohesion over Operations Box Plot Graphic</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>6.13 Outlier Analysis for Lack of Cohesion over Operations</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>6.14 Instability Metric for Service or Component Graphic</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>6.15 Instability Metric for Service or Component Box Plot Graphic</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>6.16 Outlier Analysis for Instability Metric for Service or Component</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>A.1 R4II domain feature model simplified</td>
<td>134</td>
<td></td>
</tr>
<tr>
<td>A.2 R4II use cases diagram</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>A.3 R4II architectural styles</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>A.4 R4II modules</td>
<td>136</td>
<td></td>
</tr>
<tr>
<td>A.5 R4II authentication service, item and authorization components</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>A.6 R4II fine service, loan component, permission component, and user component</td>
<td>137</td>
<td></td>
</tr>
<tr>
<td>B.1 Travel reservation business process</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>B.2 Travel reservation domain feature model</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>B.3 Services, components, and its relationships through a service-oriented product line design</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>B.4 Services and components specification</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>B.5 Services and components specification</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>B.6 Travel agency product line entities modeling</td>
<td>144</td>
<td></td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Summary of the technologies implementation.</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Results for the Complexity Analyses.</td>
<td>49</td>
</tr>
<tr>
<td>4.3</td>
<td>ANOVA applied with level of significance of 5% to cyclomatic complexity</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>collected data.</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>ANOVA applied with level of significance of 5% to weighted operations per</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>component or service collected data.</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Results for the Stability Analyses.</td>
<td>52</td>
</tr>
<tr>
<td>4.6</td>
<td>ANOVA applied with level of significance of 5% to instability metric for</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>service or component collected data.</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>Results for the Modularity Analyses.</td>
<td>56</td>
</tr>
<tr>
<td>4.8</td>
<td>ANOVA applied with level of significance of 5% to coupling between</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>components and services collected data.</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>ANOVA applied with level of significance of 5% to lack of cohesion over</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>operations collected data.</td>
<td></td>
</tr>
<tr>
<td>4.10</td>
<td>ANOVA applied with level of significance of 5% to concern diffusion over</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>components and services collected data.</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>General data collected during the case study.</td>
<td>100</td>
</tr>
<tr>
<td>6.2</td>
<td>Results for the Complexity Analysis.</td>
<td>105</td>
</tr>
<tr>
<td>6.3</td>
<td>Results for the Modularity Analysis.</td>
<td>109</td>
</tr>
<tr>
<td>6.4</td>
<td>Results for the Stability Analysis.</td>
<td>111</td>
</tr>
<tr>
<td>6.5</td>
<td>Results for the Understandability Analysis.</td>
<td>111</td>
</tr>
<tr>
<td>6.6</td>
<td>Effort Analysis.</td>
<td>112</td>
</tr>
<tr>
<td>B.1</td>
<td>Fulfill Itinerary.</td>
<td>140</td>
</tr>
<tr>
<td>B.2</td>
<td>Check Itinerary.</td>
<td>140</td>
</tr>
<tr>
<td>B.3</td>
<td>Reservation.</td>
<td>141</td>
</tr>
<tr>
<td>B.4</td>
<td>Payment.</td>
<td>141</td>
</tr>
<tr>
<td>B.5</td>
<td>Notification.</td>
<td>141</td>
</tr>
<tr>
<td>B.6</td>
<td>Compensation.</td>
<td>142</td>
</tr>
</tbody>
</table>
List of Acronyms

ANOVA  ANalysis Of VAriance
BPM   Business Process Model
BPMN  Business Process Modeling Notation
CBD   Component-Based Development
CBCS  Coupling between Components and Services
CORBA Common Object Request Broker Architecture
CDCS  Concern Diffusion over Components and Service
CC    Cyclomatic Complexity
CIN-UFPE  Informatics Center - Federal University of Pernambuco
DI    Dependency Injection
EJB   Enterprise Java Beans
LOC   Lines of Code
FODA  Featured-Oriented Domain Approach
GQM   Goal Question Metric
HTTP  Hypertext Transfer Protocol
IMSC  Instability Metric for Service or Component
JAX-WS Java API for XML Web Services
JAX-RPC Java API for XML Remote Protocol Control
JVM   Java Virtual Machine
LCOO  Lack of Cohesion Over Operations
OO    Object-Oriented
OASIS The Organization for the Advancement of Structured Information Standards
OSGi  Open Services Gateway Initiative
R4II  Rental for All Software Factory
REST  Representational State Transfer
RMI   Remote Method Invocation
RPC   Remote Procedure Call
RISE  Reuse in Software Engineering Group
SCA   Service Component Architecture
SoC   Separation of Concerns
SOA   Service-Oriented Architecture
SOC   Service-Oriented Computing
SO    Service-Orientation
SOD   Service-Oriented Development
SO-DDSA Service-Oriented Domain Design Specification Architecture
SPL   Software Product Lines
SOPL  Service-Oriented Product Lines
SOPLE-IM Service-Oriented Product Line Engineering IMplementation
SpringDM Spring Dynamic Modules
UFPE  Federal University of Pernambuco
UML   Unified Modeling Language
URL   Uniform Resource Locator
WOCS  Weighted Operations per Component or Service
WS    Web Service
WSDL  Web Services Description Language
XML   eXtensible Markup Language
Introduction

The long and winding road...
—LET IT BE (Paul McCartney)

Software reuse is a key factor for enterprises interested in productivity gains, reduced development costs and time-to-market, and increased software quality (Northrop, 2002). However, these benefits are not assured by the application of ad-hoc reuse, which is an opportunistic reuse, not systematic, and generally restricted to source code. In this sense, two approaches emerge towards systematic and predictable way to achieve reuse:\footnote{The systematic reuse process is characterized by the planned, managed, and monitored execution of the organization reuse practices.}: Software Product Lines (SPL) and Service-Orientation (SO) (Pohl et al., 2005; Erl, 2005; Segura et al., 2007). Furthermore, the combination of SPL and SO, called as service-oriented product lines, has received a considerable attention in research and practice lately. In this context, service-oriented applications that share common business processes or part of them that can be developed as SPL.

This combination can increase the service reuse rates through managed variability and supports the non-opportunistic reuse of SO environments (Cohen and Krut, 2007; Segura et al., 2007). It also aids the development of flexible, loose coupled and dynamic software product lines due to the specific characteristics of the service-oriented architectures, e.g., contract-based communication and dynamic service discoverability. However, service-oriented product lines consider several products that can be customized for specific customers, thus, making the implementation of such customization an essential task to deal with these circumstances. In this way, the focus of this dissertation is providing and exploring the implementation concerns in order to maximize the benefits of systematic
1.1. MOTIVATION

The remainder of this Chapter describes the focus of this dissertation and starts by presenting its motivation in Section 1.1 and a clear definition of the problem in Section 1.2. An overview of the proposed solution is presented in Section 1.3, while Section 1.4 describes some related aspects that are not directly addressed by this work. Section 1.5 presents the main contributions and, finally, Section 1.6 describes how this dissertation is organized.

1.1 Motivation

Software reuse is a key factor for enterprises interested in productivity gains, reduced development costs and increased software quality (Northrop, 2002). In the context of software reuse, Software Product Lines (SPL) explores commonality and variability among related products, while Service-Orientation (SO) is a way of developing service-based applications, which has in its roots the Service-Oriented Architecture (SOA). In addition, the combination of these two paradigms encourages organizations to reuse existing services and components instead of redeveloping them repeatedly from scratch.

On the one hand, the development of a SPL using software architectures and implementation supports systematic planned reuse, high software flexibility and contract-based communication. Moreover, it produces a common, managed set of core assets for rapidly developing customized products for specific customers. On the other hand, SOA is a model with the support of designing, developing, deploying, and managing systems in which services provide reusable business functionality (Erl, 2005; Smith and Lewis, 2009). Furthermore, applications or other service consumers are built using functionalities from available services, and the SOA infrastructure enables the discovery, composition, and invocation of services (Erl, 2005; Smith and Lewis, 2009).

However, the issues related to the combination of SPL and SO concepts, especially, variability implementation and service technological concerns become a very challenging task. The challenges consist the implementation activities that become more complex, since it is important to understand the variability mechanisms for realizing variabilities in different levels of granularity (e.g., components, services, service-compositions) at runtime (Gacek and Anastasopoulos, 2001). Consequently, it becomes more critical and there is a need of mapping these aspects with the interaction and life cycle of services through a service technology, making the assembly of distinct services easier to perform, since it is expected that it is under some quality factors, such as modularity (loose coupling), complexity, discoverability and composability.
1.2. PROBLEM STATEMENT

In the literature, many studies (Smith and Lewis, 2009; Segura et al., 2007; Günther and Berger, 2008) have discussed the relationship between SPL and SOA concepts. Nevertheless, they focus on services as core assets in SPL in which the services handle variabilities and they address the relationship between SOA and SPL from a conceptual level but do not go into details about the implementation tasks. Consequently, they do not provide decisions for recommending specific variability mechanisms according to granularity, modularity and Separation of Concerns (SoC) principles. Finally, they do not provide the relationship between variability and technological concerns, such as the maintenance of reusable services as SPL core assets through a transparent life cycle mechanism, neither address a quantitative analysis.

In order to understand and mitigate this scenario, this study presents an approach to implement core assets in service-oriented product lines herein named Service-Oriented Product Line Engineering IMplementation (SOPLE-IM). The approach consists of a set of activities, tasks, steps and guidelines for implementing variability with inputs and outputs specified, each of which is held by a predefined set of roles with defined responsibilities. In addition, the proposed approach was implemented by a service technology to control the core assets life cycle. Finally, in order to assess the quality of the proposed approach, case studies were defined to evaluate a decision model and the approach.

1.2 Problem Statement

The goal of this dissertation can be stated as follows:

*This work investigates the problem of the core assets implementation in service-oriented product lines, which is related to variability implementation and the use of service technologies to manage the life-cycle of its core assets in a systematic and practical method.*

1.3 Overview of the Proposed Solution

In order to implement a set of core assets, it was developed a service-oriented product lines implementation approach. The remainder of this section describes the context where it was developed and the outline of the proposal.
1.3. OVERVIEW OF THE PROPOSED SOLUTION

1.3.1 Context

This dissertation is part of the Reuse in Software Engineering Group (RiSE) (Almeida et al., 2004), formerly called RiSE Project, whose goal is to develop a robust framework for software reuse in order to enable the adoption of a reuse program. RiSE Labs is influenced by a series of areas, such as software measurement, architecture, quality, environments and tools, and so on, in order to achieve its goal. The influence areas are depicted in Figure 1.1.

![RiSE Labs Influences](image1)

**Figure 1.1** RiSE Labs Influences

Based on these areas, the RiSE Labs is divided in several different projects related to Software reuse, as shown in Figure 1.2:

![RiSE Labs Projects](image2)

**Figure 1.2** RiSE Labs Projects

- **RiSE Framework**: It involves reuse processes (Almeida et al., 2005), Component
1.3. OVERVIEW OF THE PROPOSED SOLUTION

Certification (Alvaro et al., 2006) and Reuse Adoption and Adaptation Processes (Garcia et al., 2008).

- **RiSE Tools**: Research focused on software reuse tools, such as the Admire Environment (Mascena et al., 2006), the Basic Asset Retrieval Tool (B.A.R.T) (Santos et al., 2006), which was enhanced with folksonomy mechanisms (Vanderlei et al., 2007), Semantic Layer (Durao, 2008), Facets (Mendes, 2008) and Data Mining (Martins et al., 2008), the Legacy InFormation retrieval Tool (LIFT) (Brito et al., 2008), the Reuse Repository System (CORE) (Burégio et al., 2008), and the Tool for Domain Analysis (ToolDAy) (Lisboa et al., 2007).

- **RiPLE**: Stands for RiSE Product Lines Engineering Process and aims at developing a methodology for Software Product Lines, composed of scoping, requirements engineering, design, implementation, test, and evolution management.

- **SOPLE**: Development of a methodology for Software Product Lines based on Service-Oriented concepts.

- **MATRIX**: Investigates the area of measurement in reuse and its impact in quality and productivity.

- **BTT**: Research focused on tools for detection of duplicated change requests (Cavalcanti, 2009).

- **Exploratory Research**: Investigates new research directions in software engineering and its impact on reuse.

- **CX-Ray**: Focused on understanding with empirical data the C.E.S.A.R, its processes and practices in software development, including reuse.

This dissertation is part of the SOPLE project and its goal is to support the core assets implementation in The service-oriented product lines context.

1.3.2 Outline of the Proposal

The goal of this dissertation is to implement core assets in the context of service-oriented product lines, by defining a systematic approach composed by three main activities: implement services and components, refine services and components specification, and assess services and components integrated in a macro flow that guides all approach
activities. In summary, the implement services and components provide guidelines for the software engineer to specify and implement services and components in the provider and consumer side; refine services and components specification aims to perform the refinement of the detailed design and the service and components specification and implementation; and, finally, assess services and components aims at assess through software metrics the service providers/consumers and components implemented in order to guarantee that its implementation is in accordance with some criteria (e.g., modularity and complexity).

1.4 Out of Scope

- **Product Development.** An important issue in a SPL and service-oriented product lines approach is to create individual products by reusing the core asset, i.e. performing the products development with reuse. However, this aspect can be as complex as core assets development, involving the definition of activities, inputs, outputs, guidelines and roles. Thus, it is out of the scope of this work, but it has direct relationship with the approach defined in this dissertation.

- **Evolution Management.** The SPL and service-oriented product lines evolution control is ensured by appropriate practices of changes management, including changes in implementation artifacts. These practices are part of the Evolution Management approach, thus they will not be considered in this study.

- **Domain Architecture or Product Line Architecture (PLA).** The domain architecture or PLA establishment is not detailed. The good practices from SOPLE-DE (Service-Oriented Product Line Engineering DEsign) (Medeiros, 2010) attend the need of defining common and variable points within the domain architecture.

- **Quality Attributes.** For the same reasons of domain architecture, the quality attributes (performance and availability) represented with variability establishment is not detailed. It is an extensive area with a great discussion in the academy. However, there is lack of historical data in such area.

- **Scalability and Portability Metrics.** Measurement activities are essential in any engineering process. Both measurement activities inside the approach and metrics to be used outside the approach (to formally evaluate) could be incorporated to the
1.5. STATEMENT OF THE CONTRIBUTIONS

process. Scalability and Portability requires extensive applications in more than three projects at least. However, it is not considered in this initial study.

- **Composability Criterion.** In a service-oriented environment, the composition of services enable its reusability in order to provide business functionality. As the the size of the projects using in this dissertation is limited to a simple application, thus it will not possible to draw conclusion on services composability. Thus, composability is out of scope in this dissertation. However, when applying coupling and cohesion metrics, it is possible to make an initial study in this direction. However, will encourage to take this criterion as an essential taks in further work.

- **Combination of Technologies.** The combination of service technologies provide implementations with other service components written in different languages and existing in a distributed network of systems using a range of communication methods. However, due to the scope of this study, the combination will not be considered.

1.5 Statement of the Contributions

As a result of the work presented in this dissertation, a list of the main contributions may be enumerated:

- An assessment for comparing technologies to implement core assets in service-oriented product lines through the definition and planning of a case study.
- A detailed analysis was also performed with the assessment.
- The definition of a decision model for guiding software engineers on the task of choosing technologies with the support of variability implementation mechanisms to implement core assets in service-oriented product lines.
- The definition of an implementation approach for service-oriented product lines.
- The definition and planning of a case study to evaluate the service-oriented product lines implementation approach.
- A detailed analysis was also used in order to understand the approach efficiency for generating assets according to a set of criteria.
1.6 Organization of the Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2 discusses the software product lines basic concepts and activities, as well as product lines implementation. The relation between software product lines and implementation activities is also discussed.

Chapter 3 introduces the service-oriented development, its fundamental aspects and the circumstances in which its use is appropriate, such as, implementation activities concerning technological aspects.

Chapter 4 presents an assessment through the definition, planning, operation, analysis and interpretation of a case study, which evaluates the use of technologies to implement core assets in service-oriented product lines.

Chapter 5 describes an approach to implement core assets in the service-oriented product lines context, presenting the principles, roles associated, activities, guidelines, inputs and outputs and the key concepts of the approach.

Chapter 6 presents the definition, planning, operation, analysis and interpretation of the case study, which evaluates the approach to implement core assets in service-oriented product lines.

Chapter 7 concludes this dissertation by summarizing the findings and proposing future enhancements for the solution.

Appendix A describes the artifacts produced by the pilot project used in the case study on technologies reported in Chapter 4.

Appendix B describes the artifacts produced by the pilot project used in the case study on the service-oriented product line implementation approach reported in Chapter 6.
During the last few years, Software Product Lines (SPL) received considerable adoption in the software industry and academy, and have proved to be a very successful approach to intra-organizational software reuse (Bosch, 2002). In this sense, the main goal of this chapter is to cover the basic concepts and ideas involved with software product lines. Thus, in section 2.1, the concepts of software reuse is discussed. In section 2.2 the basic concepts related to SPL are discussed. In section 2.3.2, some variability mechanisms responsible to implement variability in SPL are covered. Finally, a briefly summary about the chapter is presented in section 2.4.

2.1 Software Reuse

The idea of software reuse emerged in the early of 60’s, during the NATO Software Engineering Conference, generally considered the birthplace of the field, which the focus was the software crisis. In particular, the problem of building large, reliable software systems in a controlled and cost-effective way (Almeida et al., 2007). In this way, the software reuse has been a goal in software engineering, which software components usually compared to routines should be available in families arranged according to precision, robustness, generality and performance (Mcilroy, 1968). These components were denominated libraries, allowing components to be widely applicable to different machines and users.

The library approach requires the establishment of reusable components for indexing, locating and distinguishing among similar components (Prieto-Diaz and Freeman, 1987).
However, some problems with this approach include: managing the large number of components that such a reuse library. Moreover, this reusable library is composed of building blocks used to construct new systems. In this sense, according to (Gomaa, 2004), instead of reusing an individual component, it is much more advantageous to reuse a whole design or subsystem consisting of components and their interconnections. This scenario lies in contrast with traditional software reuse approaches, where the overall reuse is low, and the emphasis is on code to be reused. Consequently, reuse requirements, architecture, code and tests has much greater potential rather than only component reuse, since it is a large-grained reuse and at higher levels of abstraction.

2.2. SOFTWARE PRODUCT LINES BACKGROUND

According to (Clements and Northrop, 2001; Pohl et al., 2005), SPL is a set of systems that shares a set of common managed set of features satisfying specific needs of a particular segment need. SPL explores the commonalities and variabilities among related products that can be constructed from a set of reusable and customizable core assets. By reusing such core assets, it is possible to build a large scale of flexible products through variable and common aspects defined according to the customer requirements (Clements and Northrop, 2001; Pohl et al., 2005). Reusable assets can be artifacts that are part of the development process, such as requirements and design documents. On the product derivation side, these assets can be, for example, binary files and documentation.

2.2.1 Activities and Motivation

SPL involves three essential activities: Core Asset Development, Product Development and Management (Clements and Northrop, 2001), which are depicted in Figure 2.1. The Core Asset Development activity is responsible for the creation of the common assets (e.g., models, requirements, design, source code and tests) of SPL and the evolution of the assets in response for the product feedback and market needs. Product Development activity creates individual products by reusing the core assets, gives feedback to core asset development, and evolves the products. Furthermore, the Management activity includes technical and organization management that are responsible for the requirements control and the coordinate between core asset and product development.

The main goal of SPL is to increase the productivity and build products faster to provide a range of benefits described in (Clements and Northrop, 2001; Pohl et al., 2005) as follows: Reduction of Development Costs, which implies in a cost reduction for each
2.2. SOFTWARE PRODUCT LINES BACKGROUND

application, since there is no need to develop them from scratch. As a consequence, the initial costs to develop a SPL are higher than the initial cost to develop a single system. However, the total accumulated cost is lower as the SPL evolves, when comparing to the development of single systems; *Enhancement of Quality*, where core assets are developed and tested to be reused in different products, there is an increase in quality assurance of many instances, which make them more reliable than instances of products developed from scratch (Matos, 2008); and, *Reduction of Time-to-Market*, that enables the short time that the assets are delivery for the customers. However, the creation of these assets increases the time to market, since they are build first.

In order to develop a SPL, different strategies have been applied. In (Krueger, 2002), the products are created using the *proactive approach*, which products come to market quickly with a minimal effort for writing code. The proactive approach is concerned with analysis, design and implementation of a SPL. In the *reactive approach*, the organizations begin with one or a small number of products that they already have and, then, the SPL core assets are generated (Krueger, 2002). On the other hand, the *extractive approach* extracts existing products into a single SPL (McGregor et al., 2002). In the both strategies, the applicability of the SPL is advised for the domains where there is a demand for specific products, but with a set of common characteristics and well-defined variabilities.

### 2.2.2 Commonalities and Variabilities

In SPL, it is necessary to identify the commonalities and variabilities among the products in terms of its requirements. Commonalities are usually defined as the set of decisions that are common to all products. In this way, it can be considered as the basis for every SPL
2.2. SOFTWARE PRODUCT LINES BACKGROUND

(Matos, 2008). On the other hand, variability is tangible difference among the products of the SPL that can be revealed and distributed in every phase of development cycle of the product line. Variabilities are the main key to understand SPL, since it enables the development of customized products by reusing predefined and adjustable assets (Pohl et al., 2005).

In the context of SPL, for some researchers, both commonalities and variabilities are called as features, which usually represents reusable requirements. Kang et al. (Kang et al., 1990) define a feature as a prominent and distinctive aspect or characteristic that is visible to several stakeholders, such as end users, domain experts, developers, and, so on. Moreover, Kang et al. (Kang et al., 1990) proposed a model called as feature model as part of the Featured-Oriented Domain Approach (FODA). This model is used for modeling common and variable functionalities among the products of the SPL. However, there are indeed other approaches whose focus is modeling commonalities and variabilities in SPLs, such as (Ardis and Weiss, 1997; Bayer et al., 1999; Atkinson et al., 2002; Clements and Northrop, 2001; Muthig and Patzke, 2002; Gomaa, 2004; Pohl et al., 2005; Lisboa et al., 2007).

According to (Kang et al., 1990; Muthig and Patzke, 2003), features can be classified as follow:

- **Mandatory Features**: represent the common functionality that must be presented in all products of the family.
- **Optional Features**: a functionality that must be part of a product or not.
- **Alternative Features (XOR-Features)**: it belongs to a group of features from which no more than one must be selected, i.e., they are mutual-exclusive functionalities.
- **OR-Features**: the selection of more than one feature of this group is allowed.

As a refinement of the features classification, Coplien (Coplien, 2000) classified the relation of variability and commonality in terms of:

- **Positive Variability**: add an extra behavior to the commonality based without the need to modify the core.
- **Negative Variability**: removes existing behavior in the commonality base, breaking some existing assumptions in the commonality model.

Positive variability is preferred against negative variability, since it preserves the common structure, increasing the understandability and maintainability of the overall SPL (Muthig and Patzke, 2002).
2.3 SOFTWARE PRODUCT LINES IMPLEMENTATION

2.3 Software Product Lines Implementation

According to (Coplien, 2000), in traditional software engineering, the implementation activities refine a system’s architecture down to a level that can be interpreted by a machine. Hence, the information captured by implementation artifacts is at the lowest level of abstraction compared to the content of all other development artifacts (Muthig and Patzke, 2002). On the other hand, SPL implementation has the focus on the detailed design and the implementation of reusable software assets based on the reference architecture defined in the design discipline. In this context, the main reusable software assets are mainly reusable components and interfaces (Clements and Northrop, 2001; Pohl et al., 2005).

