



Pós-Graduação em Ciência da Computação

# “SOPLE-DE: An Approach to Design Service-Oriented Product Line Architectures”

by

**Flávio Mota Medeiros**

M.Sc. Dissertation



Universidade Federal de Pernambuco  
posgraduacao@cin.ufpe.br  
[www.cin.ufpe.br/~posgraduacao](http://www.cin.ufpe.br/~posgraduacao)

Recife, March/2010





Universidade Federal de Pernambuco  
Centro de Informática  
Pós-graduação em Ciência da Computação

Flávio Mota Medeiros

## **“SOPLE-DE: An Approach to Design Service-Oriented Product Line Architectures”**

*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, March/2010

Medeiros, Flávio Mota  
SOPLE-DE: an approach to design service-oriented  
product line architectures / Flávio Mota Medeiros. - Recife: O  
Autor, 2010.  
xii, 133 p. : il., fig., tab.

Dissertação (mestrado) – Universidade Federal de  
Pernambuco. CIN. Ciência da Computação, 2010.

Inclui bibliografia, glossário e apêndice.

1. Engenharia de software. 2. Reuso de software. 3.  
Arquitetura de software. I. Título.

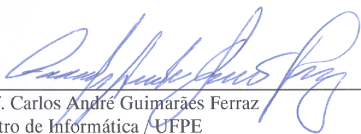
005.1

CDD (22. ed.)

MEI2010 – 056

---

Dissertação de Mestrado apresentada por **Flávio Mota Medeiros** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título “**An Approach to Design Service-Oriented Product Line Architectures**”, orientada pelo **Prof. Silvio Romero de Lemos Meira** e aprovada pela Banca Examinadora formada pelos professores:



Prof. Carlos André Guimarães Ferraz  
Centro de Informática / UFPE

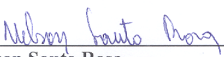


Prof. Uíra Kulesza  
Departamento de Informática e Matemática Aplicada / UFRN



Prof. Silvio Romero de Lemos Meira  
Centro de Informática / UFPE

Visto e permitida a impressão.  
Recife, 26 de março de 2010.



**Prof. Nelson Souto Rosa**  
Coordenador da Pós-Graduação em Ciência da Computação do  
Centro de Informática da Universidade Federal de Pernambuco.

*I dedicate this dissertation to my wonderful family, friends,  
and my lovely girlfriend.*

# Acknowledgements

Initially, I would like to thank all the professors from the Federal University of Alagoas (UFAL) and Federal University of Pernambuco (UFPE) who gave me the necessary educational support to get here.

Next, I would like to thank CNPq for the financial support, which helped me to live in Recife during my master degree. Without this support, I could not spend my time researching and trying to do my best to complete this dissertation on time.

The results of this dissertation could not be achieved without the support of the Reuse in Software Engineering (RiSE) group. My gratitude to the RiSE members for their patience during the experiment, seminars, presentations, and discussions about my work.

Some key researchers in the software reuse area offered valuable feedback for this work. My gratitude to John McGregor and Sholom Cohen for their suggestions during our discussions that improved the quality of my work.

Finally, I would like to thank my great family, friends, and my wonderful girlfriend. Especially, my parents that always stood by me with everything I needed during my life, and my girlfriend for her patience and comprehension.