In addition, SPL incorporate variability implementation mechanisms that enable the application implementation to select the variants and build an application with the reusable components and interfaces. However, other artifacts such as, thread designs, database tables, data streaming formats, among others, are also products of the implementation phase. In the next subsections, some activities concerning software product lines implementation activities and variability implementation mechanisms are briefly discussed.

2.3.1 Implementation Activities

According to (Pohl et al., 2005), the reference architecture determines the decomposition of an application into software artifacts, such as components and interfaces. The detailed design provides designs for each component and interface, and, after validation, they are implemented. In order to detail the implementation concerning variable aspects, Pohl et al. (Pohl et al., 2005) identified three activities that are described as follows.

Realizing Interfaces. Interfaces are intended to specify functionalities provided by several components and required by others through a contract between the parts. Variability is implicitly through the independent abilities of having variability both at the providing component’s side and at the requiring side. In this context, the interface itself is an invariant and both the providing and requiring components have to interpret it in the same way (Pohl et al., 2005). It means that the components are responsible to encapsulate variants and to decide which of the variant will be exposed in the interface in a transparent manner.

Realizing Variable Components. As aforementioned in the realizing interfaces activity, the component design is constrained by interfaces provided/required and
the variability is encapsulated in components. Thus, it is stated that such variabilities of a SPL have to be realized in terms of reusable components (Pohl et al., 2005). In many cases, several variants of the component are realized, where each of them combines certain variants. Clements and Northrop stated in (Clements and Northrop, 2001) that the component development provides the required variability in the developed components through the use of variability implementation mechanisms according to specific feature types.

**Compilation.** It is a simple activity whose goal is to compile the components into object files. In order to perform it, the component and its interfaces must be designed and implemented correctly following specific guidelines as described in the previous subsections. Thus, when the object files are created, they must be linked into working executables during the application realization.

### 2.3.2 Implementing Variability

The development of SPL requires the management of commonality and variability among a set of related products. Variability is the ability to change or customize software systems (van Gurp et al., 2001). In this context, there are several customers expecting products with specific features according to their requirements. Hence, the derivation of products must be easy without much effort in order to deliver products fast and customized to specific customers.

In this sense, specific variability mechanisms have to be selected to implement variability in a simple and maintainable way. A variability mechanism is responsible for implementing a *variation point*. A variation point is a place in software system where choices are made as to which *variant to use*. Moreover, variant is a specific decision that is bound to a variation point for each individual instance of a SPL (Pohl et al., 2005). The selection of the variability implementation mechanism must be based on a systematic analysis of technical requirements, constraints, the context where the mechanism will be used, and the binding time of the variability to be implemented (Gacek and Anastasopoulos, 2001).

As the context of this dissertation is the service-oriented product lines, thus we decide to use variability mechanisms that provide support to map variability concerns into core assets represented as services and components. The idea of these mechanisms is to cover infrastructure usage patterns and best practices based on runtime configuration required in service-oriented product line environment. For example, dynamic variabilities bound at runtime according to the customer needs. In this sense, for the study’ purpose we
2.3. SOFTWARE PRODUCT LINES IMPLEMENTATION

selected three variability mechanisms: *Binding variants within services, Parameters* and *Strategy Design Pattern*.

However, there are indeed other variability mechanisms, such as Aggregation/Delegation, Aspect-Oriented Programming (AOP), Conditional Compilation, Mixins, Generic Programming, Domain Specific Languages, Defaults (Null Objects), Active Libraries, Program Transformations and Refinements, identified and described in the literature *(Gamma et al., 1995; Czarnecki and Eisenecker, 2000; Coplien, 2000; Gacek and Anastasopoulos, 2001; Muthig and Patzke, 2003; Schnieders, 2006; Ribeiro, 2008; Nascimento et al., 2008; Matos, 2008)* not considered for this study’s purpose. These approaches were not discussed here, since one part is required only in compile-time environments (Conditional Compilation, Defaults, Active Libraries, and Refinements), and many provide more complex concerns that is out of the studys’ scope. Thus, there may be particular studies to cover them, e.g., AOP, Generic Programming, DSL, and Program Transformations.

Hence, the objective of this section is not to provide a detailed description of each mechanism, but to briefly introduce their concepts and to discuss how they support the implementation of variabilities.

**Binding variants within services**

In service-oriented applications, the binding time of services or components are usually defined at runtime, without the need to explicit dependencies among them. It allows the loose coupling, since they do not need to know who will implement its functionalities. Thus, enabling what will be exposed and what will be hidden in a modular way. In this sense, Dependency Injection (DI) *(Fowler, 2004)* as a mechanism for binding variabilities may be applied using a container to perform such binding. DI is a design principle that aims to decouple the component and service providers from consumers. The components are instantiated externally by a container, which is responsible for controlling the components and services dependencies.

In the context of service-oriented product lines development, dependency injection may be used to bind different components and services considering specific customer’s requirements to implement variability. As a consequence, such approach may be useful as a mechanism to implement SPL variabilities at runtime. Moreover, DI requires much less infrastructural code when comparing to an approach, which uses inheritance in combination with the Decorator/Strategy pattern *(Gamma et al., 1995)*. However, DI depends on a container responsible for finding and passing the necessary services to the component.
The Listing 2.1 shows how the *Binding variants within services mechanism* works in the Spring Container (*Spring, 2009*).

**Listing 2.1** Binding variants within services Example using the Spring Container

```xml
<beans>
  <bean id="AuthenticationService"
    class="br.cin.ufpe.authenticationService.spring.AuthenticationService">
    <property name="authentication" ref="AuthenticationByPassword"/>
  </bean>
  ...
</beans>
```

In order to authenticate an user, the *AuthenticationService* (line 3) depends on an authentication type (line 4). The container will create and pass to the Authentication-Service service an instance of the authentication type configured. In this example, the *AuthenticationByPassword* will be instantiated (line 8). The AuthenticationByPassword is depicted in Listing 2.4. Moreover, the *authentication property* is an attribute of the AuthenticationService class (Listing 2.2). This class must provide the *setAuthentication* method, so that the container can set up the *Authentication* (Listing 2.3) type correctly.

**Listing 2.2** Authentication Service Component Example

```java
public class AuthenticationService {
    private Authentication authentication;

    public void setAuthentication(Authentication authentication){
        this.authentication = authentication;
    }

    public boolean doAuthentication(User user){
        return this.authentication.authenticate(user);
    }
}
```

**Listing 2.3** Authentication Component Example

```java
public interface Authentication {
    [...]
    boolean authenticate(User user);
}
```

**Listing 2.4** AuthenticationByPassword Class Example

```java
public class AuthenticationByPassword implements Authentication {
    private UserService userService;
    [...]
}
Since there are no explicit dependencies, it is possible to develop another product even if it requires other functionalities, such as Authentication by Card. In this way, all we have to do is modifying the spring file adding an AuthenticationByCard instance. Furthermore, it is also necessary in the same file to change the AuthenticationService (Listing 2.1, Line 4) for its implementation be referenced now by the AuthenticationByCard. It is important to emphasize that the AuthenticationByCard should be already been developed likewise the AuthenticationByPassword (Listing 2.4).

Parameters

In the context of a service-oriented product line, in which products have a potentially number of customers, expecting different features, they often have to be configured before or after it is delivered to the customer, or even at runtime. Furthermore, it is desirable that after delivering the application for the customers, no extra development effort needs to be done when configuring the system. In addition, the customers require that they configure some aspects of the software by themselves, since they do not have knowledge about the source code of the software.

As a consequence, the use of a configuration file can play an important role, since it can be useful to separate the source code from the definition of the values that may change from one version of the application to another. These values are usually mapped as Parameters. When executing, the application loads the configuration file and check in the source code if there are any entries referencing these files (Santos and Santos, 2005).

Figure 2.2 shows an example of Parameters, where occurs the mapping between parameters and values. The example shows a configuration file and a code snippet. Suppose that an application has a basic functionality, such as penalty for delay feature. Now, the customer needs to introduce another feature type, for example, penalty for violation for calculating the infringement committed by users. In this case, the productConfigura-
tion.properties file should be selected. Thus, when the customer requests for the penalty for violation feature, then the application loads the configuration file and searches for the appropriate value. As can be seen in Figure 2.2, the PropertiesFactory class will be responsible for checking if the feature is available. In the case that feature is not available, then a message is sent for the customer indicating that the application does not provide
2.3. SOFTWARE PRODUCT LINES IMPLEMENTATION

that feature.

```java
public abstract class Authentication {
    public boolean authenticate(User user);
}

public class PasswordAuthentication extends Authentication {
    public boolean authenticate(User user);
}
```

**Figure 2.2** Configuration Files Example

**Design Patterns: Strategy Design Pattern**

The Strategy Pattern (Gamma et al., 1995) can be defined as a mechanism that defines a family of algorithms, encapsulate each one, and make them interchangeable. Firstly, it identifies the behaviors or algorithms that vary and separate them with the intention to achieve dynamicity. The behaviors or algorithms are encapsulated in sub-process implementation (Concrete Strategy) that implements a common interface or abstract class through Inheritance. Each of these objects is referenced by classes (Context) through the common interface (Strategy).

This mechanism is an alternative for using subclasses to implement different functionality and actually allows us to plug-in new strategies that has variations dynamically. In a Service-Oriented context, a product line may support dynamic variations trough the combination of mechanisms that can handle such dynamic behavior. In the context of the Strategy Pattern, we can dynamically decide which strategy to choose, but the choice is still or limited by the set of strategies defined at compile-time. In this sense, a technology which controls the modules at run-time may be applied to enable the full dynamicity of the Strategy Pattern.

Listing 2.5 shows the implementation of a common variation of the Strategy Pattern.

**Listing 2.5** Strategy Pattern Example
The abstract class \textit{Authentication} (line 1) defines the common behavior of an authentication scenario. The strategy is to define the authentication type required in a particular application. The different strategies (or authentication types) could be defined depending on the application characteristics, for example, \textit{AuthenticationByCard} or \textit{AuthenticationByPassword}. As can be seen in Listing 2.5, more specifically in lines 6 and 15, two strategies were defined: \textit{PasswordAuthentication} and \textit{CardAuthentication}. The context class \textit{AuthenticationService} provides a \textit{setter method} that is responsible for instantiating the adequate strategy according to the application characteristics mentioned above and is responsible for executing the strategy behavior (e.g., authenticate method).

\section*{2.4 Chapter Summary}

In this chapter some relevant topics were discussed in order to provide a better understanding of this dissertation, such as software reuse, and how it is placed in the context of software product lines concept, as well as some variability mechanisms that can be
applied within variability implementation.

In the next chapter, we present the main concepts related to service-oriented development, including service-oriented computing, service-oriented architectures and the main basis for understanding the connection between software product lines and service-oriented development.
Service-Oriented Development (SOD) present approaches for building distributed system that are responsible to delivery application functionality as services to either end-user applications or other services. It uses software reuse (Chapter 2) in its roots, since features may be decomposed into distinct units of logic (services) and components, or in more complex systems. Moreover, for allowing such decomposition, it is composed of an architectural model, called as Service-Oriented Architecture (SOA). Additionally, new research contributions (Raatikainen et al., 2007; Trujillo et al., 2007; Segura et al., 2007) have been discussed possibilities for increasing the SOD capabilities for reuse, by providing a way for combining service-orientation concepts with software product lines. This new paradigm is called as service-oriented product lines (Cohen and Krut, 2007), and uses the basic concepts of SOA and SPL combined to build customized applications according to specific customers and market segment needs.

Thus, the goal of this chapter is to introduce the service-oriented development, its fundamental aspects and the circumstances in which its use is appropriate. In section 3.1, it is discussed the service-oriented development in terms of Service-Oriented Computing (SOC). In Section 3.2, it is presented presents the principles of SOA. Secion 3.3 presents the evolution of the SOD field with the combination between service-orientation and SPL. Moreover, the service-oriented development concepts are discussed in Section 3.4, as
3.1 Service-Oriented Computing

Service-Oriented Computing (SOC) is a paradigm whose goal is to develop software systems integrated and released in a centrally synchronized way in distributed environments. It encompasses several aspects, including its own design paradigm and design principles, design pattern catalogs, pattern languages, a distinct architectural model, technologies and frameworks (Erl, 2005). Furthermore, it uses services as fundamental elements for developing service-oriented applications (Papazoglou and Georgakopoulos, 2003). According to (Raatikainen et al., 2007), such services must have readily available functionality and be platform independent, self-describing, and location transparent computational element (i.e., they are developed and deployed independently), that support rapid low-cost composition of distributed applications.

Furthermore, SOC encompasses a set of cooperating services where application components are assembled with minimal effort into a network of services that can be loosely coupled to create flexible dynamic business process (Papazoglou et al., 2008). In this sense, services appear in two flavors: simple and composite services (Papazoglou, 2003). The service as a unit corresponds to functionality that is in place and readily available. On the other hand, composite services involve assembling existing services that access and combine information from possibly multiple service providers.

Service providers are usually used by organizations that search service implementations, supply their service descriptions, and provide business support over the distributed environment. In order to enhance efficiency, agility, and productivity of an organization by positioning, designs, and developing services as the primary means, SOC relies in another concept called as Service-Oriented Architecture (SOA). SOA establishes an architectural model that provides an infrastructure into an interconnected set of reusable services, each one accessible through standard interfaces and messages protocols (Papazoglou, 2003). In the next subsection, the SOA paradigm is briefly discussed in details.

3.2 Service-Oriented Architectures

In the current years, methodologies for programming and system architectures have been evolved. Such evolution can be described in the context of centralized monolithic single-threaded systems to distributed parallel systems. Currently, a new research method-
3.2. SERVICE-ORIENTED ARCHITECTURES

ology, called Service-Oriented Architecture (SOA) for building such complex distributed systems is receiving a great attention in academy. In order to provide more detail about this methodology, its motivations, roles, and characteristics are briefly discussed in the next subsections.

3.2.1 SOA Motivations

The Organization for the Advancement of Structured Information Standards (OASIS)\(^1\) (OASIS, 2006) has attempted to define a Reference model for SOA whose goal is to aid to define the several terms that are used in SOA. According to (OASIS, 2006), SOA is a paradigm that provides a uniform means to offer, discover, interact with and use capabilities that may be under the control of distributed and different ownership domains.

According to (Erl, 2005), SOA is a model in which automatic logic can be decomposed into smaller, distinct units of logic. Collectively, these units comprise a larger piece of business automation logic. Individually, these units can be distributed. In this way, SOA encourages individual units of logic to exist autonomously yet not isolated from each other. When building an automation solution consisting of services, each service can encapsulate a task performed by an individual step or a sub-process comprised as a set of steps as depicted in Figure 3.1. In a more complex scenario, services may encompass the logic encapsulated by other services.

\[\text{Figure 3.1} \text{ Services encapsulating logic [Erl, 2005].}\]

The relationship among the services is based on an understanding that for services to interact, they must be aware of each other (Erl, 2005). This awareness is achieved

\(^1\)OASIS is a global consortium that drives the development, convergence and adoption of e-business and web service standards.
3.2. SERVICE-ORIENTED ARCHITECTURES

through the service descriptions, in which is the basic format that establishes the name of the service and the data expected and returned by the service. For services to interact and accomplish something meaningful, they must exchange information. In SOA, it is achieved through a communication framework called as *messaging*, which is responsible to preserve the loosely coupled relationship.

### 3.2.2 SOA Roles

It is important to state that SOA is not an architecture only about services, its loosely coupled relationship of services encompasses three kinds of roles (Papazoglou, 2003; Erl, 2005): *service provider*, *service discovery agency* and *the service client (or consumer)*. The interaction between the three participants involve the main operations that is publish, find and bind operations as depicted in Figure 3.2. Moreover, these roles and operations act upon the main services artifacts, such as the service description and the service implementation. The service provider defines a service description of the service and publishes it to a client or a service discovery agency, which made the service published and discoverable. The service client (or consumer) uses a find operation for searching and retrieve the service description from the service discovery agency, and uses such description to bind with the service provider and invoke the service or interact with its implementation (Papazoglou, 2003).

![Figure 3.2](image.png) 

*Figure 3.2* Service-Oriented Architecture Roles and Operations [Papazoglou, 2003].

There is a fourth role that is not depicted in Figure 3.2 related to the SOA contract (McGovern *et al.*, 2003). The main characteristic of such role is to define a contract between the service consumer and service provider. This contract specifies necessary information about the service, such as operations, pre- and post conditions, format of request and response from the service, and QoS (Quality of Service) information.
3.2.3 SOA Characteristics

SOA has a set of characteristics that makes it special type of software architecture. These characteristics can be described as follows (Brown et al., 2002; Erl, 2005; Dias, 2008):

**Discoverability.** Services need to be found not only by unique identity but also by interface identity and by services type (Erl, 2005).

**Interoperability.** It is the ability of systems to use different platforms and languages to communicate with each other. However, it is necessary to provide protocols that the roles of SOA can to interoperate.

**Composability.** Services can use or call other services. In order to develop and evolve composable services, SOA support automation of flexible and highly adaptative business process. A business process can be broken down into a series of services, each with well-defined activities responsible to execute pre-defined portions of the process.

**Extensibility.** When service logic is properly portioned via an appropriate level of interface granularity, the scope of functionality offered by a service can sometimes be extended without breaking the established interface (Erl, 2005).

**Interface-Based Design.** SOA has as the main characteristic its ability to separate the service’s implementation (“what”) from its interface (“how”). Thus, multiple services can implement a common interface and a service can implement multiple interfaces.

**Coarse-grained Interfaces.** In the context of service-oriented applications, granularity may occur in two ways. First, it may be applied to the scope of the domain that the entire service implements. Second, it is applied to the scope of the domain that each method within the interface implements.

**Loose Coupling.** The loose coupling is promoted through the well-know dependencies, contract and binding between service consumers and service providers (Dias, 2008).

These characteristics are responsible to determine the benefits in the SOA adoption. For example, most of the applications need to evolve and they must be adapted to continuously changes in the requirements. However, the architecture must support it in a flexible way in order to be reconfigured in accordance with the requirements. Another benefit relies on the reusability due the ability of SOA to compose new services out of existing ones, providing a major possibility for a rapidly reuse according to the demanding business needs (Dias, 2008).
3.3 Service-Oriented Product Lines

The main goal of SOA and SPL is that both approaches encourage an organization to reuse existing 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. On the one hand, SPL explores commonalities and variabilities among a set of related products. On the other hand, SOA enable orchestration, assembly and maintenance of enterprise solutions to quickly react to changing business requirements (Cohen and Krut, 2007). Reusing services can be improved by creating a product line that satisfies several variability requirements from different customer applications.

According to (Trujillo et al., 2007), applications interfaces are customized by using standards to consume supplied services, but also the application functionality is customized by using product lines of supplied services. As a consequence, it enables the possibility that an entire SOA application itself could require its customization. Hence, the SOA itself turns into a product line. This scenario requires that a product line consume products that are supplied from third-party product lines (Trujillo et al., 2007). Another possible scenario is in the case of including services within a product line architecture (Cohen and Krut, 2007; Günther and Berger, 2008). Thus, the product line encompasses its model, design and development in the context of a SOA.

In this approach, variabilities are included in the product line architecture following SOA concepts and implemented using services and components. For example, service can exist in high level of abstraction and components are in low level responsible for encapsulating business functionalities and exposed them through services. In this scenario, a specific configuration could select the component or service, depending on the specific functional or quality features needed by the application and satisfied by each alternate (Cohen and Krut, 2007). This configuration can be applied in the context of service technologies responsible to implement the services and components and variability mechanisms to customize specific and variable features. This scenario is depicted in Figure 3.3. The core services are the main reusable assets that can be configured to build customized products. It enables new dimensions of customizations, which includes support for dynamic variation, exploitation of existing services (when it is appropriate, and the time-to-market (i.e., the rapid development of product line systems). It is important to stress that the focus of this dissertation is in the case of the services within a product line.
3.4. SERVICE-ORIENTED DEVELOPMENT

3.4 Service-Oriented Development

Service-Oriented Architectures use services as the constructs to support the development of service-oriented applications. According to (Papazoglou et al., 2008), a well-constructed SOA empower a business environment with a flexible infrastructure and processing environment by provisioning independent, reusable automated business processes (as services), and providing a robust foundation for levering these services. The main basic concepts that allow the service-oriented development include the Component-Based Development (CBD) and business process modeling (Harmon, 2003). Moreover, Web Services are used as the largest and common technology through the flows of information among services, and components with the support of CBD for realizing service functionalities.

However, components and services present differences along the dimensions of the communication type, coupling type, and type of interface. One the one hand, services are subject to continuous maintenance and improvement, and its selection is usually done dynamically on the basis of a set of policies. On the other hand, components are distributed objects, deployed on middleware servers and does not allow for the same kind of reuse and dynamic behavior of services. However, components can contribute with Service-Orientation development with principles such as modularization, and separation of concerns (Papazoglou et al., 2008), providing a flexible way to encapsulate functionalities and expose them in terms of Web Services (or services in a technology independent way).

Nowadays, with the growth of the use of SOA in many organizations, new technologies have been emerged, such as Open Services Gateway Initiative (OSGi), Apache
Tuscany, Enterprise Java Beans (EJB) 3 with Web Service (WS), among others. These technologies implement the main SOA characteristics mentioned in the earlier subsection. For example, the interoperability characteristic is implemented by Web Services through XML-based standard protocols (Dias, 2008). In contrast, this characteristic is not provided by EJB 3 or OSGi technologies. However, with the both two technologies it is possible to modularize components and expose them as services, allowing loose coupling, composability, extensability, and interface-based design.

3.4.1 Service-Oriented Technologies

In a service-oriented context, considering orchestrating and designing applications, it is expected that it must be under SOA quality attributes, such as loose coupling, interface-based design, composability, and discoverability (Erl, 2005). However, this scenario increases with the management of variability in service-oriented product lines, since the development of services as core assets must be produced with a good modularity, well-defined interfaces and granularity concerns in variabilities identification. In this case, granularity concerns are services decomposition as possible variants under an approach that focus on its life cycle control. This approach relies on service technologies, which are useful to provide support for service implementation and to allow the configuration (e.g., addition, exclusion or replacement) of services at runtime.

Indeed, there are many service technologies that can fits well in this context, such as Open Services Gateway Initiative (OSGi), Apache Tuscany and JAX Web Services. Such technologies are generally coupled with a configuration environment to allow the adequate management (e.g., assembly, adaptation and connection) of core assets. Moreover, core assets configuration allows discoverability and composability for different customers or application at run-time. These characteristics are the core of the both technologies, which will be briefly described next.

**OSGi.** OSGi (Alliance, 2007) is an interface specification that defines an open, common architecture to develop, deploy and manage services in a coordinated fashion. OSGi is a non-proprietary software component framework for manufacturers, service providers and developers. The development using the OSGi platform encompass some APIs that controls the life cycle of applications, components and services running in a single Java Virtual Machine (JVM). Moreover, OSGi is a model that enables the capability of applications to dynamically discover and user services provided by other applications running inside the same OSGi platform (Alliance, 2007).
It is specific to Java and it requires a runtime container to manage the component and service registrations. This container has only the minimal features required by the components in order to execute and interact, thus, reducing the amount of unnecessary functionalities present in the product. The reason for using OSGi is that it is a suitable technology for implementing variation points, by providing an easy way to dynamically perform changes among different implementations (Almeida, 2007). In this scenario, new components and services can be included without the need to recompile the whole system, thus, facilitating the upgrading and replacement of single components.

In the current Release (R4), the OSGi software has the following layers (Alliance, 2007): Security Layer, it defines a secure packaging format as well as the runtime interaction with the Java security layer; Module Layer is responsible for the management of the units of modularization, called bundle. A bundle is comprised of java classes and other resources, e.g., database configurations and spring dynamic modules (Spring, 2008), which together can provide functions to end users; Lifecycle Layer, it provides an API to control the security and lifecycle operations of bundles; and Service Layer, which is responsible for registering services, searching for them, or receiving notifications when their registration state changes (Alliance, 2007).

Apache Tuscany. The Apache Tuscany is an open source project at the Apache Software Foundation that aims to develop SOA solutions by providing an infrastructure for its development and management. It provides a lightweight infrastructure that implements Service Component Architecture (SCA) (SCA, 2006), which has the capability for composing applications from business services resulting in an assembly model. The assembly model defines an eXtensible Markup Language (XML) language for assembling components into applications. SCA promotes the benefits of constructing large and complex enterprise systems out of well-defined and sometimes replaceable component parts called services.

A SCA service provides reusable pieces of business function and has a well-defined interface which identifies how it can be called to provide a particular function. The reason for using Apache Tuscany is that the use of such assembly model can be suitable for selecting components or services depending on specific features selected for distinct SPL applications. In a running SCA application, the component assembly is described by using XML construct wherein every SCA service has a name and an interface. The interface defines the operations that the service provides. Multiple services can define the external interface of a component. The business logic of a service is provided by a
Moreover, SCA components call a service by using a reference and the component implementation is given access to references in an implementation language specific way. The connection between the reference and the service is called a wire. References are wired to services in which are connected with components and described within a composite application.

**Java Web Services.** Java API for XML Web Services (JAX-WS) (JAX-WS, 2009) is a new programming model that simplifies application development through support of a standard, annotation-based model to develop Web Service applications and clients. The JAX-WS specification aligns itself with the current industry trend towards a more document-centric messaging model and replaces the remote procedure call programming model as defined by Java API for XML Remote Protocol Control (JAX-RPC) (JAX-RPC, 2009).

JAX-WS introduces support for annotating Java classes with meta-data to indicate that such class is a Web service. Using annotations within the Java source code and within the Java class simplifies development of Web services by defining some of the additional information that is typically obtained from deployment descriptor files (e.g., Web Services Description Language (WSDL) files), or mapping meta-data from XML and WSDL files into the source artifacts. The reason for using JAX-WS is that such flexibility for developing JAX Web services through annotations and the delegation model can enable variability mechanisms to design customized service-based applications.

The adoption of SPL principles within JAX-WS helps software engineers by providing a systematic way to create customized service-oriented applications, for example, selecting interfaces of feature-based services developed with JAX-WS dynamically. The implementation of JAX-WS provides support for document-oriented Web Services, asynchronous services, and services that use transports other than the HTTP protocol, such as REST (Fielding and Taylor, 2000). It provides consumer and provider interfaces for dealing directly with SOAP messages or REST messages.

### 3.5 Chapter Summary

This chapter discussed the concepts of service-oriented development, where enterprise solutions are generally viewed as a set of services connected via well-defined contracts that define services interface in the context of Service-Oriented Computing (SOC) and
Service-Oriented Architecture (SOA). SOC is responsible to develop software systems in a distributed way, including its own design paradigm and design principles, design pattern catalogs, pattern languages, a distinct architectural model (e.g., service-oriented architectures) and services technologies. Moreover, the basic concepts related to the combination of service-orientation and software product lines were also covered, since it is the focus of this dissertation.

Next chapter will present a case study using the technologies described in this chapter, and product line variability mechanisms implementation in the context of service-oriented product lines.
4

An Assessment on Technologies to Implement Core Assets in Service-Oriented Product Lines

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.

—DOUGLAS ADAMS

4.1 Introduction

Current research (Cohen and Krut, 2007; Segura et al., 2007; Günther and Berger, 2008; Smith and Lewis, 2009) has explored potential synergies between Service-Orientation (SO) and Software Product Lines (SPL), in particular how variability can be implemented in service-oriented product lines. However, these work focus on services as core assets in SPL in which the services handle variabilities, and they adress the relationship between SO and SPL from a conceptual level but to not go into details about the implementation. An important factor when implementing services as core assets in SPL is the analysis of existing variability mechanisms (see Chapter 2) to manage variation points exposed as service functionalities according to binding times (e.g., runtime).

However, the dynamicit of service-oriented applications implies on the services lifecycle control, since many of them are independent and need to be discovered at runtime throught the network (Erl, 2005). In this way, a service technology (see Chapter 3) may
be important to manage the services life-cycle and to map the SOA concepts down to the level of SPL practices. Thus, the technology used for implementation should be analyzed as well as it can require additional technological-specific concerns that may cause different measurable benefits, e.g., complexity, modularity and understandability. For example, when services candidates handle specific variants (depending on the feature type (see Chapter 2)) it may affect how the service consumer will support each variant, and must prepare the change according to the system requirements.

In order to understand and mitigate these issues, this chapter provides a quantitative assessment through a case study for comparing service technologies with the support of variability mechanisms to implement core assets in a service-oriented product lines project. According to the results of the assessment, we have constructed an initial decision model. In this sense, given a variability mechanism and some criteria (e.g., modularity, complexity, understandability, and stability), the decision model is able to indicate which technology is suitable for supporting, and consequently, implementing them.

In order to conduct the assessment, we organized the chapter following the concepts proposed by (Kitchenham et al., 1995; Kitchenham and Pickard, 1998b; Wohlin et al., 2000). Thus, Section 4.2 presents the case study definition, which defines the objectives, hypotheses, and the pilot project used in the study. In Section 4.3 the planning takes place. Section 4.4 shows the analysis and the interpretation of the results. Section 4.5 presents the decision model based on the results generated by the analysis. Finally, Section 4.6 summarizes this Chapter.

4.2 The Definition

In order to define the assessment, it is important to state the context where it should be applied, as well as the hypotheses that must be detailed enough to make clear what measurements are needed to demonstrate the expected effect according to the context, and the pilot project, which defines the domain used in the study.

4.2.1 Criteria for the Assessment

This subsection presents the criteria that will be useful to define the case study, allowing the collection and interpretation of the data in the context of service-oriented product lines projects. The idea behind service-oriented product lines is to develop and consume reusable service modules from a common base and map its variations. In the context of service-orientation, the central purpose is to achieve improved SoC through the
decomposition of modules into services in a high-level and components in a low-level, affecting attributes such as instability, coupling, cohesion, and size.

We believe that such scenario could be significantly enhanced when a service technology is properly tailored to a domain where, culminating with activities that focus in providing and consuming reusable services. Furthermore, this scenario increases when SPL takes place with the definition and mapping of variability within the service technology. Consequently, the use of variability implementation mechanisms becomes unavoidable, making the decisions on the implementation more complex.

However, such complexity in addressing variability implementation mechanisms within a technology has a positive side, since it may modularize critical concerns as services and components, reducing the replication of code, the size of the system, and making concerns more stable to be consumed by others. On the other hand, the inappropriate use of a service technology and variability implementation mechanisms together may affect negatively under these attributes and increase the complexity of the application domain, making the assets hard to maintain and reuse. According to (Briand et al., 1995; Sant’anna et al., 2003), software metrics are the most effective way to supply empirical evidence that may improve the understanding of different dimensions of the software reusability and maintainability.

The metrics are more effective when they are associated with some measurement framework, allowing software engineers to understand and interpret the meanings of the collected data (Sant’anna et al., 2003). In this sense, creating a metric suite is important to capture information about the code generated in service-oriented product lines projects in terms of attributes such as size, coupling, cohesion, instability and SoC. This work defines a suite that reuses and refines classical and current metrics defined in (McCabe, 1976; Chidamber and Kemerer, 1994; Martin, 1994; Basili et al., 1996; Sant’anna et al., 2003; Castor et al., 2006; Eaddy et al., 2007; Almeida, 2007; Figueiredo et al., 2008; Ribeiro, 2008; Matos, 2008; Quynh and Thang, 2009). The criteria for the selection and refinement of these metrics were based on theoretical and practical demands and will be described in subsection 4.2.2. In addition, they were chosen and refined to reflect new abstractions introduced by the combination of SOA and SPL.

Our suite is composed of six metrics grouped according to the attributes they measure, for example: size, coupling, cohesion, instability, and SoC. Furthermore, the description of each metric emphasizes how it satisfies the measurement requirements defined. According to (Sant’anna et al., 2003), measuring particular internal attributes, such as cohesion, is useful if it is related to a measure of some external attribute of the object of
study, for example, reusability or maintainability. In this sense, we developed a measurement framework to capture the understanding of the size, coupling, cohesion, instability, and separation of concerns attributes in terms of their usefulness as predictors of the maintainability and reusability qualities. Figure 4.1 depicts the measurement framework defined for this study’s purposes.

Figure 4.1 Measurement Framework.

As it can be seen in Figure 4.1, the framework is composed of three different elements: *qualities*, *factors* and *attributes*. The qualities are the attributes that must be observable in the system (*reusability* and *maintainability*). The factors (*complexity, understandability, stability* and *modularity*) are secondary quality attributes that influence the primary qualities. The attributes are related to well-established software engineering principles, which are essential in SOA and SPL development.
4.2. THE DEFINITION

4.2.2 Context

The case study should be conducted in accordance with a specific goal, a set of questions that represent the operational definition of the goal, and the related metrics the must be collected to aid answering the questions (Kitchenham and Pickard, 1998b). In this sense, the Goal Question Metric (GQM) method (Basili et al., 1994) was used to define the goal of the case study and to derive the questions that should be answered in order to determine whether the goal was achieved.

**Goal.** Analyze technologies (OSGi, Apache Tuscany and JAX-WS) to implement core assets for the purpose of evaluation with respect to reusability, maintainability, complexity, stability, modularity, and understandability from the point of view of the software engineers and researchers in the context of service-oriented product lines projects.

**Question 1.** How complex are the services and components developed using a particular technology? Structural dependencies between services and components based on provider/consumer relationships have considerable influence on system complexity. This scenario goes deeper when there may be variabilities encapsulated as services or in its operations and components. In this context, the following metrics were defined to help answering this question:

- **M1: Weighted Operations per Component or Service (WOCS).** The number of methods and their complexity are a measure of how much time and effort are required to develop and maintain a service or component. Thus, the Weighted Operations per Component or Service (WOCS) will measure the complexity of the service or component, in terms of its operations (methods) that will be required by other services or components. Hence, consider a Component or Service $C$ with operations $O_1, \ldots, O_n$. Let $c_1, \ldots, c_n$ be the complexity of the operations. Then:

$$WOCS = c_1 + \ldots + c_n$$

The metric indicates that operations with many formal parameters are more likely to be complex than operations in which require less parameter. In this sense, the complexity of operation $O_k$, $c_k$ can be defined as follows: $c_k = \alpha_k + 1$, where $\alpha_k$ denotes the number of formal parameters of $O_k$. According to (Chidamber and Kemerer, 1994), a value lower than 10 is considered acceptable for a simple application.
4.2. THE DEFINITION

• **M2: Cyclomatic Complexity (CC).** The Cyclomatic Complexity (CC) measure is a mathematical technique developed by McCabe Mccabe (1976) and was adapted for this study’s purposes. In this context, it will provide a quantitative basis in order to identify services or components that will be difficult to test or maintain. Cyclomatic Complexity is measured in terms of the number of paths (e.g., loops and conditional statements) through a program, and can be useful to keep the size of the services and components manageable and allow the testing of all the independent paths. This measure is calculated from a connected graph of the services or component as follows:

\[
CC = E - N + p,
\]

where:

\(E\) = 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), this metric indicates that a unit with CC between 1 and 10, it is a simple program, without much risk. A value between 11 and 20 represents a more complex program, with moderate risk. Values ranging between 21 and 50 represent a complex program, with high risk. Finally, a complexity greater than 50 is an untestable program and presents a high risk.

**Question 2. How stable are the services and components developed using a particular technology?** A single change may begin a cascade of changes of independent services or components making the design and development fragile, and difficult to reuse. In order to answer this question, we define one metric:

• **M3: Instability Metric for Service or Component (IMSC).** This metric is based on the metrics, e.g. **fan.in** and **fan.out**, reported by the Metric Advisor from C-DAC (CDAC, 2009). The **fan.in** metric of function **A** is calculated by the number of functions that call to function **A**, and **fan.out** metric is the number of functions that are called by function **A**. Both **fan.in** and **fan.out** are used to evaluate software’s maintainability. Thus, the cost of maintain for a function, which has high **fan.out** value is very high. On the other hand, the **fan.in** metric’s value of function is high means that there are many functions that use and depend it. In this sense, (Perepletchikov et al., 2007) defined a metric in order to measure the interaction between a service or component through sending and receiving messages.
Suppose a service $S$, the $fan.in$ metric can be calculated by the number of messages which are sent to service or component $S$ and the $fan.out$ metric is the number of messages which are sent by the service or component $S$. In order to measure this, Perepletchikov et al. (2007) proposed the following formula for calculate the instability of a service:

$$IMSC = \frac{fan.out}{fan.in + fan.out} \times 100\%$$

If the value of the instability is low, it affects the level of the service dependency whereas others depend on it higher (Perepletchikov et al., 2007). Thus, according to (Perepletchikov et al., 2007), $IMCS = 0$ means that the stability of the service is very high. On the other hand, $IMCS = 1$ means that the service is very instable, difficult the maintenance.

**Question 3.** Do the technologies produce services and components with low coupling? It is important to measure the extent to which interdependencies exist between services and components. The design and development of a SOA application with lower coupling among services and components indicates a better design in terms of modularity. A well-modularized design and development of a SOA application brings forth potential benefits in multiple aspects, such as acceleration of development, reduction of maintenance cost, as well as the enhanced flexibility and reusability. We define one metric to understand this question:

- **M4: Coupling Between Components or Services (CBCS).** This metric is calculated based on the number of relationships between one service $A$, for example, and other services in an application. It can be mathematically described as follows:

$$CBCS = \sum_{i\neq j=1...n} A_i B_j$$

In which: $n$ is the number of services in application; $A_i B_j = 0$ if $A_i$ does not connect to $B_j$ and $A_i B_j = 1$ if $A_i$ connects to $B_j$. For a service $A$, the larger the value of Coupling between Components and Services (CBCS) metric, the tighter the relationship with other services is. In other words, service $A$ depends much more on others (Quynh and Thang, 2009). This metrics is based on the Coupling between Objects (CBO) metric from CK metrics (Chidamber and Kemerer, 1994) and Coupling between Components (CBC) defined in (Sant’anna et al., 2003;
Ribeiro, 2008; Matos, 2008) work. As can be seen in these works, a CBCS value greater than 6 is considered an acceptable value for a simple application.

**Question 4.** Do the technologies produce services and components with high cohesion? In the context of SOA design and development, low cohesion between services or components decreases software modularity, thereby increasing the probability of errors during the development process. In order to understand and help answering this question, we define the following metric:

- **M5: Lack of Cohesion over Operations (LCOO).** According to (Sant’anna et al., 2003), this metric can be measured by counting the number of pairs of operations that share attributes, minus the number of pairs of operations that do not share any attribute. In the context of this work, as services and components are usually used since it is used in service-oriented applications this metric will be adapted to support it. Thus, the terms services and components will be used in terms of modules of a system. This metric can be represented mathematically as follows:

\[ LCOO = |P| - |Q|, \]

\[ \text{if } |P| > |Q|, \text{LCOO} = 0, \text{otherwise}. \]

A high Lack of Cohesion Over Operations (LCOO) value could indicate that the implementation of the component or service is poor (i.e., it might be worthwhile to split both into two or more modules). This metric can be exemplified according to the following Java code snippet. In this example, there are two pairs of methods that do not access any attribute in common (methodA, methodB) and (methodA, methodC), as just a couple of methods (methodB, methodC) shares the sd attribute. Therefore, the LCOO measure value for ServiceAImpl is 1, since: \(|2| - |1| = 1.\)

```java
public class ServiceAImpl {
    private IServiceB sb;
    private IServiceC sc;
}
4.2. THE DEFINITION

```java
private IServiceD sd;
private IServiceE se;

public void methodA() { [... uses sb, sc [...])
public void methodB() { [... uses sd [...])
public void methodC() { [... uses sd, se [...])
```

This metric is based on the Lack of Cohesion over Methods (LCOM) metric from C & K metrics (Chidamber and Kemerer, 1994) and Lack of Cohesion Over Operations (LCOO) used in (Sant'anna et al., 2003; Figueiredo et al., 2008; Ribeiro, 2008; Matos, 2008) works. According to (Chidamber and Kemerer, 1994), a LCOO value lower than 10 implies in a low cohesion.

Question 5. How easy is to separate concerns using a particular technology?

The decomposition of SOA SPL applications into services and components may allow the SoC in specific self-contained and replaceable building blocks (Medeiros et al., 2009). In order to understand and answering it, the following metric will be used:

- **M6: Concern Diffusion over Components or Services (CDCS).** This metric was adapted to the context of service-oriented product lines context using services and components to implement possible concerns identified. Thus, the Concern Diffusion over Components and Service (CDCS) will count the number of services or components whose main purpose is to contribute to the implementation of concern. Furthermore, it counts the number of services or components that are being used in attribute declarations, formal parameters, return types, throws declarations and local variables (Sant’anna et al., 2003). A high value for CDCS metric indicates that a concern implementation might be scattered. Figure 4.2 shows a code snippet that was implemented in this study with respect to the Authentication service, in particular, the Authentication by Password concern. The main purpose of the Authentication abstract class, the PasswordAuthentication class and the authentication spring bean is to contribute for the concern implementation. Hence, they should be counted in the CDCS metric. Hence, the metric value is $C_{CDCS_{PasswordAuthentication}} = 3$. It is important to highlight that we did not found standardized values for the CDCS metric. Thus, we will consider the range between 3 and 5 as acceptable values for a simple application.
4.2. THE DEFINITION

4.2.3 Hypotheses

Defining the hypotheses correctly is both the most difficult and the most important part of defining a case study. If the hypotheses are wrong, the case study will not produce useful results (Kitchenham et al., 1995). According to the case study, it was established the following null hypotheses and alternative ones. According to (Wohlin et al., 2000), the null hypotheses are those who the experimenter wants to reject, while the alternative hypotheses are those that the experimenter wants to confirm.

Null hypotheses. In this study, the null hypotheses state that there is no difference among the technologies for implementing variabilities in service-oriented product lines projects.
4.2. THE DEFINITION

\[ \begin{align*}
H_0 :  & \quad \mu_{\text{Tuscany WOCS}} = \mu_{\text{JAX–WS WOCS}} = \mu_{\text{OSGi WOCS}} \\
       & \mu_{\text{Tuscany CC}} = \mu_{\text{JAX–WS CC}} = \mu_{\text{OSGi CC}} \\
       & \mu_{\text{Tuscany IMSC}} = \mu_{\text{JAX–WS IMSC}} = \mu_{\text{OSGi IMSC}} \\
       & \mu_{\text{Tuscany CBCS}} = \mu_{\text{JAX–WS CBCS}} = \mu_{\text{OSGi CBCS}} \\
       & \mu_{\text{Tuscany LCOO}} = \mu_{\text{JAX–WS LCOO}} = \mu_{\text{OSGi LCOO}} \\
       & \mu_{\text{Tuscany CDCS}} = \mu_{\text{JAX–WS CDCS}} = \mu_{\text{OSGi CDCS}}
\end{align*} \]

**Alternative Hypotheses.** In this study, the alternative hypotheses state that there are differences among the technologies for implementing variabilities in service-oriented product lines projects.

\[ \begin{align*}
H_1 :  & \quad \mu_{\text{Tuscany WOCS}} \neq \mu_{\text{JAX–WS WOCS}} \neq \mu_{\text{OSGi WOCS}} \\
       & \mu_{\text{Tuscany CC}} \neq \mu_{\text{JAX–WS CC}} \neq \mu_{\text{OSGi CC}} \\
       & \mu_{\text{Tuscany IMSC}} \neq \mu_{\text{JAX–WS IMSC}} \neq \mu_{\text{OSGi IMSC}} \\
       & \mu_{\text{Tuscany CBCS}} \neq \mu_{\text{JAX–WS CBCS}} \neq \mu_{\text{OSGi CBCS}} \\
       & \mu_{\text{Tuscany LCOO}} \neq \mu_{\text{JAX–WS LCOO}} \neq \mu_{\text{OSGi LCOO}} \\
       & \mu_{\text{Tuscany CDCS}} \neq \mu_{\text{JAX–WS CDCS}} \neq \mu_{\text{OSGi CDCS}}
\end{align*} \]

4.2.4 Pilot Project

The case study presented was based on the development of a SPL in the domain of libraries, herein named the Rental for All Software Factory (R4ll). The R4ll was developed as part of a postgraduate course at the Federal University of Pernambuco, Brazil. The SPL project development involved all life cycle phases ranging from scoping to testing concerns. For validation purposes, the reference architecture and the feature model of R4ll were analyzed and adapted in order to support SOA concepts. The feature model following the notation proposed by (Kang et al., 1990) was defined for capturing commonalities and variabilities of the domain, which resulted in fifty-nine features, and sixteen variabilities could be identified. For a more detailed explanation about the pilot project, see Appendix A. For the complete artifacts generated during the R4ll definition, such as the domain feature model, services and components diagrams, class entities diagrams, sequence diagrams and the source code, see the case study hosting site.  

44
4.3 The Planning

After the definition of the case study, the planning was started. The case study was conducted in off-line (not industrial software development) in the context of a service-oriented product line project.

Subjects. Only one subject was chosen to perform the case study, and he is a master student (the dissertation author) at the Informatics Center in the Federal University of Pernambuco Informatics Center - Federal University of Pernambuco (CIN-UFPE). The decision to choose only one subject is further detailed in the internal validity section.

Training requirements. It is important to highlight that the subject had no knowledge about OSGi and Apache Tuscany technologies usage. In this way, initially, the subject read the technologies specifications. Next, examples were implemented, and other sources, such as papers and technical reports were considered to complement the study. Moreover, the subject had previous experience in the development of one software product lines project. Thus, it was assumed that the subject was already familiarized with concepts related to variability and domain engineering, then, facilitating the understanding of the domain without the need of training.

Quantitative analysis mechanism. The quantitative analysis was divided as complexity, modularity, stability, and understandability for the implementation steps. Each of the analyses sections will describe the analysis concerning the technologies according to the variability mechanisms. It is important to highlight that these quality factors are predictors for reusability and maintainability, and thus it will be analyzed as well. Furthermore, it will also use descriptive statistics, such as mean, maximum, minimum, standard deviation, and bar charts to analyze the data generated by the case study. Regarding the hypotheses defined for the case study, it will be used the ANalysis Of VAriance (ANOVA) test (Wohlin et al., 2000). The ANOVA can be used to analyze case studies (or experiments) from a number of different designs. In this case study, the ANOVA was applied with its simplest form, i.e., the one factor for different treatments.

Case study design. In order to provide more quality in what will be assessed, we decide to divide the assessment into three treatments: (i) the first stage encompassed the implementation of the R4II artifacts (service modules, components, and classes) using Apache Tuscany as the main technology and three variability mechanisms (Binding
variability within services, Parameters, and Strategy Design Pattern); (ii) in the second stage, the subject implemented the R4ll artifacts using JAX-WS technology and the three variability mechanisms aforementioned; and, (iii) in the third and last stage, the R4ll artifacts were implemented using OSGi.

**Data collection procedures.** The case study was performed during March-July 2009, at the Federal University of Pernambuco, Brazil. The study was composed of one subject and the project was developed around 580 hours and 45 minutes, including all the implementation. The project was implemented with the support of three technologies (OSGi, Apache Tuscany and JAX-WS), and for each technology, three variability mechanisms were applied (e.g., *Binding variability within services, Parameters and Strategy Design Pattern*) in order to implement three feature types (Alternative, Optional and OR type) defined in the R4ll feature model. Table 4.1 summarizes the total data (e.g., *number of services, number of components, packages, number of classes, and the total lines of code*) for each technology with the respective variability mechanism according to a feature type.

For each of the technologies implementation, several metrics data were collected, such as *complexity, modularity, stability, and understandability* metrics. The metrics data were collected using the following tools: metrics (Metrics, 2003) and Borland Together (Together, 2008). For the SoC metric, the collection was made following the guidelines proposed by (Eaddy *et al.*, 2007).

**Table 4.1  Summary of the technologies implementation.**

<table>
<thead>
<tr>
<th>Technology</th>
<th>Variability</th>
<th>LOC</th>
<th>Packages</th>
<th>Classes</th>
<th>Interfaces</th>
<th>Components</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mechanism</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OSGi</td>
<td>Binding</td>
<td>1,758</td>
<td>22</td>
<td>44</td>
<td>18</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>variants</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td>1,772</td>
<td>16</td>
<td>43</td>
<td>18</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Strategy</td>
<td>1,656</td>
<td>16</td>
<td>38</td>
<td>17</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Pattern</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache</td>
<td>Binding</td>
<td>2,322</td>
<td>24</td>
<td>50</td>
<td>14</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Tuscany</td>
<td>variants</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td>2,246</td>
<td>19</td>
<td>45</td>
<td>15</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Strategy</td>
<td>2,103</td>
<td>17</td>
<td>41</td>
<td>15</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Pattern</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAX-WS</td>
<td>Binding</td>
<td>2,243</td>
<td>22</td>
<td>47</td>
<td>15</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>variants</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td>2,252</td>
<td>17</td>
<td>42</td>
<td>15</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Strategy</td>
<td>2,161</td>
<td>16</td>
<td>39</td>
<td>15</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Pattern</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Internal validity.** This case study has five dimensions of expertise, such as *software product lines, service-orientation concepts, variability implementation mecha-
nisms, the combination between service-orientation and software product lines, and R4ll domain. Present knowledge in these areas might takes considerable time, one year at least. Consequently, as the deadline to perform the case study was shorter, the subject team formation and training required could take quite some effort and time. In this sense, we chose to use only one subject with considerable expertise in service-orientation, web services, variability management and software product lines. This might lead with mistreatments with the data collected, since different acquirements are not considered.

Nevertheless, the participation of the subject in the R4ll development is fundamental to justify this case study execution with a single subject. Furthermore, it is important to state and we encourage that this case study might be replicated using subject teams providing thus more accurate data. Moreover, the small domain considered may have also influence in the internal validity, since the variabilities considered may not have functional value with service-oriented product lines concepts. However, trying to solve these possible issues, critical variations were chosen, and, as the subject have been participated on the entire R4ll development life cycle it is acceptable that it might be controlled.

**External validity.** According to (Wohlin et al., 2000), external variability is concerned with the generalization of the experiment (or a case study) results. Due to time constraints, it will not possible to apply the study in others research groups. Nevertheless, the external validity of the study is considerable sufficient, since it aims to select suitable technology that fits well in the context of service-oriented product lines. Therefore, it may be used as baseline for comparison in further studies developed in the same context of this one.

**Conclusion validity.** According to (Wohlin et al., 2000), the conclusion validity determines the capability of the study to generate conclusions. In this sense, the analysis and interpretation of the results of this case study will be described using descriptive statistics, which are appropriate to the data type collected.

**Validity threats.** The following validity threats related for this case study were identified:

- **Boredom.** As the case study will compare three technologies, and the subject must implement for each technology a certain quantity of code concerning also three different variability implementation mechanisms, thus it might be a somewhat repetitive work in some cases. In this way, the subject may feel upset or disappointed with the case study. This situation led to define the case study desing in three treatments. In order to reduce the boredom of the case study, the subject will
have a period of five days of rest before to execute the next treatment.

- **Lack of historical data in the field.** Although some work in the literature had covered the implementation of variability in service-oriented product lines projects, none of them has performed a case study involving a subject on the development of a service-oriented product line using service technologies in the implementation phase, as in the case of this study. Thus, we could not compare our results with a baseline in the context of a service-oriented product line.

- **Subject knowledge.** The results of the case study may be compromised by the lack of knowledge by the subject concerning to the technologies of the selected project. As a consequence, if the subject has not any prior knowledge of technologies, the implementation will take a considerable time and effort.

### 4.4 The Analysis

This section involves analyzing the data collected after the implementation. The output of the analysis is a decision model for recommending technologies for implementing core assets in service-oriented product lines projects. The following subsections were organized based on the quantitative analysis mechanisms defined in the planning phase. In this sense, descriptive statistics will be applied, as well as the hypotheses testing through the ANOVA analysis. The hypotheses testing will be useful to verify suitable(s) technology(ies) according to the discussed and analyzed criteria, helping thus the built of our decision model.

#### 4.4.1 Complexity Analysis

This section presents the discussion with respect to the complexity criterion results. As depicted in Figure 4.1, the complexity criterion is related with two measures, such as *Weighted Operations per Component or Service (WOCS)*, and *Cyclomatic Complexity (CC)*. Thus, the descriptive analysis will be detailed for each measure as follows.

**Descriptive Statistics Analysis.** Figure 4.3 show the complexity data, respectively, for OSGi (4.3(a)), Apache Tuscany (4.3(b)), and JAX-WS graphically (4.3(c)). As it can be seen in both figures, in the *Binding variants within services mechanism* analysis, the WOCS values are higher for the Apache Tuscany, because it was necessary to define a method to instantiate the *UserService*, and for OSGi and JAX-WS it was not necessary. However, the values are also higher for JAX-WS because it was important to
define `onCycleDetected` methods in the `Item` and `Category` variants to avoid the circular reference issue.

In the context of the `Parameters` mechanism, a Factory Method Pattern (Gamma et al., 1995) was defined to instantiate the correct feature, which is reflected for the higher values for CC and WOCS concerning to Apache Tuscany and JAX-WS. It happens because extra operations for instantiate the `PenaltyForDelay` and `PenaltyForViolation` variants, processed by the web services. In the context of the Strategy Pattern mechanism, the values of OSGi for WOC metric is lower when compared to the Apache Tuscany and JAX-WS. In the OSGi implementation, the `Fine` variant defines only four methods that will be overridden by the concrete classes. On the other hand, Apache Tuscany and JAX-WS define six methods.

In order to summarize the complexity values, we defined statistical values. These values are detailed in Table 4.2. These values are important in order to understand the ANOVA analysis and conduct our decision to build the decision model.

<table>
<thead>
<tr>
<th>Table 4.2 Results for the Complexity Analyses.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
</tr>
<tr>
<td>OSGI</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>Strategy Design Pattern</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Apache Tuscany</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Strategy Design Pattern</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>JAX-WS</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Strategy Design Pattern</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Analysis of Variance (ANOVA). Table 4.3 summarizes the results of the ANOVA test for the CC and WOCS measures. In the context of the ANOVA analysis, it is important to verify if the treatments have different mean values. Thus, we have to observe the $F_0$ and $F_{CRITIC}$ values. If $F_0$ and $F_{CRITIC}$, the null hypotheses are rejected,
4.4. THE ANALYSIS

(a) OSGi

(b) Apache Tuscany

(c) JAX-WS

Figure 4.3 Complexity Analysis.
concluding thus that have differences in the means among the treatments. Hence, as
can be seen in the table, the ANOVA did not reject the null hypothesis concerning both
**Binding variants within services, Parameters and Strategy Pattern** mechanisms ($F_0 < F_{CRITIC}$) for CC. In this sense, it means that there is no difference among the technologies. However, they can be classified in different priority levels, according to the points discussed in the descriptive statistics analysis. It is important to state these assumptions, since it will help in the construction of our decision model. For WOCS (Table 4.4), the hypotheses cannot be rejected, since $F_0 < F_{CRITIC}$.

### Table 4.3 ANOVA applied with level of significance of 5% to cyclomatic complexity collected data.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Cyclomatic Complexity</th>
<th></th>
<th>p-value</th>
<th>$F_{CRITIC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binding variants within services</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Between treatment</td>
<td>68.32941</td>
<td>2</td>
<td>34.16471</td>
<td>0.814554</td>
</tr>
<tr>
<td>Error</td>
<td>587.2</td>
<td>14</td>
<td>41.94266</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>655.5594</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Between treatment</td>
<td>46.50375</td>
<td>2</td>
<td>23.28137</td>
<td>0.774264</td>
</tr>
<tr>
<td>Error</td>
<td>420.0657</td>
<td>14</td>
<td>30.06905</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>467.5594</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Strategy Design Pattern</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Between treatment</td>
<td>49.71569</td>
<td>2</td>
<td>24.85784</td>
<td>0.701397</td>
</tr>
<tr>
<td>Error</td>
<td>496.1667</td>
<td>14</td>
<td>35.44048</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>545.8824</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4.4 ANOVA applied with level of significance of 5% to weighted operations per component or service collected data.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Weighted Operations per Component or Service</th>
<th></th>
<th>p-value</th>
<th>$F_{CRITIC}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binding variants within services</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Between treatment</td>
<td>62.92549</td>
<td>2</td>
<td>31.46275</td>
<td>0.476639</td>
</tr>
<tr>
<td>Error</td>
<td>924.1333</td>
<td>14</td>
<td>66.00952</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>987.0588</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Between treatment</td>
<td>36.89804</td>
<td>2</td>
<td>18.44902</td>
<td>0.320906</td>
</tr>
<tr>
<td>Error</td>
<td>804.8667</td>
<td>14</td>
<td>57.49048</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>841.7654</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Strategy Design Pattern</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Between treatment</td>
<td>53.71569</td>
<td>2</td>
<td>26.85784</td>
<td>0.368577</td>
</tr>
<tr>
<td>Error</td>
<td>1020.167</td>
<td>14</td>
<td>72.86905</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>1073.882</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.2 Stability Analysis

As can be seen in Figure 4.1, the stability criterion is related with the **Instability Metric for Service or Component (IMSC)**. Thus, the analysis will cover the behavior of such
4.4. THE ANALYSIS

measure in the data collected for the implementation of the artifacts.

**Descriptive Statistics Analysis.** Figure 4.4 presents the stability data, respectively, for OSGi (4.4(a)), Apache Tuscany (4.4(b)) and JAX-WS (4.4(c)), graphically. As can be seen in the both figures, the OSGi values are higher and a little close for Apache Tuscany. With respect to the Apache Tuscany it happens because it must exists a method for instantiating the UserService correctly. In the OSGi implementation side, it must be defined the bundle context and additional code for variants discoverabilty. Consequently, due to the dynamic behavior of OSGi, we cannot separate the persistence from the Bundle, creating a high coupling among them. This impact was reflected in the values for the Binding variants within services mechanism. Moreover, such condition is reflected also in the two other variability mechanisms (Parameters and Strategy Pattern).

Table 4.5 summarizes the stability values that were defined based on the statistical values.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Variability Mechanism</th>
<th>Measure</th>
<th>Statistic</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mean</td>
<td>Maximum</td>
<td>Minimum</td>
<td>Standard Deviation</td>
</tr>
<tr>
<td>OSGI</td>
<td>Binding variants within services</td>
<td>IMSC</td>
<td>0.517</td>
<td>0.601</td>
<td>0.489</td>
<td>0.056</td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td>IMSC</td>
<td>0.517</td>
<td>1.000</td>
<td>0.489</td>
<td>0.213</td>
</tr>
<tr>
<td></td>
<td>Strategy Design Pattern</td>
<td>IMSC</td>
<td>0.575</td>
<td>0.683</td>
<td>0.483</td>
<td>0.091</td>
</tr>
<tr>
<td>Apache Tuscany</td>
<td>Binding variants within services</td>
<td>IMSC</td>
<td>0.425</td>
<td>0.667</td>
<td>0.133</td>
<td>0.227</td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td>IMSC</td>
<td>0.380</td>
<td>0.750</td>
<td>0.080</td>
<td>0.248</td>
</tr>
<tr>
<td></td>
<td>Strategy Design Pattern</td>
<td>IMSC</td>
<td>0.342</td>
<td>0.375</td>
<td>0.125</td>
<td>0.334</td>
</tr>
<tr>
<td>JAX-WS</td>
<td>Binding variants within services</td>
<td>IMSC</td>
<td>0.255</td>
<td>0.333</td>
<td>0.125</td>
<td>0.070</td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td>IMSC</td>
<td>0.352</td>
<td>0.552</td>
<td>0.133</td>
<td>0.178</td>
</tr>
<tr>
<td></td>
<td>Strategy Design Pattern</td>
<td>IMSC</td>
<td>0.286</td>
<td>0.667</td>
<td>0.133</td>
<td>0.200</td>
</tr>
</tbody>
</table>

**Analysis of Variance (ANOVA).** As can be seen in Table 4.6, the ANOVA test did not reject the null hypotheses concerning Parameters and Strategy Pattern mechanisms ($F_0 < F_{CRITIC}$). On the other hand, it rejects the null hypothesis concerning the Binding variants within services mechanism ($F_0 > F_{CRITIC}$). In the context of the null hypothesis rejected, JAX-WS is a well suitable technology. In contrast, OSGi was the technology with the highest value followed, respectively, by Apache Tuscany.
4.4. THE ANALYSIS

(a) OSGi

(b) Apache Tuscany

(c) JAX-WS

Figure 4.4 Stability Analysis.
Table 4.6 ANOVA applied with level of significance of 5% to instability metric for service or component collected data.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Instability Metric for Service or Component</th>
<th>( F )</th>
<th>( p )-value</th>
<th>( F_{CRIT} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding variants within services</td>
<td>Between treatment</td>
<td>0.198853</td>
<td>2</td>
<td>0.059932</td>
</tr>
<tr>
<td>Error</td>
<td>0.295894</td>
<td>14</td>
<td>0.021135</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>0.494757</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>Between treatment</td>
<td>0.273105</td>
<td>2</td>
<td>0.136552</td>
</tr>
<tr>
<td>Error</td>
<td>0.651664</td>
<td>14</td>
<td>0.046547</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>0.924768</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strategy Design Pattern</td>
<td>Between treatment</td>
<td>0.250229</td>
<td>2</td>
<td>0.125115</td>
</tr>
<tr>
<td>Error</td>
<td>0.795004</td>
<td>14</td>
<td>0.056786</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>1.045233</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4.3 Modularity Analysis

As depicted in Figure 4.1, the modularity criterion is related with Coupling between Components and Services (CBCS), and Lack of Cohesion over Operations (LCOO). Thus, the analysis will cover the behavior of such measures in the data collected for the implementation of the artifacts.

**Descriptive Statistics Analysis.** Figure 4.5 presents the modularity data, respectively for OSGi (4.5(a)), Apache Tuscany (4.5(b)), and JAX-WS (4.5(c)). For modularity reasons required for OSGi, it was necessary to expose interfaces for Item and Category variants, reflecting in low values for both metrics (CBCS and LCOO). In contrast, for Apache Tuscany and JAX-WS, both entities were implemented as simple JavaBean classes, reflecting in a high coupling, and low cohesion for both classes, since they cannot exist independently. In the context of the Parameters mechanism, a Factory Method Pattern was defined to instantiate the PenaltyForDelay and PenaltyForViolation variants. In this way, it can be observed that the values for CBCS metric are lower for OSGi and higher for JAX-WS and Apache Tuscany.

In the OSGi implementation, it was defined an infrastructure for discovering variants explicitly in the code, contrasting the JAX-WS implementation that defines such infrastructure in a XML file. As can be seen in the Figures, the CBCS values are quite similar for OSGi, Apache Tuscany and JAX-WS concerning the FineService. However, the user can choose JAX-WS or OSGi if he prefers to define additional infrastructure that increases the coupling value for the PenaltyForDelay variant, or, he should use Apache Tuscany that does not need to define such infrastructure.

In order to summarize the modularity values, we defined statistical values as detailed
4.4. THE ANALYSIS

(a) OSGi

(b) Apache Tuscany

(c) JAX-WS

Figure 4.5 Modularity Analysis.
in Table 4.7.

Table 4.7 Results for the Modularity Analyses.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Variability Mechanism</th>
<th>Measure</th>
<th>Statistic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>Maximum</td>
</tr>
<tr>
<td>OSGi</td>
<td>Binding variants within services</td>
<td>CBCS</td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LCOO</td>
<td>33.4</td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td>CBCS</td>
<td>3.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LCOO</td>
<td>33.4</td>
</tr>
<tr>
<td></td>
<td>Strategy Design Pattern</td>
<td>CBCS</td>
<td>3.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LCOO</td>
<td>24.6</td>
</tr>
<tr>
<td>Apache Tuscany</td>
<td>Binding variants within services</td>
<td>CBCS</td>
<td>4.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LCOO</td>
<td>32.8</td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td>CBCS</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LCOO</td>
<td>34.8</td>
</tr>
<tr>
<td></td>
<td>Strategy Design Pattern</td>
<td>CBCS</td>
<td>4.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LCOO</td>
<td>36.5</td>
</tr>
<tr>
<td>JAX-WS</td>
<td>Binding variants within services</td>
<td>CBCS</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LCOO</td>
<td>34.6</td>
</tr>
<tr>
<td></td>
<td>Parameters</td>
<td>CBCS</td>
<td>8.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LCOO</td>
<td>34.1</td>
</tr>
<tr>
<td></td>
<td>Strategy Design Pattern</td>
<td>CBCS</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LCOO</td>
<td>40</td>
</tr>
</tbody>
</table>

Analysis of Variance (ANOVA). Table 4.8 summarizes the results of the ANOVA test for the CBCS and LCOO measures. In the analysis of the CBCS metric, the ANOVA did not reject the null hypotheses concerning Binding variants within services and Strategy Pattern mechanisms ($F_0 < F_{CRITIC}$). On the other hand, it rejects the null hypothesis concerning the Parameters mechanism ($F_0 > F_{CRITIC}$). In the context of the null hypotheses rejected, OSGi is a well suitable technology. In contrast, Apache Tuscany was the technology with a middle value followed by a higher value for Apache Tuscany, respectively. Moreover, it also presents that the hypotheses cannot be rejected for the results of the LCOO metric (Table 4.9), since $F_0 < F_{CRITIC}$.

4.4.4 Understandability Analysis

In order to measure the CDCS metric, we considered the following concerns: penalty for delay, penalty for violation, comments, and authentication by password, fingerprint, and card. These concerns are related to variants that are spread in two or more place in the application.

Descriptive Statistics Analysis. Figure 4.6(a) shows the understandability data for OSGi graphically. In the OSGi implementation concerning the Binding variants...
4.4. THE ANALYSIS

Table 4.8 ANOVA applied with level of significance of 5% to coupling between components and services collected data.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Coupling between Components and Services</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Between treatment</td>
<td>5.603922</td>
<td>2</td>
<td>2.801961</td>
<td>0.644482</td>
<td>0.539821</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>60.86667</td>
<td>14</td>
<td>4.347619</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>66.47059</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Between treatment</td>
<td>76.86863</td>
<td>2</td>
<td>38.43431</td>
<td>3.741539</td>
<td>0.050235</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>145.3667</td>
<td>14</td>
<td>10.38333</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>222.2333</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Between treatment</td>
<td>3.02549</td>
<td>2</td>
<td>1.512745</td>
<td>0.271402</td>
<td>0.76623</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>78.03333</td>
<td>14</td>
<td>5.57381</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>81.05882</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.9 ANOVA applied with level of significance of 5% to lack of cohesion over operations collected data.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Lack of Cohesion over Operations</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Between treatment</td>
<td>16.51569</td>
<td>2</td>
<td>8.257843</td>
<td>0.007607</td>
<td>0.992426</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>15.19737</td>
<td>14</td>
<td>1085.526</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>1521.388</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Between treatment</td>
<td>436.1922</td>
<td>2</td>
<td>218.0961</td>
<td>0.222403</td>
<td>0.803367</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>1372.87</td>
<td>14</td>
<td>980.6333</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>14153.06</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Between treatment</td>
<td>694.3588</td>
<td>2</td>
<td>347.1794</td>
<td>0.366646</td>
<td>0.699516</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>13256.7</td>
<td>14</td>
<td>946.9071</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>13951.06</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

within services mechanism, it was defined the bundle context and additional code for services discovery. In the context of OSGi, the PasswordAuthentication concern is spread in two classes and a spring configuration file. It increases the value of the CDCS metric, which is reflected by adding the reference for the PasswordAuthentication in the applicationContext-osgi.xml configuration file that will be available for a particular client that wants to use such reference.

Figure 4.6(b) shows the understandability data for Apache Tuscany and Figure 4.6(c) shows the understandability data for JAX-WS. As it can be seen, the value for Apache Tuscany is lower because the PasswordAuthentication concern is spread over two classes (Authentication and PasswordAuthentication) and the spring tuscany-content.xml file. On the other hand, for JAX-WS, it is spread over three classes and the spring beans.xml file. In the Parameters mechanism side, what changes is that instead of mapping concerns in the spring files, here it will be mapped in a class responsible for load the configuration
file, the *PropertiesFactory* class. However, when we compare the three technologies, OSGi and Apache Tuscany has a quickly lower value with respect to each other and the JAX-WS. In the Strategy Pattern, only we have to do is mapping the concerns in a context class. In the Apache Tuscany and JAX-WS side, the higher value (5) for CDCS occurs because we decided to adapt the Strategy, creating a *Data Structure Type (new abstraction)* for defining the *PenaltyForDelay* variant due to constraints for web services implementation.

**Analysis of Variance (ANOVA).** As can be seen in Table 4.10, the ANOVA test did not reject the null hypotheses concerning Parameters and Strategy Pattern mechanisms ($F_0 < F_{\text{CRITIC}}$). On the other hand, it rejects the null hypotheses concerning the *Binding variants within services* mechanism ($F_0 > F_{\text{CRITIC}}$). In the context of the null hypotheses rejected, Apache Tuscany is well suitable. In the context of the hypotheses that was not rejected, priority levels will be defined according to the points discussed, for each technology.

**Table 4.10** ANOVA applied with level of significance of 5% to concern diffusion over components and services collected data.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Concern Diffusion over Components and Services</th>
<th>SS</th>
<th>DF</th>
<th>MS</th>
<th>F0</th>
<th>p-value</th>
<th>F_{\text{CRITIC}}</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binding variants within services</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Between treatment</td>
<td>3</td>
<td>2</td>
<td>1.5</td>
<td>7.5</td>
<td>0.005524</td>
<td>3.68232</td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td>3</td>
<td>15</td>
<td>0.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>5</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Between treatment</td>
<td>1</td>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>0.391127</td>
<td>3.68232</td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td>7.5</td>
<td>15</td>
<td>0.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>8.5</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Strategy Design Pattern</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Between treatment</td>
<td>1</td>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>0.391127</td>
<td>3.68232</td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td>7.5</td>
<td>15</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>8.5</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**4.4.5 Analysis Conclusion**

The descriptive statistics showed that there are some aspects that must be considered when using technologies to implement variability, and it was detailed in the analysis for each criterion considered, such as *complexity, stability, modularity, and understandability*. In this sense, it was verified quite differences among the data analyzed for both technologies in the descriptive statistics, and most of the data analyzed were not significant, which turns hard to draw a concrete conclusion saying what technology is better. However, it was important to lead us to define an initial decision model based on some important
4.4. THE ANALYSIS

(a) OSGi

(b) Apache Tuscany

(c) JAX-WS

Figure 4.6 Understandability Analysis.
analyses earlier discussed. Furthermore, the ANOVA applied to test the hypotheses did not provide sufficient data to reject the null hypotheses. However, in some cases, some null hypotheses were rejected and helped to the decision model definition. Nevertheless, we can conclude that a more detailed study is needed in order to verify more deeply some hypotheses that were not rejected. For instance, as the size of the domain evaluated was simple, other domains could be used in order to identity the application of the variability mechanisms to implement specific feature types. Moreover, the use of other variability mechanisms may be useful in the context of service-oriented product lines, such as Program Transformations, Aspect-Oriented Programming, Generics and many other Design Patterns (Gamma et al., 1995).

Nevertheless, we believe that another case study following these issues can be more conclusive saying which technology is more appropriate according to specific variability mechanisms, and thus improving precision of the decision model.

4.5 Decision Model

The motivation for the decision model relies on the management of variability identified in the core assets, and how complex is the mapping of such approach in a service-oriented environment. Nevertheless, differently of the existing models (Gacek and Anastasopoulos, 2001; Muthig and Patzke, 2002), our decision model is based on a quantitative analysis. In the work reported in (Ribeiro, 2008), they constructed a decision model in the product lines context, for recommending only variability mechanisms for three criteria. In contrast, our approach has the focus on the service-oriented product lines context, which implies in the analysis of different criteria and metrics for recommending service technologies combined with variability mechanisms.

Since our decision model is variability mechanism centric, the first input consists of the variability mechanisms (e.g., Binding variants within services, Parameters, and Strategy Design Pattern). The second input is some criteria used to compare the technologies. The criteria are the quality factors, e.g., complexity, stability, modularity, and Understandability factors. Finally, the last input consists of a catalog of technologies (e.g., OSGi, Apache Tuscany, etc).

As exemplified in Figure 4.7, the output is represented by a tuple that contains besides the technologies recommended, the variability mechanisms and a particular criterion. The tuple depicted in the Figure is an example of the result. It is important to highlight that not always will be possible to decide which technology fits better to a set of criteria.
For instance, if the *complexity* criterion is considered and all the technologies are similar according to such criterion, both technologies are recommended. However, in some cases, we leave the decision for the user. This decision is based on the points discussed in the analysis section of the case study.

For a better understanding of the model and due to the size limitation, we decide to split the model in three parts as depicted in Figure 4.8, Figure 4.9 and Figure 4.10. However, the three parts can be considered as a unique decision model as a whole. The "lamp (idea)" icon indicates the result of the answer about the criterion question (represented by the "?” icon), i.e., the technology(ies) recommended. The answer is performed in two ways: **yes** or **no**. The "**yes**" icon indicates that the technologies satisfy a particular criterion.

Moreover, the "**yes**" also indicates the low complexity with respect to the “complexity criterion”. On the other hand, the “**no**” is a negative aspect. Moreover, as can be seen in the figures, **priorities** were defined. These priorities were defined based on the analysis discussion, in particular when the hypotheses were not rejected, i.e. there is no difference among the technologies. Thus, each of the technology was marked according to a specific priority that differentiates each one. Furthermore, it was also defined a “v” with respect to the metrics values applied for each criterion. As the complexity and modularity criteria contain two metrics for each one, thus the values where added. For the understandability criterion, the concerns values were also added.

For instance, as can be seen in Figure 4.8, OSGi, Apache Tuscany, and JAX-WS were considered suitable to be applied using the *Binding variants within* mechanism in a requirement that needs low complexity. In this example, OSGi was defined with the *Priority 1*, since according to the analysis of the values in subsection 4.4.1 OSGi is suitable in some situations that provide an implementation with low complexity.

---

**Figure 4.7** An application of the decision model.

![Decision Model Diagram](image-url)
4.6 Chapter Summary

This chapter presented an assessment through the definition, planning, analysis and interpretation of a case study performed to evaluate technologies (OSGi, Apache Tuscany and JAX-WS) for implementing variability with the support of variability mechanisms.
4.6. CHAPTER SUMMARY

Figure 4.10 Strategy Pattern mechanism model.

(binding variants within services, parameters, and strategy design pattern) in service-oriented product lines. Moreover, the output of the assessment was a decision model to guide software engineers in the task of choosing service/component technologies when dealing with issues related to variability mechanisms in service-oriented product lines projects.

The results of the case study pointed out that the differences among the technologies for implementing variability, is very few. However, it was important to lead us for defining an initial decision model based on some important analyses earlier discussed. It is important to highlight that when we were building the decision model, some decisions based on the descriptive statistic analysis were considered, and some technologies were pointed out with a priority level. In most of the cases, OSGi was the technology marked with first priority level as can be seen in the decision model. In this sense, such technology may be suitable to be applied in service-oriented product lines and it will depend of the context or customer needs.

Nevertheless, more variabilities must be considered such as, service contract variability, which is important to understand about services, priorities, responsibilities and warranties through the Service Level Agreement (SLA). Consequently, the use of other variability mechanisms decisions for service-oriented product lines must also be applied
4.6. CHAPTER SUMMARY

to refine the model. In addition, the replication of the case study using more subjects is needed in order to compare it with the results previously reported.

In the next Chapter, an approach to implement core assets in service-oriented product lines is defined through systematic activities, tasks, steps, and roles, using among others inputs, the decision model defined earlier.
5

SOPLE-IM: An Approach to Implement Core Assets in Service-Oriented Product Lines

Run, rabbit run.
Dig that hole, forget the sun,
And when at last the work is done
Don’t sit down it’s time to dig another one.

—ROGER WATERS, DAVID GILMOUR AND RICHARD WRIGHT - MUSICIANS (Breathe, The Dark Side of The Moon)

5.1 Introduction

The purpose of this chapter is to describe a practical approach for implementing core assets in service-oriented product lines providing guidance to manage variability at the code level. The approach provides the mapping of Service-Orientation (SO) concepts to the Software Product Lines (SPL) context with guidelines for applying variability mechanisms according to different levels of granularity, besides the binding time to decide which possible services will be excluded and included in an application (Dolstra et al., 2003). The idea is to be possible to switch between two or more features, or remove an optional feature without too much effort using Open Services Gateway Initiative (OSGi) technology combined with these variability mechanisms. OSGi was chosen due to its applicability in many different scenarios, in particular, its modular way to develop SOA applications dynamically and the possibility to manage features at runtime by its life-
5.2 PRINCIPLES

In order to provide a practical and effective way to implement services, components and variability in the service-oriented implementation phase with the support of OSGi technology, the approach is based on a set of Implementation Principles (IP). The motivation behind the definition of these principles is the mapping of some SOA characteristics, such as interface-based design, loose coupling, extensibility, discoverability, composability, and coarse-grained interfaces (see Chapter 3) for the approach’s purposes. It is important to highlight that many of the characteristics are intrinsic in the technology adopted, and their discussion will not be restricted for this section thus being discussed along the Chapter.

**IP 1. A component or service must have well-defined interfaces:** the functionality of a component is defined by its interface. In the approach, one or more components can provide one or more interfaces that can be used to provide functionality for services and require one or more interfaces or services. In the same way, the services can expose interfaces in service-oriented applications including operations in which are implemented internally by components (Medeiros et al., 2009). The concept of required interfaces exposed is essential to enabling software plug-and-play (Bronsard et al., 1997; Almeida, 2007). They expose operation invocations that a service makes to others as well. On the other hand, services can be connected together to form service compositions usually called as services orchestrations (Erl, 2005). Such connection is realized by binding a required service of some components to a provided service of another one in the context, for example, a web interface.

**IP 2. Transparent life cycle mechanism:** the services and components need an independent mechanism to manage their life cycle transparently. For instance, the Life Cycle Layer in OSGi allows bundles to be managed in one of the following states depicted in Figure 5.1 (Alliance, 2007): The bundle must first be Installed. When it is required
to start, the package-level dependencies with other bundles are *Resolved*. When all dependencies are resolved, the bundle activator is launched: the `start()` method is called, and the related code is executed. Typically, these operations consist in configuration and publication of services. Next, the bundle is in the *Started* state. Moreover, *Updating*, *Stopping* and *Uninstalled* (the bundle has been uninstalled, thus, it cannot move to another state) build the last possible operations for bundle management.

![Bundle life cycle](image)

**Figure 5.1** Bundle life cycle *(Alliance, 2007)*

**IP3. Separation of Concerns and Information Hiding**: the decomposition of applications into services and components allows the Separation of Concerns (SoC) in specific self-contained and replaceable building blocks *(Medeiros et al., 2009)*. The declarative approach employed by OSGi provides a non-invasive way to take advantage of understandability and modularity capabilities. In this sense, OSGi separates concerns into bundles comprising not just application code, but also other resources that together can provide services and packages to other bundles *(Alliance, 2007)*.

**IP4. Modularity**: Parnas, in his work *(Parnas, 1972)*, stated that modularity is closely related to design decisions that decompose and organize the system into a set of modules. In the context of OSGi, such behavior can be achieved through the decomposition into services in a high-level and components in a low-level. The Service-Orientation approach enforces such decomposition criteria aiming at hiding the implementation details of services into components. It reflects some benefits, such as, reusing specific services available in other applications or even another service in the same application.

**IP5. Granularity**: according to *(Papazoglou and Georgakopoulos, 2003)*, granularity refers to the scope of functionality by a service or a component. In this
5.3 Overview

The approach provides the implementation of core assets in the context of service-oriented product lines with the use of OSGi as the main technology to control the life-cycle of the core assets. Thus, it focus on the implementation step, abstracting the previous steps, such as the service-oriented product lines analysis and design. The next sub-sections present an overview of the inputs, outputs, roles, development cycles and activities of the approach.
5.3. OVERVIEW

5.3.1 Inputs and Outputs

The approach receives four mandatory artifacts that are used as inputs for the service-oriented domain implementation. These artifacts are: Business Process Model (BPM), Domain Feature Model, the Service-Oriented Domain Design Specification Architecture (SO-DDSA), and the Decision Model defined in Chapter 4. The decision model can be used by the software engineers to define for a given quality attribute, which variability mechanism can be selected through a catalog of technologies used as input. These artifacts are briefly described as follows.

Business Process Models. The business process models represent the business process for a specific domain and are usually modeled using Business Process Modeling Notation (BPMN) or the Unified Modeling Language (UML) activity diagrams (Razavian and Khosravi, 2008; Medeiros et al., 2009). Furthermore, it is able to model an appropriate process variant (i.e., the models can contain variability), suitable for specific customer needs. In this way, business process activities can be marked as mandatory, optional or alternative (Ye et al., 2007; Boffoli et al., 2008; Medeiros et al., 2009). Figure 5.2 shows an example of a BPM represented as a UML activity diagram. The model represents a submission process in the context of a conference management domain. The dashed balloons are the representation of optional variability.

![Figure 5.2 Example of a business process model](image)

Domain Feature Model. The feature model explicitly represents the commonalities and variabilities of the service-oriented product line, i.e., common and variable aspects of the domain. Moreover, it provides the basis for designing, developing and configuring reusable core assets. Figure 5.3 shows an example of a feature model using the notation proposed by (Kang et al., 1990) and implemented by (Lisboa et al., 2007). The diagram represents an alternative feature type in the conference management domain, where a submission can be complete or partial.

Service-Oriented Domain Specification Architecture. The SO-DDSA is responsible for identifying and describing services and components, as well as, variability modeling and particular decisions, e.g., variability mechanism support (Medeiros et al., 2009).

Decision Model. Modeled in terms of a mind map, the decision model aims to
5.3. OVERVIEW

Figure 5.3 Example of a feature model

recommend services technologies based on given inputs, such as *variability mechanisms* (e.g., parameters and design patterns) and some *criteria* (e.g., complexity, modularity, and stability). Therefore, the use of a decision model can be applied to represent the relationships among technologies, mechanisms, and binding times, providing useful guidance to software engineers.

Differently of many work in the literature that focus in service-oriented product lines implementation approaches (Segura *et al.*, 2007; Günther and Berger, 2008; Smith and Lewis, 2009), we intended to provide guidelines and steps for implementing service-oriented aspects, such as, *service providers*, *service consumers*, *service register*, *service- orchestrations* (Erl, 2005), and *components*, as in software product lines. In addition, some variability implementation mechanisms are suggested to implement specific feature types mapped as aspects aforementioned. Hence, the approach is concerned with creating common and variable artifacts including reusable services and components. Consequently, the output of the approach comprises all the reusable services and components implemented respecting the implementation principles defined in Section 5.2.

5.3.2 Roles

A role defines the behavior and responsibilities of an individual or group of people as conceptualized by actors, working together as a team. According to (Jacobson *et al.*, 1999), in the context of an approach or process, it is also important to assign roles to actors indicating who can perform an activity. Thus, the approach defines one role described as follow:

- *Domain Architect*: specialist in variability modeling, responsible for the design of variation points in the architecture and definition of the variability implementation mechanisms that are going to be used; and
5.3. OVERVIEW

• Developers: responsible to implement and deploy the components and services in the target technology (e.g., OSGi), using the SO-DDSA as a reference guide.

5.3.3 Development Cycles

The service-oriented domain implementation is divided in two lifecycles as in software product lines (Clements and Northrop, 2001; Pohl et al., 2005): core assets and product development. The core assets development aims to provide guidelines and steps for implementing variability in reusable artifacts, in this case, services and components. During product development, these architectural elements are specialized to a particular context according to specific customer requirements or market segment needs (Medeiros et al., 2009). However, the approach focuses only in the core assets development, in particular, on the implementation of reusable assets in the context of service-oriented product lines applications.

5.3.4 Activities

The service-oriented domain implementation approach starts with the implement services and components activity. It receives the business process models, SO-DDSA, the domain feature model and the decision model as mandatory inputs. This activity is composed of tasks that provide guidelines for the developers to implement services and components in the provider and consumer side with the support of the integration with the OSGi technology. Moreover, some guidelines are presented in order to suggest variability mechanisms to implement specific feature types according to a set of criteria. Although the approach starts with this activity, it is important to mention that in most cases it can be performed in parallel with the other two activities assess services and components activity and refine services and components specification activity.

The assess services and components activity receives the service providers/consumers and components implemented, and the assessment framework define in Chapter 4 as mandatory inputs. The main goal of this activity is to provide guidelines for assessing through software metrics the service providers/consumers and components implemented in order to guarantee that its implementation is in accordance with the criteria defined in the assessment. In the case of a negative result, then, the developers must follow to next activity in order to refine them.

After assessing the services and components, the next and last activity is the refine services and components specification activity, which receives the SO-DDSA, domain
feature model and the business process models as the mandatory inputs. The decision model is also a mandatory input and is required for guiding the developers and domain architects to draw decisions in terms of variability with respect to quality factors that services and components must have, e.g. *modularity* and *complexity* in the context of OSGi technology. The main goal of this activity is to perform the refinement of the detailed design, which is executed between the developers and domain architects. Figure 5.4 shows the activities of the approach. As it can be seen, its activities are performed in parallel.

![Figure 5.4 Approach Activities.](image)

### 5.4 The Approach

The approach for service-oriented domain implementation consists of three activities: (i) *implement services and components*, (ii) *assess services and components*, and (iii) *refine services and components specification*. Since the approach activities are based on the OSGi technology (Alliance, 2007), the term *bundle* will be used to represent OSGi services and components that provide services to other services or components. Additionally, the Spring Framework (Spring, 2009) was combined with OSGi, since Spring applications can be deployed in OSGi execution environment, and that take advantage of the services offered by the OSGi technology. This integration can be performed through the use of the Spring Dynamic Modules (SpringDM) (Spring, 2008).
According to (Cirilo et al., 2008), SpringDM provide a smooth integration between Spring and OSGi by allowing an application to import and export Spring packages and services.

This integration is possible because the Spring framework offers a model to applications as a collection of simple components and services, which is the core of service-oriented applications that may be customized using, for example, the Dependency Injection mechanism (Fowler, 2004) to inject dependencies at runtime. In order to clarify and explain the activities, some examples scenarios will be described using a conference management domain. Such domain is a relevant motivation, since different conferences may expose their business processes through services, allowing integration with services exposed by other conferences for controlling the life-cycle of the submission of papers and reviews. However, they must follow the principles described in Section 5.2, and, supply the roles and inputs defined in this approach. Figure 5.5 shows the simple feature model of the conference management domain used as input for the approach and Figure 5.6 shows a UML class diagram, which describes by the SO-DDSA, the services composition in which services and components are distinguished by stereotypes. This diagram is important for the software engineers because it exemplifies which services or components provide functionalities and which services or components are being consumed by other services or components.

5.4.1 Implement Services and Components

This activity receives the architectural decisions documented in the SO-DDSA, the business process models, and the domain feature model as mandatory inputs. The architectural decisions comprise the services interfaces definition as well as its internal information, such as the components that hold the internal implementation of the services with the variability implementation mechanism support (when it is required). The motivation to use components to implement services internal functionality emerges from the need to separate service-oriented mechanisms (e.g., communication) from the logical code implementing the services behavior. These service-oriented mechanisms are delegated to a component container, which will interact with the service registry (through a discoverability mechanism employeed by OSGi) at runtime to find the services when available.

Hence, there are four tasks that should be followed to perform the implement services and components activity: (i) describe service and component provider, (ii) describe service and component consumer, (iii) implement service and component, and (iv) deploy
5.4. THE APPROACH

Figure 5.5 Conference Management Domain Feature Model.

Figure 5.6 Bundles Diagram.
service and component. The importance to start the implementation activity by describing both service and component providers/consumers is that they are essential to map the dependencies among the services or components identified in the design step. These dependencies describe important variation points and the approach must deal with them through the OSGi life-cycle mechanism that integrates services (or service-orchestrations) and components, allowing their communication.

In general, the service-oriented product line approaches (Segura et al., 2007; Günther and Berger, 2008; Smith and Lewis, 2009) do not focus these aspects, providing only possible solutions to deal with variability implementation in a conceptual level. However, besides to perform the variability implementation, the description is important to be used as a start point for the implementation tasks. For example, the tasks responsible for describing service and component providers/consumers map possible variation points identified in the conference management that may occurs in a particular application, and provides guidelines on how to deal with them, facilitating the integration, the maintenance of the services and components, and the software engineer general view of the application. Figure 5.7 shows the inputs, outputs, tasks and roles of this activity.

![Figure 5.7 Implement Services and Components Activity.](image_url)
Describe Service and Component Provider. It receives the SO-DDSA as the mandatory input, which contains the services identified, specified and modeled. There are two roles responsible to perform this task: developers and domain architects. In OSGi applications, the service and component provider description specifies relevant information, such as the bundle name, vendor, version, package, among others, which are described in the manifest file of the bundle (Alliance, 2007). This file provides information about the service or component, which OSGi technology needs to correct install and activate (Almeida, 2007). In this file, the most important properties are Bundle-Activator and Export-Package. The Bundle-Activator informs the container which class is responsible for activating a bundle. On the other hand, the Export-Package is mandatory and informs the container, which services or components can be consumed by other services or components in the same application or different ones.

In the service-oriented product lines design and implementation, the component describes the internal implementation of services, thus, enabling the separation of concerns. In this sense, components and services may encapsulate variabilities in different levels of granularity, for example, coarse-grained variabilities (encapsulated as components or services). For example, variabilities encapsulated as services correspond to a particular type of service composition in which variation points and variants are separated as services that are linked at runtime to meet complex service requirements. The submission feature depicted in Figure 5.5 is a potential variation point, since in the conference domain a submission can be partial or complete. Moreover, it is the main activity that must be performed in a conference management domain. In order to exemplify it, a conference A may require a partial submission service that a conference B provides.

On the other hand, components may encapsulate variabilities in a low level (internal components that provide internal functionalities for the services), or may encapsulate general functionalities that will be required by other components and services. These general components might be separated as bundles when persistence activities are defined. For example, due the dynamic behavior of OSGi, it is possible to isolate different persistence frameworks in a data access component bundle and export both to be reused for every bundle activated in different products.

Describe Service and Component Consumer. The second task is similar to describe service and component provider task, but its main focus is on the services or components that are consumed. In order to import the required service or component, the service and component provider description must be available, which makes it a mandatory input for
5.4. THE APPROACH

this task. Furthermore, it also receives the SO-DDSA as the mandatory input. In this task, the Import-Package header must be declared, specifying the name of the service that will be consumed. It is also available for the components implemented as separated bundles. Hence, the service that requires the implementation of the component must import the package containing the class responsible for such functionality.

In the example of the SubmissionServiceBundle, if the conference A requires the partial submission service provided by the conference B, thus, the conference A must include in the manifest file of submission service bundle the Import-Package, in which will import the appropriate variant (partial submission). The roles responsible for performing this task are the developers and domain architects. Moreover, the output of this task is the service and component consumer description depicted.

Implement Service and Component. The goal of this task is to implement the services and components described in the previous tasks, which make them the main inputs for this task. There are two roles to perform this task: developers and domain architect. In this approach, the domain architect will act as a consultant and developer too. These services and components are implemented with the support of the SpringDM. It combines the Spring bean management with the OSGi services behavior (providers and consumers), as well as the service register in order to make them available. This task also provides guidelines to manage variability at the code level using the descriptions defined earlier. In particular, using appropriate variability mechanisms to implement specific variants in the Spring’s OSGi environment. This activity is organized through implementation steps because it needs to implement the dependencies defined in the description tasks. The approaches (Günther and Berger, 2008; Smith and Lewis, 2009) that focus on the variability implementation do not define systematic activities on how service and component providers/consumers are implemented within variability mechanisms.

This task receives the BPM, SO-DDSA, the feature model, and the decision model as the main inputs for this task. Therefore, in order to provide a practical way to implement the services and components, this task was divided in four steps: implement business entities, implement service and component provider, implement service and component consumer, and implement variability. These steps are important to guide software engineers to deal with SpringDM concerns, variability mechanism implementation, and feature types. Moreover, in most cases, these steps can be performed in parallel.

Implement Business Entities. The SO-DDSA is an input for this step, because it contains the entities diagram, which is important to represent the relationship among the
entities. In the Spring’s OSGi environment, the business entities identified and modeled must be encapsulated as entity service (Erl, 2005) provider’s bundles. Accordingly, it is important that every entity service bundle must specify the Activator class in its manifest file, as reported in describe service and component provider task. The Activator class implements the BundleActivator interface, which requires the implementation of two methods, start(BundleContext) and stop(BundleContext). The start() method can allocate resources that a service needs, start threads, register services, and more. On the other hand, the stop() method must clean up and stop any running threads or specific service configurations (Almeida, 2007). The relationship between the framework and its installed services is performed by the use of BundleContext object. A BundleContext object represents the execution context of a single service within the OSGi platform, and acts as a proxy to the underlying framework. This step is usually performed at the same time when the following steps are being defined.

Implement Service and Component Provider. In this step, the developer and the domain architect must implement an interface of the service implementation that exposes functionalities in which will be consumed by other services or applications. The service interface is instantiated via Spring as a bean. In addition, the software engineer must create a META-INF folder, a spring sub-folder, and define a spring-context configuration file into these folders. Such file is usually defined with the name of the service, for example, <service-name>.xml. Likewise, it is responsible for communicating the OSGi framework with the service interface that must be registered. Therefore, it can be performed by the definition of the <osgi:service> tag for register the service that uses the bean responsible to instantiate the service implementation as a reference in a <service-name>-osgi.xml file. In the same way, it can be performed for every component that provides internal functionality for a service and is placed in a separate bundle.

In the context of the conference management domain, the NotificationServiceBundle depicted in Figure 5.6 may be used as example to clarify the points previously discussed. For a better understanding of the example, Figure 5.8 shows the notification feature that the NotificationServiceBundle encapsulates. As can be seen in the Figure, the NotificationServiceImpl is responsible to implement the well-defined INotificationService interface. This interface will be exposed as a service and further consumed by other services or components.

The submission service implementation in the provider side can be detailed in the following code snippet 5.1.
5.4. THE APPROACH

**Figure 5.8** Notification Service Diagram.

**Listing 5.1** Implement Service and Component Provider example

```java
public interface INotificationService {
    void sendMail(String from, String[] to, String subject,
                   String message, ICallback callback);
}
```

```java
public class NotificationServiceImpl implements INotificationService {
    private ISubmissionComponent submissionComponent;

    public NotificationServiceImpl(ISubmissionComponent submissionComponent) {
        this.submissionComponent = submissionComponent;
    }
}
```

```xml
<beans>
    <bean id="submissionServiceBean" class="br.com.rise.puresubmission.service.impl.SubmissionServiceImpl"/>
</beans>
```

```xml
<beans>
    <osgi:service id="submissionServiceOSGi" ref="submissionServiceBean" interface="br.com.rise.puresubmission.service.ISubmissionService" />
</beans>
```

In the next step, the *service and component consumer implementation* will be detailed.
using the notification service example. However, the context will be addressed in the case the notification service will be consumed by another service, for example, the submission service.

**Implement Service and Component Consumer.** Regarding the properties defined in describe service and component consumers task, the main goal of this step is to provide low coupling between the required service and the code responsible to perform the service tracker. For instance, in a pure OSGi implementation, a service tracker must be implemented to hold all currently available services. In the SpringDM, it is performed transparently. Thus, the developer and the domain architect must define in the OSGi service configuration file depicted in the previous step the `<osgi:reference>` tag for the service interface required. In this way, services can be more easily modified or replaced in the future without the need to change the code. In order to consume the required well-defined service interface, the developer must inject the interface reference.

Such injection is performed by Spring transparently, which makes the code simple and easy to understand. For example, as can be seen in Figure 5.6, the SubmissionServiceBundle needs the NotificationServiceBundle to notify the authors when a paper is submitted. This scenario is exemplified in the BPM depicted in Figure 5.9.

![Figure 5.9 Submission Business Process Model.](image)

In this example, the SubmissionServiceImpl injects the interface of the required service, e.g. INotificationService. In order to correct instantiate such service, the developer must defines in the submissionservice-osgi.xml, for example, the reference for the notification service. After this definition, the next step is to set the Spring property in the submission implementation bean in the submissionservice.xml file to allow the injection of the service. This example is depicted in the following code snippet 5.2.

**Listing 5.2 Implement Service and Component Consumer example**

```java
public class SubmissionServiceImpl implements ISubmissionService {
```

80
Implement Variability. The main goal of this step is to provide guidelines to manage variability at the code level through example scenarios. Particularly, to use appropriate variability mechanisms for handling SPL core assets implemented as OSGi services in the Spring’s OSGi environment. Furthermore, most of these core assets should be bound with each other at runtime, and may be encapsulated in different levels of granularity resulting not only in the implementation of OSGi services, but in the implementation of classes, components, and services. It is necessary because, depending of the granularity level (Kästner et al., 2008), for example, coarse-grained variabilities, these core assets should be excluded or included of specific products or applications easily, depending on the business activities. Regarding the approach principles defined in Section 5.2, it is important to define these granularity concerns, once the approach treats service-oriented product lines implementation as services in the high-level and more internally, the components that provide functionalities for such services.
In this way, in order to select appropriate variability mechanisms, we decide to use the decision model defined as result of the case study described in Chapter 4 as input. The case study described the implementation of a SPL project for selecting suitable service technologies combined with variability mechanisms according to some criteria. The result of the case study was a decision model modeled in terms of a tuple of kind: \( R = \{ \text{Variability Mechanism, Criteria, Catalog of Technologies} \} \). In the current status of the decision model, three variability mechanisms were defined (e.g., Binding variants within services, Parameters, and Strategy Design Pattern) implemented with OSGi, Apache Tuscany, and JAX-WS technologies. In this sense, the developer or domain architect is able to consult the decision model maps and verify which variability mechanisms are suitable in the context of OSGi technology.

Additionally, due the dynamic behavior of service-oriented product lines applications where the variants should be bound with each other at runtime, as well as granularity decisions are important, some variability mechanisms must be applied to fit these aspects. However, it is important that these specifics variability mechanisms must be combined with other mechanisms for allowing a better management of variabilities, for example, the communication among service consumers and providers that encapsulate variabilities, and the decomposition of a variation point into services and components. Thus, in order to provide guidance for the developers and domain architects to deal with such aspects, we decide to apply the following variability mechanisms: (a) binding variants within services, (b) parameters and configuration files, and, (c) encapsulating variability within a service. However, there are other variability mechanisms in the literature (Segura et al., 2007; Smith and Lewis, 2009) that may be applied to fit these aspects. Nevertheless, they focus on web services development aspects, in contrast with our approach that focuses on OSGi services. The reasons for this choice, as well as, the guidelines for implementing them, are detailed as follow.

\( a) \) Binding variants within services

In SOA applications, the binding time of service consumers and providers are usually defined at runtime, without the need to explicit dependencies among them. It allows the loose coupling, since service consumers do not need to know how service providers implement its functionalities. Thus, enabling what will be exposed and what will be hidden in a modular way. It is important in order to decrease the complexity and to allow a better understandability of the services implementation by developers and domain
architects. In this sense, dependency injection as a mechanism for binding variabilities may be applied using a container to perform such binding.

The choice of this variability mechanism is based on the decision model defined in Chapter 4, where it can be applied following definition: \( R = \{ \text{Variability Mechanism, Criteria, Catalog of Technologies} \} \). Hence, we applied such map in a scenario, which receives as inputs: \( R = \{ \text{Binding variants within services, (Modularity and Complexity), OSGi} \} \). This scenario was detailed in Chapter 4 in which OSGi was used integrated with Spring demonstrating its effect with a context file used to map the dependencies and its relationships at runtime. Moreover, due to granularity concerns this mechanism was used also to inject other types of resources, such as objects and components internally within the services.

Figure 5.9 will be used as a practical example, since it contains the flow of the submission main activity, i.e., author submitting papers. It is important to note that this activity contains variability, i.e., the authors can submit abstracts or papers. This feature type is depicted in the feature model (Figure 5.5). In the feature model, the submission service was classified as an alternative feature type in which a submission can be partial (abstract submission) or complete (paper submission). As a consequence, both partial and complete submissions were considered variants of the submission feature (variation point). Figure 5.10 depicts the submission service bundle diagram defined by the SO-DDSA representing the variability of the submission feature.

As can be seen in the figure, there are two alternative variants, each one corresponding...
5.4. THE APPROACH

to PartialSubmissionComponentImpl and CompleteSubmissionComponentImpl. These variants are part of the submission service. However, in order to decide which variability will be bound, it is necessary to create a common point that must be accessible by the submission service. In this way, the interface SubmissionComponent was created and the both variants must implement such interface. Consequently, if the author wants the complete submission functionality, than the developer must define a bean for such variant and inject it reference in the submission service bean. Indeed, such reference will be the common point represented by the SubmissionComponent as depicted in the following code snippet 5.3.

Listing 5.3 Binding variants within services example

```java
public class SubmissionServiceImpl implements ISubmissionService {
    // ...
    public SubmissionServiceImpl(SubmissionComponent submissionComponent) {
        this.submissionComponent = submissionComponent;
        // ...
    }
}
```

```
<beans>
    <bean id="submissionServiceBean"
        class="br.com.rise.puresubmission.service.impl.SubmissionServiceImpl">
        <constructor-arg ref="completeSubmission" />
    </bean>
    <bean id="completeSubmission"
        class="br.com.rise.submission.component.impl.CompleteSubmissionComponentImpl"
        parent="submissionComponentParent" />
</beans>
```

The solution previously presented corresponds to the implementation of an alternative feature type for demonstrating the runtime binding with the binding variants within service using the component implemented internally within a service provider. Further, will be presented another possible way to implement the same example, but using variabilities encapsulating within services. This example will present the separation of the variants (encapsulated as services) and the submission service consuming the partial submission exposed as a service (instead of components), allowing thus a service composition.

b) Parameters and Configuration Files

According to (Smith and Lewis, 2009), Parameters is a simple variation mechanism
in which parameters are used to invoke different variations of a service, for example, the *notification type* that must define to send messages for the authors in the conference domain example. In order to decide the notification type, *Configuration Files* (Gacek and Anastasopoulos, 2001; Santos and Santos, 2005) can be useful to separate the source code from the definition of the parameters that may change, facilitating the code *understandability* and *modularity*. This motivation is depicted in the following map that we applied using the decision model definition receiving the following inputs: \( R = \{ \text{Parameters}, \text{(Modularity and Understandability)}, \text{OSGi} \}. \) This map was defined through the results of the decision model applied for the *Parameters* mechanism defined in Chapter 4. In this sense, the feature *news* depicted in Figure 5.5 will be used to represent an optional feature type using a scenario example. Such feature is responsible to *send event news*, e.g. deadline changes.

Suppose that the *news* feature is included in the event service operation, which is responsible to send event news. In this example, the service consumer is free to select the news type. However, it must be available in the product, otherwise, an exception is thrown. The developer is responsible to define such functionality in a *configuration file* that must be loaded in the *spring-context file*. In this way, the software engineer creates a *bean* responsible to instantiate the *java.util.Properties class* that will map the corresponding variant. In the SpringDM, after the definition of the properties file, the software engineer injects it and the Spring will load the file internally. Thus, if the variant was included, the developer needs to define a conditional statement passing as a parameter the string containing the name of the functionality. This example scenario is described in the following code snippet (Listing 5.4).

**Listing 5.4 Parameters and Configuration Files example**

```java
91 EventTest.java
92
93 public class EventTest {
94     // [...] 
95     // Injecting the configuration properties, which will be initialized
96     // by the Spring container
97     public void setConfigurator(Properties configurator) {
98         this.configurator = configurator;
99     }
100
101     public void start() throws NotificationNotFoundException {
102         // Select the property mapped in the configuration file
103         String notification = this.configurator.getProperty("notification");
104         // This condition verify if the optional feature was included
105         // or not. If not, then, an exception is thrown. Otherwise, the
106         // sendEventNews operation is executed.
107     }
```

5.4. THE APPROACH
if(notification.equals("news")){
    this.eventService.sendEventNews();
} else {
    throw new NotificationNotFoundException("The functionality news "
    + "is not available in this product.");
}

<beans>
    <osgi:reference id="notificationServiceOSGi" interface="br.com.rise.notification.service.INotificationService" />
    <osgi:service id="eventServiceOSGi" ref="eventServiceBean" interface="br.com.rise.event.service.IEventService" />
</beans>

<beans>
    <bean id="eventServiceBean" class="br.com.rise.event.service.impl.EventServiceImpl">
        <!-- Consuming the notification service -->
        <constructor-arg ref="notificationServiceOSGi" />
    </bean>
    <bean id="propsBean" class="java.util.Properties">
        <constructor-arg>
            <props>variability.properties</props>
        </constructor-arg>
    </bean>
    <bean name="test" class="br.com.rise.event.test.EventTest"
        init-method="start" destroy-method="stop">
        <property name="eventService" ref="eventServiceBean" />
        <property name="configurator" ref="propsBean" />
    </bean>
</beans>

**c) Encapsulating variability within a service**

This variability mechanism isolates core service functionality from aspects that are either high changeable, or in the case of an SPL, potential variation points (Smith and Lewis, 2009). In order to allow this mechanism, service-orientation aspects may be used for decomposing these variation points into services. Consequently, service providers
are exchanged, their invocation order is changed, or a specific service call may be optional with the purpose of implementing variability. In the consumer side, there are two alternatives to perform the binding of the service providers: (i) consume the service separate as bundles, or, (ii) call the services implementing each variant directly in the source code of the consumers. However, there is a drawback in the second alternative, since it requires the source code modification every time it needs a different variant, i.e., a different service.

Therefore, the first alternative will be considered in the approach. Hence, the submission example presented in the binding variants within services mechanism previously discussed will be used to exemplify this scenario. As a consequence, variants are encapsulated in services and exposed in which of one variant must be used. Moreover, it can be extended to support, for example, the use of negative variabilities concept (Coplien, 2000; Muthig and Patzke, 2002). For example, for maintenance reasons, the developer or the domain architect wants to exclude the complete submission without affecting the entire application. This variability mechanism may be appropriate to implement this issue with the support of the Sprin’s OSGi module as exemplified in the following code snippet 5.5.

Listing 5.5 Encapsulating variability within a service example

```java
encoding="UTF-8"?>
Example of CommonPoint.java
// The Common point is important to centralize the variants. This centralization
// will enable that the consumer use the variation point service without knowing
// its internal behavior
public abstract class CommonPoint implements ICommonPoint { [...] 
    public abstract ISubmission submit(IDocument document, String title,
        String keywords, Set<IFile> files); [...] 
}

Example of SubmissionServiceImpl.java
// The SubmissionService is the intermediate responsible to inject the
// correspondent variant that other service or application may require
public class SubmissionServiceImpl implements ISubmissionService { [...] 
    private ICommonPoint commonPoint; [...] 
    // Constructor injection Common point
    public SubmissionServiceImpl(ICommonPoint commonPoint){
        this.commonPoint = commonPoint;
    }
```
public ISubmission submit(IDocument document, String title,
String keywords, Set<IFile> files) {

[...] // Send confirmation for authors
}

Example of submissionservice-osgi.xml

<beans> [...] <osgi:reference id="completeSubmissionOSGi"
interface="br.com.rise.puresubmission.common.ICommonPoint" /> [...] </beans>

Example of CompleteSubmissionImpl.java

// The complete submission variant is decomposed as a service and must
// implements the Common Point class.
public class CompleteSubmissionImpl extends CommonPoint { [...] public ISubmission submit(IDocument document, String title,
String keywords, Set<IFile> files) {
    // Complete submission algorithm [...] }
}

Example of completesubmission-osgi.xml

<beans> [...] <osgi:service id="completeSubmissionOSGi" ref="completeSubmissionBean"
interface="br.com.rise.puresubmission.common.ICommonPoint" /> [...] </beans>

**Deploy Service and Component.** In this task, the goal is to build and install the services and components. This is valid for the both service providers and consumers. It involves compiling and packaging the services and components in a form that is suitable to be deployed in the production environment. The bundles compiled are the main output of the service-oriented domain implementation. They can be used as input, for instance, in the context of domain test discipline. It is important to stress that all the bundles generated must be running in an OSGi container appropriate such as Equinox (Equinox, 2009), OSCAR (Oscar, 2005), or Knopflerfish (Knopflerfish, 2005). The container used to implement the conference domain in this approach was the Equinox. Equinox is a simple OSGi container in which is possible to manage the bundles in a visual way, providing the integration with the Eclipse IDE platform (Eclipse, 2009) used to develop the entire conference application. In addition, Equinox supports the OSGi release 4, in contrast
with OSCAR and Knopflerfish in which support the release 3.

In the context of the SpringDM, for example, the developer has the possibility to use the Apache Tomcat (Apache, 2008) or Glassfish (GlassFish, 2009) web containers integrated with any OSGi container to expose the bundles, in particular the OSGi services that may be used in the final application or to be reused by other applications. Such integration between SpringDM and these web containers is out of scope of the approach. For more information, the Spring DM reference guide (Spring, 2008) is the most appropriate resource.

5.4.2 Assess Services and Components

It is a simple activity whose goal is to perform the assessment through software metrics of the services and components implemented in the first activity. In order to perform such assessment, the *assess services and components activity* receives the list of services and components implemented and the assessment framework defined in Chapter 4. Moreover, the developers and the domain architects are roles to perform this activity. Figure 5.11 shows the inputs, outputs, and roles of this activity. It is important to mention that this activity does not focus on services and components refinement, since it is performed in the activity described in Subsection 5.4.3. However, the assessment is important to provide for the developer directions to improve the implementation and consequently impacting in the refinement activity.

![Figure 5.11 Assess Services and Components Activity.](image)

The assessment framework is an important input for this activity because it contains the criteria that software engineers must apply to verify whether the services and components were implemented with a certain level of quality. In this way, the developers or domain architects must compare the assessment of the current domain implementation with the assessment procuced, for example, in the case study. Furthermore, the criteria depicted in the assessment were defined based on a set of software metrics used to assess
the domain developed for the study. The basic set of metrics defined is related to CBCS, LCOO, IMSC, CC, ESLOC, WOCS and CDCS measures, which are concerning with modularity, stability, complexity, and understandability criteria.

The assessment data collection is performed by the developer or the domain architect, which must select the appropriate software metric tool or collect in his own way, for example, through scripts. Indeed, what is important is that the data must be collected carefully, as well as, its comparison in order to achieve the properly benefits. Accordingly, the main output of this activity is an assessment framework refined or improved, whose objective is to store a certain quantity of data for further comparisons in the context of service-oriented product lines projects. In the next section, the Refine Services and Components Specification is described. This activity is also important to refine the current services and components implementation and the services and components specification.

5.4.3 Refine Services and Components Specification

The main goal of this activity is to perform the refinement of the detailed design when it is necessary. For this purpose, it receives the SO-DDSA, the business process models, and the feature model as the main inputs. The domain architect and the developers are responsible to perform this activity. Nevertheless, in some cases, the business analyst can be required when there are some inconsistent business rules and the changes in the business process models become essential. As a consequence, the changes in the architecture specification and implementation activities are also unavoidable. The main task of this activity is the update services and components specification. Figure 5.12 shows the inputs, outputs, tasks and roles of this activity.

![Figure 5.12 Refine Services and Components Specification Activity.](image)

Update Services and Components Specification. One of the reasons that can be pointed out concerning the architecture specification refinement can be, for example,
5.5. CHAPTER SUMMARY

5.5 Chapter Summary

This chapter presented the proposed approach to implement core assets in service-oriented product lines as well as its principles, roles, activities, tasks, inputs and outputs. The approach is divided in three activities: implement services and components, refine services and components specification, and assess services and components. During the implement services and components activity, it was proposed guidelines for describing, implementing services and components, variability management, and deployment in the context of

granularity reasons that may affect services and components modularity and complexity. For instance, when internal variability are identified when the implementation phase has been started. It is a practical scenario, since the business activities are always changing, and according to customer needs, it can imply in additional resources or the removal of existing ones. In this sense, the software engineer notifies the domain architect and redefines the internal components or service structure through structural models. In order to guide both domain architect and developer, the decision model defined in Chapter 4 may be used as an input to overcome this situation. The decision model can aid the developers and domain architects offering variability mechanisms according to some criteria for a particular technology.

Furthermore, other issues may emerge in the level of communication and messages exchanged among service consumers and providers. For instance, suppose a domain where applications require that OSGi services communicate with services of other application(s) in the same organization (service orchestrations). Thus, the domain architect must define the adequate model to provide necessary resources that must be used with OSGi to guide the software engineer in the implementation step. Such situation may have impact in the integration view defined in the SO-DDSA, and, in this case, it must be refined to support such technological issues. Moreover, it will imply also in a refinement of the current code implemented.

For example, the submission example presented in the encapsulating variability within a service mechanism definition, it was necessary the inclusion of a communication level between the submission service and the variable services (complete and partial submissions) to make a service composition. In order to apply such scenario we needed to refine the architecture with the purpose to document this issue for further products. Hence, this task is essential and must be applied when it is required, usually in parallel with the implement services and components activity.
Spring’s OSGi environment. Moreover, it the variability management was conducted with respect to specific feature types, service-oriented and component-based concerns, and, quality attributes based on the decision model depicted in Chapter 4.

The refine services and components specification activity presented how some situations can lead to the architecture specification refinement with respect to technological concerns and variability management issues. Moreover, some solutions were discussed in order to aid developers and domain architects to deal with such situations. The main output of this activity is the SO-DDSA updated, as well as, the services and components implemented. In the last activity, the goal was to assess the services and components through a set of software metrics based on the criteria depicted in the decision model in order to aid the developers to properly verify the quality of the services and components implementation. The main output of this activity is an assessment framework composed of a data collection. This collection may be used in a comparison model in further applications that is being developed.

In the next chapter, it will be presented a case study using the service-oriented product lines implementation approach. The case study will describe its definition, planning, analysis and interpretation of the results.
6

SOINGLE-IM Evaluation

Simplicity is prerequisite for reliability.
—EDSGER DIJKSTRA

6.1 Introduction

In the previous chapter, we defined an approach to implement core assets in service-oriented product lines using Open Services Gateway Initiative (OSGi) as the technology to control the life-cycle of the assets. As a consequence, once a new service-oriented product line implementation approach has been described, an evolution must be performed to understand and evaluate if it achieves its proposed objectives. In this sense, this chapter presents a case study to evaluate the proposed approach with respect to some criteria, such as modularity, complexity, stability, understandability, and effort.

In order to conduct this case study, we organized the chapter following the concepts proposed by (Kitchenham et al., 1995; Kitchenham and Pickard, 1998a; Wohlin et al., 2000). Thus, Section 6.2 presents the case study definition, which defines the objectives, hypotheses, and the pilot project used in the study. In Section 6.3 the planning of the case study takes place. Section 6.4 shows the analysis and the interpretation of the results. Section 6.5 describes the lessons learned during the study, and, finally, Section 6.6 summarizes this Chapter.

6.2 The Definition

In order to define the case study, it is important to state the context where it should be applied, as well as, the hypotheses that must be detailed enough to make clear what
measurements are needed to demonstrate the expected effect according to the context, and the pilot project, which defines the domain used in the study.

### 6.2.1 Context

The context was performed following the Goal Question Metric (GQM) methodology (Basili et al., 1994). The GQM is based upon the assumption that for an organization to measure in a purposeful way, it must first specify the goals for itself and its projects, and then it must trace those goals to the data that are intended to define it operationally. Finally, it must provide a framework for interpreting the data with respect to the stated goals. Thus, the GQM can be defined as follows.

**Goal.** Analyze the service-oriented product lines implementation approach with the purpose of evaluating it with respect to efficiency, and effort from the point of view of the software engineer and researchers in the context of service-oriented product lines projects.

**Questions.** In order to achieve this goal, we defined quantitative questions. It is related to the data collected during the period that the case study will be performed. The questions are described as follow.

**Q1.** *Does the approach generate assets with low complexity?* The main objective of the approach is to reduce the complexity of services and components implementation with OSGi in a systematic way. Consequently, we would like to understand if this goal would be achieved.

**Q2.** *Does the approach generate assets with low coupling?* A SOA application enables the loose coupling among service consumers and providers. Hence, we want to investigate and understand if the product lines aspects concerning to variability and commonality combined with SOA design style will affect the coupling among the services and components.

**Q3.** *Does the approach generate assets with high cohesion?* The same assumption aforementioned in the coupling question will be investigated in services and components cohesion.

**Q4.** *Does the approach generate assets with low instability?* The more coupled are the services and components, the more instable it will be. Thus, this question aims to
answer such issue.

Q5. Does the approach generate assets respecting the separation of concerns principle? OSGi separates concerns into bundles comprising not just application code, but also other resources that together can provide services and components to other bundles. In this way, this question aims to understand and answer it.

Q6. What is the effort during the implementation with the use of the approach? This question aims to investigate the effort that will be taking with the use of the approach.

**Metrics.** The purpose of using metrics in this case study is to assess software attributes such as size, coupling, cohesion, instability, separation of concerns, and effort for writing code. The metrics and the reason for they choice were defined earlier in the case study described in Chapter 4. However, the only difference here is that it will be considered the effort attribute. These metrics are relevant for this case study, since both were evaluated in the context of a SPL adapted to support service-orientation aspects (e.g., the decomposition of features into services and components, and the entire service provider side implementation) providing useful results for comparison with other baselines, for example. In addition, the metrics were evaluated in the context of a service-oriented product line using three technologies (OSGi, Apache Tuscany and Java API for XML Web Services (JAX-WS)) in the implementation phase.

Figure 6.1 presents the measurement framework defined in Chapter 4 including the attribute related to the hours spent writing code in the effort as attribute. These attributes are associated with every question to answer it objectively in a quantitative way, for example, code quality and productivity.

In order to answer the Question 6, the effort metric was defined following the assumptions described in (Basili et al., 1996) work, which is related to the hours spent to writing code and testing individual components of the system. Thus, in this case study, this metrics can be defined as follows.

Q6. What is the effort during the implementation with the use of the approach?

- **M_E: Effort for writing code (Number of Hours Spent).** This metric can be defined in terms of the Development Effort (DE) metric defined in (Basili et al., 1996) work. However, it was adapted for the context of this study in order to support the service-oriented product lines design and implementation, which increases the complexity of the application developed. Moreover, this
6.2. THE DEFINITION

Figure 6.1 Measurement Framework.

The study is interested in analyzing the effort for implementing not only individual pieces of the application, but also the relationship among the assets produced. Thus, this metric can be assessed as:

\[ \text{Effort}(\text{NHS}) = \text{DE}(T). \]

Where \( \text{DE}(T) \) is defined as the total number of hours the software engineer spent implementing the assets with a Technology \( T \).

6.2.2 Hypotheses

When specifying the hypotheses, the goals of the case study must be defined, as well as, the methods or technologies that will be evaluated (i.e., treatments). For this purpose, the
hypotheses can be defined in terms of null hypotheses and alternative hypotheses.

**Null hypotheses.** In this study, the null hypotheses determine that the use of the approach in the context of a simple service-oriented product lines project do not produce benefits that justify its use. Hence, according to the selected criteria, the following hypotheses can be defined:

\[
H_0 : \begin{align*}
\mu_{\text{approach WOCs}} & \geq 10 \\
\mu_{\text{approach CC}} & \geq 11 \\
\mu_{\text{approach CBCS}} & \geq 6 \\
\mu_{\text{approach LCOO}} & \leq 10 \\
\mu_{\text{approach IMSC}} & \geq 0.5
\end{align*}
\]

The values used for the hypotheses definition were defined in the metrics description (see Chapter 4). It is important to mention that these values were applied in different contexts and used in many empiricall studies as described in Chapter 4.

**Alternative hypotheses.** These are the hypotheses in favor of which the null hypotheses are rejected. In this study, the alternative hypotheses determine that the use of the approach applied with a simple application produces benefits that justify its use. Thus, the following hypotheses can be defined:

\[
H_1 : \begin{align*}
\mu_{\text{approach WOCs}} & < 10 \\
\mu_{\text{approach CC}} & < 11 \\
\mu_{\text{approach CBCS}} & < 6 \\
\mu_{\text{approach LCOO}} & > 10 \\
\mu_{\text{approach IMSC}} & < 0.5
\end{align*}
\]

### 6.2.3 Pilot Project

The case study presented was based on the development of a service-oriented product line in the domain of travel reservations, herein named travel agency product line. The travel reservation domain is similar to the libraries domain (aka Rental for All Software Factory (R4ll)) described in Chapter 4. In R4ll, the main goal consisted of the management of items, for example, books, theses, multimedia materials, among
6.3 THE PLANNING

After the definition of the case study, the planning takes place. On the one hand, the definition determines the foundation of the study as well as the reason for performing it. On the other hand, the planning prepares for how the case study is conducted. In this sense, the objective of this study is to evaluate the viability of using the service-oriented product line implementation approach based on a service-oriented product line project. The project will be conducted in a university lab, more specifically in off-line mode, since it is not a project developed in the industry. This study can be classified as a quasi-experiment, since there is a lack of randomization of either selection of objects or subjects (Wohlin et al., 2000). Moreover, it can be classified as a single object study, which is characterized as being a study that examines an object on a single team and a single project (Wohlin et al., 2000). However, the term single team will be modified to “single subject”, since the case study is performed by only one subject.

Training requirements. It is assumed that the subjects are already familiarized with concepts related to software reuse, such as, variability, service-oriented development, and software product lines, then, they can understand the proposed approach by only reading it, without the necessity of training.

Subjects. The subjects of the study will be requested to act as the roles defined
6.3. THE PLANNING

in the approach (software engineers). Nevertheless, during the project, the subjects can have more than one role depending on the context, e.g., domain analyst in the beginning and domain architect in the next step.

Quantitative analysis mechanisms. The quantitative analysis was divided as complexity, modularity, stability, understandability and effort for the implementation steps. It is important to stress that these quality factors are predictors for reusability and maintainability as defined in Chapter 4. Hence, the analyses will also cover these aspects and will be performed using descriptive statistics, such as mean, maximum, minimum, standard deviation, line charts, box plot graphics, and scatter charts.

Randomization. According to (Wohlin et al., 2000), this technique can be used in the selection of the subjects. Ideally, the subjects must be selected randomly from a set of candidates. However, in this study, as it was defined to be applied by only one subject, it was not necessary the randomization, once the subject has expertise in the approach domain. On the other hand, for a team of subjects it is expected that they also have expertise in the service-oriented product line domain, which is the focus of the approach.

Blocking. According to (Wohlin et al., 2000), if the effect on the factor is known and controllable, is possible to use a design technique called blocking. Blocking is used to systematically eliminate the undesired effect in the comparison among the treatments. In this study, it was not need the separation of the subjects into blocks, since the study will evaluate just one factor, which is the use of the approach, and it will be performed by only one subject who has previous knowledge on the approach.

Balancing. In some case studies, balancing is desirable because it simplifies and strengthens the statistical analysis of the data. However, in this study it is not necessary to divide the subjects, since there is only one subject.

Measures. In order to analyze the approach efficiency with regard to the objective defined, some criteria were used to evaluate the benefits with the usage of the approach. The benefits will be evaluated quantitatively through the code quality, using reusability, maintainability, complexity, stability, modularity, and understandability criteria. In addition, it was defined a measure responsible to evaluate the effort of the software engineer to implement the travel agency project using the approach. Figure 6.1 depicts the metrics related for each criterion aforementioned.

Data collection procedures. The case study was performed during January 2010, at the Federal University of Pernambuco, Brazil. The study was composed of one subject and the project was developed on 152 hours and 24 minutes. The project
was implemented using the service-oriented product line implementation approach with
the support of OSGi as the main technology. Table 6.1 summarizes the total data (e.g.,
number of services, number of components, packages, number of classes, and the total
lines of code (LOC)) for the implementation of the travel agency product line project.
In this sense, several metrics data were collected concerning quality factors such as
reusability, maintainability, complexity, modularity, stability, understandability, and
effort metrics.

Table 6.1 General data collected during the case study.

<table>
<thead>
<tr>
<th>Service</th>
<th>LOC</th>
<th>Packages</th>
<th>Classes</th>
<th>Interfaces</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>AirlineService</td>
<td>101</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>BankBilletComponent</td>
<td>245</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>34</td>
</tr>
<tr>
<td>CoreAgencyComponent</td>
<td>936</td>
<td>8</td>
<td>18</td>
<td>8</td>
<td>137</td>
</tr>
<tr>
<td>CreditCardComponent</td>
<td>334</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>DataAccessComponent</td>
<td>245</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>HotelService</td>
<td>338</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>57</td>
</tr>
<tr>
<td>NotificationService</td>
<td>149</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>PaymentService</td>
<td>474</td>
<td>8</td>
<td>9</td>
<td>5</td>
<td>120</td>
</tr>
<tr>
<td>BookingService</td>
<td>244</td>
<td>7</td>
<td>8</td>
<td>4</td>
<td>45</td>
</tr>
<tr>
<td>UtilitiesComponent</td>
<td>733</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>44</td>
</tr>
<tr>
<td>VehicleService</td>
<td>156</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>4,076</td>
<td>54</td>
<td>78</td>
<td>33</td>
<td>547</td>
</tr>
</tbody>
</table>

| Number of Services    | 6    |
| Number of Components  | 5    |

**Internal validity.** The internal validity of the study is defined as the capacity of
a new study to repeat the behavior of the current study, with the same subjects and objects
with which it was executed (Wohlin et al., 2000). The internal validity of the study is
independent of the number of subjects. This study is supposed to have one subject or at
least eight subjects to guarantee a good internal validity. The selection of one subject
depends of the size of the domain analyzed. In this case study, only one subject was
chosen, since the size of the domain chosen was small, and due to time restrictions. In
this sense, the selection of one subject is important to collect an initial set of results.

**External validity.** In this study, a possible problem related to the external
validity is the subject’s choice, since only one subject is responsible to perform the
study. Accordingly, the selection of one subject can make the study biased, since he
participated in the definition of the approach being evaluated. However, the external
validity of the study may be considered sufficient, since it aims to evaluate the viability
of the approach, implying that even with one subject performing it, the data must be considered for further analysis and replication of the study in the context of a team of subjects. As a consequence, new studies can be planned in order to refine and improve the approach.

**Conclusion validity.** The analysis and interpretation of the results of this case study will be described using descriptive statistics, which are appropriate for the data type collected during the study.

### 6.4 The Analysis

After collecting data of the artifacts produced in the implementation phase, we want to be able to draw conclusions based on this data. The analysis was organized based on the quantitative analysis mechanisms defined in the planning phase (Section 6.3). Hence, the analysis and interpretation of the data are presented as follow.

#### 6.4.1 Complexity Analysis

After collecting the information about the assets (services and components) complexity, the step of data set reduction started. It is important because errors in the data set can occur either as systematic errors or as outliers, which means that the data point is much larger or much smaller than one could expect looking at the other data points (Wohlin et al., 2000). In this way, we decided to organize this analysis according to the metrics defined to measure the complexity factor, such as **Cyclomatic Complexity (CC)** and **Weighted Operations per Component or Service (WOCS)**. Moreover, we discussed the descriptive statistics to analyze the null hypotheses described in the earlier sections. Consequently, these aspects were valid and used for analyzing and interpreting the results of the other quality factors, such as **modularity**, **stability**, and **understandability**.

**Cyclomatic Complexity Analysis.** The CC analysis consisted in analyzing the services and components complexity. This aspect is important because it can identify services or components that will be difficult to test, reuse or maintain. Therefore, it is important to implement services and components with low complexity.

*Descriptive Statistical Analysis.** Figure 6.2, shows the cyclomatic complexity data graphically. As it can be seen in Figure 6.2, **NotificationService** presented a low value (2), when compared with the other data points. In this way, the value could be considered as an outlier. In order to analyze this aspect, a box plot graphic can be useful.
6.4. THE ANALYSIS

(Fenton and Pfleeger, 1998), since it is recommended to visualize the dispersion and skewedness of samples.

![Cyclomatic Complexity Graphic](image1)

**Figure 6.2** Cyclomatic Complexity Graphic.

Figure 6.3 shows the CC box plot graphic. The middle bar in the box is the median. The lower quartile $q_1$, is the 25% percentile (the median of the values that are less than median), and the upper quartile $q_3$ is the 75% percentile (the median of the values that are greater than median). The length of the box is $d = q_3 - q_1$. The tails of the box represent the theoretical bound within all data points are likely to be found if the distribution is normal. The upper tail is $q_3 + 1.5d$ and the lower tail is $q_1 - 1.5d$. In this sense, values outside the upper and lower tails are the outliers. Figure 6.4 shows the outliers analysis for the CC represented in a scatter graphic.

![Cyclomatic Complexity Box Plot Graphic](image2)

**Figure 6.3** Cyclomatic Complexity Box Plot Graphic.
6.4. THE ANALYSIS

Figure 6.4 Outlier Analysis for Cyclomatic Complexity.

As it can be seen in the Figure, NotificationService represent a suspect outlier. When outliers are identified, it is necessary to decide what to do with them: exclude the data or include it in the analysis. In this case, the data was not considered, since in the travel agency domain, the NotificationService provides simple operations, resulting in a low complexity.

**Weighted Operations per Component or Service.** The WOCS consisted in analyzing the services and components complexity as the cyclomatic complexity metric. Reuse and maintain services/components with overrated operations that are shared with other components or services can be a difficult task and can be error-prone. Thus, it is desirable that the operations provided by services and components have a low complexity.

**Descriptive Statistical Analysis.** Figure 6.5 shows the weighted operations per component or services data graphically. It can be observed that NotificationService and PaymentService presented low values (4) and high (18), respectively, when compared to the other data points. Consequently, as stated earlier in the cyclomatic complexity analysis, a box plot graphic can be useful to analyze these values, since they could be considered possible outliers. Figure 6.6 shows the WOCS box plot graphic with its information.

As can be seen in Figure 6.6, the values outside the upper and lower tails are the outliers, respectively the min (NotificationService) and the max (PaymentService). Figure 6.7 shows the outliers analysis for the WOCS metric. As stated in the CC analysis, when outliers are identified it is important to decide what to do with them. In this case, we considered the PaymentService data and exclude the NotificationService. The reason for the exclusion of NotificationService data is that it is a simple service responsible to sending messages. It is important to highlight that the NotificationService contains an
internal variability, i.e., the type of notification, which can be notification by SMS or Email.

Even with the implementation of this variability, the internal implementation of both SMS and Email was simple, since we only had to define a common operation in an interface, and each variant was responsible to implement its own sending message algorithm through this interface with the support of their own internal operations. Hence, the interface shared with the NotificationService becomes simple as well. On the other hand, the PaymentService is responsible to communicate with an external service, called as CreditCard and implement an internal component BankBillet. In order to consume the CreditCard service, it was defined several operations with some formal parameters to
conform the contract provided by the external service. Furthermore, for the bank billet operations, it was necessary some particular operations with many formal parameters.

![Figure 6.7 Outlier Analysis for Weighted Operations per Component or Service.](image)

**Figure 6.7 Outlier Analysis for Weighted Operations per Component or Service.**

Table 6.2 shows the statistical data collected during the case study. The statistics present some relevant information for analyses. The CC mean (7.27) rejects the null hypothesis, and the WOCS mean (9.45) also rejects the null hypothesis. It indicates that the approach aids in producing services and components with a low complexity according to the values reported in Chapter 4 for a simple application. Moreover, all the services and components, except by PaymentService, present the complexity value below the null hypothesis for WOCS, reinforcing the premises that the approach allows to implement artifacts with low complexity.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Cyclomatic Complexity (CC)</th>
<th>Weighted Operation per Component or Service (WOCS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>7.27</td>
<td>9.45</td>
</tr>
<tr>
<td>Maximum</td>
<td>11</td>
<td>18</td>
</tr>
<tr>
<td>Minimum</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>2.572</td>
<td>3.587</td>
</tr>
<tr>
<td>Null Hypotheses</td>
<td>$H_0$: CC &gt;= 11</td>
<td>$H_0$: WOCS &gt;= 10</td>
</tr>
</tbody>
</table>

**6.4.2 Modularity Analysis**

This section involves the analysis of the modularity criterion through the Coupling between Components and Services (CBCS) and Lack of Cohesion Over Operations (LCOO) metrics.
6.4. THE ANALYSIS

Coupling between Components and Services. CBC consists in analyzing the coupling among services or/and components. It is desirable a low coupling for services or components communication among each other.

Descriptive Statistical Analysis. Figure 6.8 shows the coupling between components and services data graphically. It can be observed that DataAccessComponent, UtilitiesComponent, and PaymentService presented high (5), low (1), and low (1) values, respectively, when compared to the other data points. Thus, a box plot graphic can be useful to analyze these values, since they could be considered possible outliers. Figure 6.9 shows the CBC box plot graphic with its information.

![Coupling Between Components and Services Graphic](image)

**Figure 6.8** Coupling between Components and Services Graphic.

![Coupling Between Components and Services Box Plot Graphic](image)

**Figure 6.9** Coupling between Components and Services Box Plot Graphic.

As can be seen in Figure 6.9, the value outside the upper tail is the outlier, respectively the max (DataAccessComponent). Figure 6.10 shows the outliers analysis for the CBC
metric. In this case, the data was considered, since in the travel agency domain, persistence is the most intensive task that is performed. In this way, the *DataAccessComponent* has operations that are used in the most components and services of the travel agency. However, such operations are simple and do not impact in the system reusability and maintenance.

![Coupling Between Components and Services](image)

**Figure 6.10** Outlier Analysis for Coupling between Components and Services.

**Lack of Cohesion over Operations.** LCOO consists in analyzing the lack of cohesion over operations provided by services or components. The LCOO value provides a measure of the relative disparate nature of the methods in a service or component. According to the LCOO definition (see Chapter 4), a smaller number of disjoint pairs implies greater similarity of methods.

**Descriptive Statistical Analysis.** Figure 6.11 shows the lack of cohesion over operations data graphically. It can be observed that *HotelService* and *BankBilletComponent* presented high (72) and low (0) values, respectively, when compared to the other data points. Figure 6.12 shows LCOO box plot graphic in order to identify possible outliers.

As can be seen in Figure 6.12, the value outside the upper tail is the outlier, respectively the max (*HotelService*). Figure 6.13 shows the outliers analysis for the LCOO metric. In this case, we considered the *HotelService* analysis, since in the travel agency domain, *hotel booking* was the entire booking functionality implemented. The high value for *HotelService* is a response of its implementation that does not have similar methods, leading in a low cohesion.

Table 6.3 shows the statistical data collected in which present some relevant information for analyses. The Coupling between Components and Services (CBCS) mean (2.36) rejects the null hypothesis, and the Lack of Cohesion Over Operations (LCOO) mean
6.4. THE ANALYSIS

Figure 6.11 Lack of Cohesion over Operations Graphic.

Figure 6.12 Lack of Cohesion over Operations Box Plot Graphic.

Figure 6.13 Outlier Analysis for Lack of Cohesion over Operations.

(16.63) also rejects the null hypothesis. It indicates that the approach aids in producing services and components with a good modularity.
6.4.3 Stability Analysis

In order to conduct the analysis concerning the stability factor, we decided to organize this section according to the Instability Metric for Service or Component (IMSC) metric.

**Descriptive Statistical Analysis.** Figure 6.14 shows the instability metric for service or component data graphically. As can be seen in the Figure, DataAccessComponent and HotelService presented values low (0.214) and high (0.967), respectively, when compared to the other data points. In this sense, a box plot graphic can be useful to analyze these values, since they could be considered possible outliers. Figure 6.15 shows the IMSC box plot graphic with its information.

![Instability Metric for Service or Component](image)

**Figure 6.14** Instability Metric for Service or Component Graphic.

As can be seen in Figure 6.15, the values outside the upper and lower tails are the outliers, respectively the min (DataAccessComponent) and the max (HotelService). Figure 6.16 shows the outliers analysis for the instability metric for service or component metric. In this case, the both data were considered, since in the travel agency domain, hotel booking and persistence are the most intensive tasks that are performed. In this sense, the HotelService service had to be highly coupled with the booking service, resulting in a
6.4. THE ANALYSIS

Figure 6.15 Instability Metric for Service or Component Box Plot Graphic.

high instability of the service. On the other hand, \textit{DataAccessComponent} is a simple and independent component, which only provides specific persistence operations according to the requirements of the components or services.

Figure 6.16 Outlier Analysis for Instability Metric for Service or Component.

As can be seen in Table 6.4, the IMSC mean (0.6) do not reject the null hypothesis, indicating that the approach does not produce artifacts with a good stability. Moreover, only three artifacts (\textit{DataAccessComponent}, \textit{NotificationService}, and \textit{UtilitiesComponent}) present the instability value below the null hypothesis, since both are simple services or components that do not need to be highly coupled among others. On the other hand, four services concerning booking, such as (\textit{AirlineService}, \textit{HotelService}, \textit{BookingService}, and \textit{VehicleService}) were pointed out with high values. These values were expected, since as stated earlier, booking is an intensive task that must be highly coupled among each other. Perhaps the refactoring of \textit{HotelService} in more specialized services enabling service-orchestration, for example, may take acceptable values.
6.4.4 Understandability Analysis

This section presents the discussion with respect to the understandability criterion results. It is important to mention that the metrics for Separation of Concerns (SoC) generate values per concern, differently of the metrics for coupling, cohesion, size and instability that generate values per component. Accordingly, in order to measure the Concern Diffusion over Component and Services metric, we considered the concerns, such as AirlineService, BankBilletComponent, CreditCardComponent, HotelService, and VehicleService. Table 6.5 shows the concerns data collected during the case study.

As can be seen in the table, the values are low, which implies that the concerns are not much spread through other services and components. Therefore, the services and components are well localized in the system, facilitating their maintenance. However, these values may increase as the system is incremented with new features. Thus, this mapping of concerns is necessary to aid developers and domain architects to visualize if the existing features were modified and to map new possible concerns. The Access Services and Components activity defined in the proposed approach (see Chapter 5), is important in order to identify new refinements in the implementation phase.

Table 6.5 Results for the Understandability Analysis.

<table>
<thead>
<tr>
<th>Concern Diffusion over Components and Services (CDCS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artifacts</td>
</tr>
<tr>
<td>--------------------</td>
</tr>
<tr>
<td>AirlineService</td>
</tr>
<tr>
<td>BankBilletComponent</td>
</tr>
<tr>
<td>CreditCardComponent</td>
</tr>
<tr>
<td>HotelService</td>
</tr>
<tr>
<td>VehicleService</td>
</tr>
</tbody>
</table>
6.4.5 Effort Analysis

This section presents the analysis with respect to the effort criterion. In the context of such criterion, the data collected corresponds to the \( Effort_{\text{Metric}} (T) \) metric defined in subsection 6.2.1. For collecting the metric were considered the time spent in hours for developing the assets corresponding to the travel agency domain (see Appendix B for more information) using the OSGi technology. In order to analyze the effort measure, we decide to count the time for the implementation of each service and component developed as can be seen in Table 6.6.

<table>
<thead>
<tr>
<th>Assets</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>AirlineService</td>
<td>22</td>
</tr>
<tr>
<td>BankBilletComponent</td>
<td>7</td>
</tr>
<tr>
<td>CoreAgencyComponent</td>
<td>13</td>
</tr>
<tr>
<td>CreditCardComponent</td>
<td>9</td>
</tr>
<tr>
<td>DataAccessComponent</td>
<td>3</td>
</tr>
<tr>
<td>HotelService</td>
<td>40</td>
</tr>
<tr>
<td>NotificationService</td>
<td>4</td>
</tr>
<tr>
<td>PaymentService</td>
<td>7</td>
</tr>
<tr>
<td>BookingService</td>
<td>34</td>
</tr>
<tr>
<td>VehicleService</td>
<td>9</td>
</tr>
</tbody>
</table>

In this way, some drawbacks could be identified with the use of the approach. It is reflected by the technology specification that is quite extensive and requires a longer time for a deeper reading, including the lecture of the approach. Furthermore, the travel agency domain did not provide a well-defined documentation, difficulting in the understanding of some service usages, such as Hotel, Bank, Airline and Payment. The Hotel and Airline services took a considerable effort in the implementation, since it requires a lot of information to be developed and many integration that must be performed in order to consume external services. As we did not use any real agency, toy examples were created to simulate the consumed external services. Another factor was the decomposition of these services using variability and commonalities analysis according to the variability mechanisms applied in this context.

The Payment and Bank were developed with simple variability mechanisms described in Appendix B, but the more extra effort were quite complex operations for developing the bank billet, credit card components and the application of both with SpringDM constraints. However, we can not trace conclusive results with the effort in using the approach by taking this kind of measure. Thus, it is clear that the refinement of such measure and
the use of the approach in a real situation is needed in order to draw more conclusive results. Moreover, the inclusion of more subjects may allow the randomization of the study allowing the separation of subjects in teams. In this sense, different values are collected and compared for each team and the use of other approaches may also help to a construct validity.

6.5 Lessons Learned

After concluding the case study, some aspects should be considered in order to further repeat the study, since they were seen as limitations of the first execution.

**Domain.** The travel agency domain should be re-discussed since it is a non-trivial domain and often a domain expert is necessary to aid in the process. For example, as the domain where the application must communicate with external agencies, which offer services with different complexity levels, it is important that domain experts participate of the planning in order to solve possible problems of integration. Moreover, as it can be seen in Appendix B, only basic features were provided. Thus, the domain must support evolution, which will be reflected in the use of the approach for implementing such new features.

**Pilot Project.** The case study and the pilot project were performed just by one subject. This aspect should be reanalyzed in order to include more subjects that will execute the case study.

**Training.** As stated earlier, it is expected that more subjects can perform the case study. In this sense, training must be defined and should include a complete and detailed example covering all the aspects defined in the approach, such as service-oriented and software product lines concepts, variability implementation mechanisms specifics for service-oriented product lines domains, and OSGi concepts.

**Effort.** The effort measure was defined in a simple manner, thus implying in an inadequate and superficial analysis. Hence, this measure must be refined in a further study in order to achieve a proper investigation. It is important that more subjects participate in the case study for allowing the randomization and analyze the effort of different teams during the implementation.
6.6 Chapter Summary

This chapter presented a case study, following the organization proposed by (Kitchenham et al., 1995; Kitchenham and Pickard, 1998a; Wohlin et al., 2000), to evaluate the approach proposed, which handles the implementation of core assets (services and components) in the context of service-oriented software product lines. The study has analyzed a travel agency domain structured as a service-oriented software product lines based on commonality and variability analysis according to the specific characteristics of the domain. Moreover, it also analyzed the possibility of subjects using the approach to implement assets with good modularity, understandability, and low complexity implying on the good reusability of those services or components.

However, only few assets did not produce good results for stability according to the metric analysis. Nevertheless, it was expected, since such assets provides the most important tasks for the domain purposes. Furthermore, even with the reduced number of subjects (1), the analysis has shown that the service-oriented product line implementation approach can be viable in a simple application. It also identified some directions for improvements. However, two aspects should be considered: the study replication in different contexts and studies based on observation in order to identify problems and points for improvements. Moreover, the use of more subjects may allow the analysis of the approach usage with different teams.

The next chapter presents the conclusions of this work and some directions for future work.
Concluding Remarks and Future Work

*Science never solves a problem without creating ten more.*
—GEORGE BERNARD SHAW (1856 - 1950) (Nobel Prize in Literature (1925), Irish playwright)

Nowadays, Software Product Lines (SPL) are being explored in different domains and contexts. An example is the combination with the Service-Orientation (SO) approach. In this sense, this work presents a contribution for the combination of SPL and SO (service-oriented product lines), in particular, how these concepts can be used together to achieve desired benefits, such as, improve reuse, decrease development costs and time-to-market, and flexible applications which are customized to specific users or market segment needs.

In order to address the relationship between SPL and SO, it is important to deal with some challenges, such as, the complex activities concerning variability implementation, and technology specific concerns that are useful to manage specific variabilities from the core assets in service-oriented product lines applications.

In order to address understand and reduce the aforementioned issues, Chapter 4 presented an assessment through a case study performed to evaluate technologies (*OSGi*, *Apache Tuscany*, and *JAX-WS*) to implement variability in core assets with the support of variability mechanisms (*Binding variants within services, Parameters*, and *Strategy Design Pattern*). The case study was conducted with one subject, and the technologies were compared among each other. Moreover, the output of the case study was a *decision model* for guiding the software engineers on the task of choosing service technologies when dealing with issues related to variability management in core assets in service-oriented product lines projects. The decision model aims to recommend suitable technologies according to given inputs, such as some *criteria* (e.g., *complexity, modularity, stability,*
7.1 RESEARCH CONTRIBUTIONS

and understandability), variability mechanisms (e.g., Binding variants within service, Parameters and Strategy Design Pattern), and a catalog of technologies (e.g., OSGi, Apache Tuscany, and JAX-WS).

The quantitative analysis did not show many differences between the both technologies. Nevertheless, it was important for defining the decision model according to many points discussed during the analysis, showing that some technology(ies) fit best in particular cases. In this sense, OSGi was pointed out as a suitable technology for dealing with specifics situations, for example, in the decision model, the use of OSGi and the Parameters mechanisms indicates that it provides assets with a good modularity, stability and understandability. In order to understand the potential concerns and consequences of the service-oriented product lines implementation issues discussed during the case study performed in Chapter 4, it was developed an approach to implement core assets in service-oriented product lines using OSGi as the main technology.

In this sense, Chapter 5 presented an approach whose goal is to provide a systematic and practical way to implement core assets in service-oriented product lines using OSGi as the main technology to control the life-cycle of the core assets produced, and variability mechanisms for handling specific variation points. The approach uses a two life-cycle model as in SPL, and considers only core services development providing support for high customization and systematic planned reuse during service-oriented development. Moreover, the approach is based on a set of guidelines, inputs, and outputs, which are essential to provide a systematic guidance for software engineers specify the artifacts to be produced, and, define activities with practical scenarios.

The proposed approach was evaluated in a case study (Chapter 6), which presented the benefits, drawbacks, and directions for its improvement. Likewise, the case study has analyzed a travel agency domain structured as a service-oriented product lines based on commonality and variability analysis according to specific characteristics of the domain. It also analyzed the possibility of subject(s) using the approach to implement services and components with good modularity, understandability, and low complexity.

7.1 Research Contributions

The main contributions of this work can be split into the following aspects: (i) an assessment on technologies to implement variability in service-oriented product lines; (ii) the definition and systematization of an approach to implement core asset in service-oriented product lines; (iii) a case study in which the proposed approach was evaluated. These contributions are further described next.
An Assessment on Technologies to Implement Core Assets in Service-Oriented Product Lines. Through this assessment, three technologies (OSGi, Apache Tuscany, and JAX-WS) were evaluated in order to implement variability in core assets with the support of (Binding variants within services, Parameters, and Strategy Design Pattern) variability mechanisms. In addition, the output of the assessment was a decision model for guiding software engineering on the task of choosing service/component technologies when dealing with issues related to variability mechanisms in service-oriented product lines projects.

An Approach to Implement Core Assets in Service-Oriented Product Lines. After the assessment, its results were used as inputs for the definition of an approach to implement core assets in service-oriented product lines. It is defined in a systematic manner, with principles, roles, activities, tasks, inputs, outputs and guidelines with the support of practical scenarios addressed to the context of core assets implementation.

A Case Study on the proposed Approach. A case study was applied in an academic environment in order to evaluate the proposed approach. This initial validation of the service-oriented product line implementation approach helped in the identification of improvements, such as, the study replication in different contexts, the use of subject’s team (since the case study was executed with only one subject), and studies based on the lessons learned discussed.

7.2 Related Work

In the literature, several work, e.g., (Gacek and Anastasopoulos, 2001; Anastasopoulos and Muthig, 2004; Muthig and Patzke, 2002; Almeida et al., 2008) have discussed the implementation of core assets in SPL, and just a few work have considered the implementation for service-oriented product lines such as (Segura et al., 2007; Günther and Berger, 2008; Smith and Lewis, 2009).

7.2.1 Approaches for SPL Implementation

Anastasopoulos and Gacek in (Gacek and Anastasopoulos, 2001) addressed the issues of handling variability at the code level. For this purpose, they presented several variability mechanisms, e.g., aspect-oriented programming, conditional compilation, configuration files and design patterns, which can be applied in the SPL context. They also presented a model for making comparison of variability mechanisms based on the following feature types: positive, negative, optional and alternative. In addition, it compares the mecha-
nisms using criteria, such as binding time, scalability and traceability. However, the study does not perform a quantitative analysis.

In (Muthig and Patzke, 2002), variability mechanisms are analyzed from a component technology point of view. For this purpose, they surveyed some technologies, such as COM, DCOM, .NET, EJB and CORBA. The work integrates component technologies in a model-driven implementation process. It compares different component’s characteristics and relationships such as Composition, Interfaces, Interconnection, and Context Dependencies. The survey only describes each technology according to the aforementioned characteristics. Moreover, it provides a case study in which EJB is the only technology considered. However, no quantitative analysis, neither the usage of another technology to compare with EJB was presented.

Anastasopoulos and Muthig (Anastasopoulos and Muthig, 2004) presented a case study that was performed to evaluate Aspect-Oriented Programming (AOP) as a variability mechanism and AspectJ as a technology for implementing variability. In this work, they consider the technical factors that influence the implementation of variability during core assets development. The evaluation analysis identified that AOP is suitable for variability across several components, i.e., crosscutting concerns. However, the study did not present a quantitative analysis as well.

Almeida et al. in (Almeida et al., 2008), proposed a method whose goal is to implement core assets in the context of domain engineering using OSGi as the technology to control the assets life cycle. The method is defined in a systematic fashion through a series of activities based on the OSGi principles (modularity, loosing coupling, among others). However, many of the approach activities do not provide practical scenarios, and do not treat the implementation of variability. In contrast with our approach, the Almeida et al. method does not specify any variability mechanism or examples to guide the software engineer during the implementation.

### 7.2.2 Approaches for Service-Oriented Product Lines Implementation

Segura et al. (Segura et al., 2007) present a development approach in which they consider different feature types that can be applied to the web services technology, e.g., data flows, binding time and types of message. Thus, it provides a taxonomy of variability for web services flows that can be further applied in other projects. However, the taxonomy focus in variability managed by only one variability mechanism: program transformations. In this sense, it only describes the mechanism and do not present guidelines or practical
scenarios as examples, neither a quantitative analysis was performed. In contrast, our approach recommends three variability mechanisms for service-oriented product lines and presents a quantitative and qualitative analysis about the use of the approach with these mechanisms.

A development of an approach for web services in the SPL scenario is proposed in (Günther and Berger, 2008). It considers the implementation of service-oriented product line variability using the AHEAD tool suite (AHEAD, 2007) in the Web Store domain, allowing service interfaces to be modified or customized to specific contexts based on the features selected for each product. However, it does not consider the implementation of service variability using systematic activities and tasks, and focus only in code transformation tools as a variability mechanism.

In (Smith and Lewis, 2009), an approach is defined considering the use of SOA services as core assets in a SPL. The focus of such approach is to identify a small set of decisions that are required in the implementation of SOA SPL applications. Moreover, it considers the mapping of SOA concepts in the context of SPL and a set of variability mechanisms to deal with variabilities issues. However, there is a lack of details in how to achieve the decisions identified. For instance, the work does not provide examples on how to apply the variability mechanisms based on such decisions. Our approach define two variability mechanisms parameters and encapsulating variants within services mechanism which was proposed by Smith and Lewis. Nevertheless, we provide a practical scenario recommending this mechanism using specifics feature type (alternative and optional). Moreover, a quantitative and qualitative analysis is also performed including these mechanisms in the context of the application that was evaluated.

7.3 Future Work

Due to the time constraints imposed on a M.Sc. degree, this work can be seen as an initial step towards the efficient, usable and effective implementation of core assets in service-oriented product lines. In this way, there are interesting topics to improve what was started, and new paths to explore. Thus, the following issues should be investigated as future work:

**Product Derivation.** This work provides assets for supporting the product derivation. However, it does not define activities to perform development with reuse. Thus, this aspect must be addressed to cover the entire SPL lifecycle.

**Measurement framework.** This dissertation defined a measurement frame-
work for representing some criteria through metrics and its relationships to evaluate the technologies in Chapter 4, and the proposed approach used in the case study in Chapter 6. However, these metrics were used before in other contexts, therefore they needed to be refined for this study’s purpose. In this sense, this measurement framework could be also increased by several other metrics to measure more criteria (e.g., *scalability* and *portability*).

**Variability Mechanisms.** This work has considered the use of four variability mechanisms in the context of service-oriented product lines purpose. These mechanisms were: *Binding variants within services*, *Parameters and Configuration Files*, *Strategy Design Pattern*, and *Encapsulating variants within services*. However, it is important to highlight that the use of other variability mechanisms may be useful for service-oriented product lines, such as *Protocol Bridging*, *Program Transformations*, *Aspect-Oriented Programming*, *Generics* and many other *Design Patterns* (Gamma *et al.*, 1995). Moreover, *Asynchronous Queuing* is important for allowing service providers and consumers to process messages independently by remaining temporally decoupled.

**Combination of Technologies.** The combination of technologies enable a standardized way to integrate a variety of communication mechanisms in a runtime container. Moreover, integrating different implementation types allows the use of the technology best suited to the job. For example, the combination between OSGi and Rest Web Services can run on a wide range of system and has a clear and simple extension mechanism for supporting new implementation types and new communication bindings allowing the interoperability, which is a drawback in a pure OSGi implementation.

**Variation Points.** It is important to highlight that this work focus on variation points that can be employed during the service invocation, that is, the activity in which a service invokes other services to accomplish its goals through a service registry through the use a service technology. However, other variation points must be considered, in particular the *Service Contract Variability*, which is important to understand about services, priorities, responsibilities and warranties through the Service Level Agreement (SLA). Moreover, *Workflow Variability* must be addressed, since it focus on the service execution order, data exchange between participants, business rules and errors treatment. These aspects are harder to be mastered and must be performed in further studies.

**Decision Model.** The decision model defined in Chapter 4 was important to draw some decisions through a model representation in order to define suitable technologies according to a defined criteria and variability mechanisms. Nevertheless, it is important to state that with the application of the case study in other domains, new
variability mechanisms and other criteria may be applied to the defined decision model. Thus, it is important that the decision model support these new refinements.

**Application of the approach in other domains.** This dissertation presented the definition, planning, analysis and interpretation of a case study on the proposed approach. However, new case studies are necessary in conjunction with the industry, in different contexts, with a more significant number of subjects, in order to refine the approach and to improve the experiment design. We believe that more case studies must be performed, taking into account the lessons learned with the first case study, thus more concrete conclusions can be drawn.
Bibliography


Appendices
A.1 Project Context

This project aims to build a product line for the rental domain through the development of core assets to enable a rapid derivation of products. The project followed certain steps of the RIPLE (RiSE Product Line Engineering) process: RIPLE-RE as responsible for Requirements Engineering, RIPLE-DE responsible for the design of the product line, and finally the RIPLE-EM responsible for change management, embracing both engineering field, such as application. The implementation step was performed in the case study context defined in Chapter 4. The case study received as inputs the artifacts generated by the RIPLE-RE and the RIPLE-DE, such as the domain features description, domain feature model and the architecture specification. The next sections presents the artifacts in details.

A.2 Domain Feature Model

Figure A.1 shows the domain feature model using the (Kang et al., 1990) notation, which was adapted and simplified the study purpose’s.

Due to the scope of the study, three variabilities (Authentication, Fine and Details) were defined, since they are usually used in any library application. Moreover, they can be encapsulated as possible services, since a system may track authentication and fine operations from distinct applications. For a better understanding of the feature model (Figure A.1), the commonalities are related to Renting, Access Control and Item features and its mandatory sub features, such as Authentication, Fine and Details. Most of the libraries have rental management for every item available for a particular user, and
A.3. USE CASES

Figure A.1 R4ll domain feature model simplified.

Figure A.2 presents all the features provided by the R4ll product line represented as a use case diagram. The features are implemented by a user specific system, which can take on specific roles in the R4ll.

A.3 Use Cases

Figure A.2 presents all the features provided by the R4ll product line represented as a use case diagram. The features are implemented by a user specific system, which can take on specific roles in the R4ll.

A.4 R4ll Specification Architecture

This step is responsible for defining the reference architecture of the domain, considering components and their classes. However, it was refined to support SOA style design, such as the decomposition and reorganization of some modules into services candidates. In order to obtain a practical and effective way to achieve this, it was defined on a set of principles that was followed in the execution of the case study, such as separation...
of concerns and information hiding, parameterization, consistency, commonality and variability, and traceability. In order to detail the reference architecture used in the R4ll case study, we considered the following elements defined in the RIPLE-DE: architectural styles, modules catalog, and services and components catalog.

**Architectural styles.** The architectural style used was the MVC (Model/View/Controller) style, as can be seen in Figure A.3. MVC aims to decouple the presentation modules of the model. Moreover, it is one of the most widely used standards in Web systems and enables the feature design in a modular fashion.
A.4. R4LL SPECIFICATION ARCHITECTURE

The presentation (view) layer consists of the visual interface of the system. In the case of a Web system, this module would be composed of HTML pages and JSP. The controller layer connects the presentation module with the model layer, thus allowing the decoupling between them. The model layer represents the main part of the system, where rules and business services are implemented. This module will be represented by all entities of the system, as well as the classes responsible for connection with the database.

**Modules catalog.** The service-oriented product line architecture consists of the following modules: security, rental, access, persistence and presentation. Figure A.4 depicts the modules as well as its services and components.

![Figure A.4 R4ll modules.](image)

**Services and components catalog.** As can be seen in Figure A.4, the R4ll architecture is composed of three service candidates (authentication, fine, and loan), and five components (authorization, user, permission, item, and dataaccess). Moreover, the services and components defined earlier will be detailed with its internal structure, i.e., classes and its relationships. Figure A.5 and Figure A.6 presents all the classes which are part of the services and components.
Figure A.5  R4ll authentication service, item and authorization components.

Figure A.6  R4ll fine service, loan component, permission component, and user component.
Travel Reservation Product Line

B.1 Project Context

The travel reservation service-oriented product line should offer its customers the benefit of planning and reserving travel arrangements on the Internet. This product line should fit the requirements of similar travel agencies. Thus, the Travel Agency term will be used to describe this product line throughout this document. The travel agency product line should achieve four key goals through the development and deployment of its services as described next (Snell, 2002):

• The product line should allow customers to submit travel itineraries and payment information to the product line services using a Web interface;
• The travel agency services should automatically obtain and reserve the appropriate services for the airline, hotel or vehicle according to the customer itineraries;
• Its services should perform compensation operations for canceling itinerary failures; and
• It should automatically return confirmation or failure of all reservations back to the customer once the processing of the itinerary is complete.

In this sense, different products in the line will be customized to fit the requirements of specific travel agencies, e.g., from small travel agencies that deal with airline ticket reservations to bigger travel agencies that provide services to reserve airline tickets, accommodation and vehicle. These functionalities were selected because they are essential for the travel agency domain. The motivation to use a service-oriented architecture in this product line is the ability to allow travel agencies to expose their business processes
through services, allowing integration with services exposed by their business partners. This fact will give these travel agencies a significant strategic advantage and help them to streamline their business (Snell, 2002).

In addition, the products in the line should have a way to integrate its itinerary processing workflow with those of the airlines, hotels and car rental companies. In other words, a service responsible for the airline reservations, for example, should access a service registry in order to identify the airline business partners available for each specific product. However, each business partner must externalize its services, and these services must be registered in the service registry of a product. The basic process is illustrated in Figure B.1.

![Figure B.1 Travel reservation business process.](image)

**B.2 Domain Feature Model**

Figure B.2 depicts the variability presented in the Travel Agency using a feature model. As it can be seen, the **airline**, **hotel** and **vehicle** reservations are optional non-exclusive features (OR), i.e., at least one of them should be selected for a specific product. The **payment** method can be made by **credit** or **debit** card, and **bank billet**. All payment types can be presented in a product depending on its requirements. Moreover, the notification messages can be realized using **SMS** or **email**. Only one of them (SMS or email) will be present in a product.
B.3 Domain Use Cases

After the definition of the domain feature model, the next step was the identification and description of the use cases. Thus, the next tables present the use cases of travel agency product line.

Table B.1 Fulfill Itinerary.

<table>
<thead>
<tr>
<th>Actor</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Flow</td>
<td></td>
</tr>
<tr>
<td>1. The user selects the reservations desired among the reservations available for the product;</td>
<td></td>
</tr>
<tr>
<td>2. User selects the date and time for its reservations;</td>
<td></td>
</tr>
<tr>
<td>3. The user selects one of the payment types available for the product, and provides the required information, e.g., credit card number.</td>
<td></td>
</tr>
<tr>
<td>Post-Conditions</td>
<td></td>
</tr>
<tr>
<td>The travel itinerary must be fulfilled.</td>
<td></td>
</tr>
</tbody>
</table>

Table B.2 Check Itinerary.

<table>
<thead>
<tr>
<th>Actor</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Flow</td>
<td></td>
</tr>
<tr>
<td>1. The system checks possible errors in the itinerary;</td>
<td></td>
</tr>
<tr>
<td>2. The system notifies the acceptance of the itinerary.</td>
<td></td>
</tr>
<tr>
<td>Alternative Flow</td>
<td></td>
</tr>
<tr>
<td>If it found any error, the system presents the errors to the user in order to correct them.</td>
<td></td>
</tr>
<tr>
<td>Pre-Condition</td>
<td></td>
</tr>
<tr>
<td>The itinerary must be fulfilled.</td>
<td></td>
</tr>
<tr>
<td>Post-Conditions</td>
<td></td>
</tr>
<tr>
<td>The itinerary must be correctly validated.</td>
<td></td>
</tr>
</tbody>
</table>

B.4 Domain Design Architecture Reference

In this section, the services, components, classes, its relationships, commonalities and variabilities are detailed through the use of UML diagrams. After the identification and
In order to implement these variation points, variability implementation mechanisms must be selected. Figure B.3 shows the mapping of the services and components identified through the previous steps (analysis and design). As can be seen in Figure B.3, there are three possible variants such as HotelAgency, AirlineAgency, and RentalCarAgency, which are addressed by the ReservationService variation point.

In Figure B.3, some design variation techniques were used to map the commonality and variability through the services and components identified. The design techniques used were Parameters, and Encapsulate variability within services or components. These techniques are detailed in the implementation side in Chapter 5. Figure B.4 and Figure
Table B.6 Compensation.

<table>
<thead>
<tr>
<th>Actor</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main Flow</strong></td>
<td>3. If any of the reservation tasks fail, the itinerary is cancelled by performing the compensate activity. In this case, all the reservations already performed for a specific itinerary are cancelled; 4. The customer is notified about the cause of the problem.</td>
</tr>
<tr>
<td><strong>Post-Condition</strong></td>
<td>All reservations of the itinerary must be cancelled correctly.</td>
</tr>
</tbody>
</table>

**Figure B.3** Services, components, and its relationships through a service-oriented product line design.

**B.5** depicts the services and components internal structure with the support of the techniques aforementioned. For example, the Encapsulate variability within services or components was applied to define the notification type (SMS or Email) concerning the NotificationService.

**B.5** depicts the services and components internal structure with the support of the techniques aforementioned. For example, the Encapsulate variability within services or components was applied to define the notification type (SMS or Email) concerning the NotificationService.

Figure B.6 depicts the entities diagram defined showing entities mapped as variants (SMS, Email, Vehicle, Flight, Hotel, Billet, and CreditCard) and variation points (Reserve, and Payment). The entities diagram is important to identify fine-grained variabilities presented in classes (see Chapter 5). In order to deal with fine-grained variabilities, some variability implementatin mechanisms may be applied, for example, Inheritance and Design Patterns. Most of the variants identified in the entities diagram were encapsulated using simple Inheritance combined with the Strategy Design Pattern (e.g., Paymet and Reserve variation points).
Figure B.4 Services and components specification.

Figure B.5 Services and components specification.
Figure B.6 Travel agency product line entities modeling.