O reuso de software é um fator extremamente importante para empresas interessadas em aumentar sua produtividade, diminuir os custos e o tempo durante o desenvolvimento de sistemas e melhorar a qualidade de seus produtos e serviços. Nesse contexto, Linhas de Produto de Software (LPS) e Arquitetura Orientada a Serviços (SOA) são duas estratégias que estão atualmente recebendo uma grande atenção, tanto na área acadêmica quanto na indústria de software. Os conceitos de linhas de produto e arquitetura orientada a serviços compartilham alguns objetivos e características que podem ser usados em conjunto para aumentar as taxas de reuso de software. 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. Com essa visão, esse trabalho apresenta uma abordagem para o projeto de arquiteturas para linhas de produto orientada a serviços, constituída de um conjunto de atividades e sub atividades com entradas e saídas especificadas, sendo cada uma delas realizada por um conjunto predefinido de papéis com responsabilidades definidas. Essa abordagem visa ajudar arquitetos de software a projetar arquitetura orientada a serviços para domínios específicos. Para garantir a qualidade da abordagem desenvolvida, uma pesquisa extensiva foi realizada para analisar o atual estado da arte de processos para o desenvolvimento orientado a serviços. Foram então considerados os pontos fracos e fortes dos processos estudados com o intuito de identificar e preencher as lacunas neles existentes. Por fim, essa abordagem foi validada e refinada por meio de um estudo acadêmico experimental preliminar.

**Palavras-chave:** Linhas de Produto de Software (LPS), Arquitetura Orientada a Serviços (SOA), Arquitetura de Software e Processos de Desenvolvimento de Software.



# Abstract

Software reuse is a key factor for enterprises interested in productivity gains, decreased development costs, improved time-to-market, and software quality. In this context, Software Product Line (SPL) and Service-Oriented Architecture (SOA) are two reuse strategies that are getting a lot of attention in research and practice lately. SPL and SOA share some goals and characteristics, which motivate the use of both together with the purpose of increasing reuse rates. However, this combination needs a well-defined development process. Without this process, the development team may develop in an ad-hoc manner with success relying on the effort of a few individual members, what may increase the risks of failure, development costs and time-to-market. In this sense, this dissertation presents an approach to design service-oriented product line architectures with a well-defined sequence of activities and sub-activities with clearly defined inputs and outputs, and performed by a predefined set of roles with specific responsibilities. This approach aims to aid software architects during the design of service-oriented product line architectures. In order to ensure the quality of the proposed approach, an extensive study was performed to analyze the current state-of-the-art of existing service-oriented development processes. In addition, the approach was defined based on the drawbacks and strengths of the processes available with the purpose of filling the gaps in the area. Moreover, the approach was also applied in an academic and preliminary experimental study performed with the intention of evaluating and refining the proposed solution.

**Keywords:** Software Product Line (SPL), Service-Oriented Architecture (SOA), Software Architecture and Software Development Process.

# Contents

<b>Acronyms</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Motivation . . . . .	3
1.2 Problem Statement . . . . .	5
1.3 Overview of the Proposed Solution . . . . .	5
1.3.1 Context . . . . .	5
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
<b>2 Software Product Line (SPL): An Overview</b>	<b>11</b>
2.1 Introduction . . . . .	11
2.2 Motivations of Software Product Line Engineering . . . . .	12
2.3 Software Product Line Engineering . . . . .	13
2.4 Software Product Line Adoption Models . . . . .	15
2.5 Product Line Architecture (PLA) . . . . .	16
2.5.1 The Benefits of the Software Architecture . . . . .	17
2.5.2 Factors that Influence the Architecture . . . . .	18
2.6 Chapter Summary . . . . .	19
<b>3 Service-Oriented Architecture (SOA): An Overview</b>	<b>20</b>
3.1 Introduction . . . . .	20
3.2 SOA Characteristics . . . . .	21
3.2.1 Distributed Systems . . . . .	21
3.2.2 Different Owners . . . . .	22
3.2.3 Heterogeneity . . . . .	22
3.3 SOA Origins and Influences . . . . .	22
3.3.1 Object-Oriented Programming . . . . .	23
3.3.2 Web Services . . . . .	23
3.3.3 Business Process Modeling (BPM) . . . . .	24
3.3.4 Enterprise Application Integration (EAI) . . . . .	24
3.3.5 Aspect-Oriented Programming (AOP) . . . . .	24

---

3.4	SOA Motivations . . . . .	25
3.5	Service-Oriented Principles . . . . .	26
3.6	SOA Roles . . . . .	27
3.7	Enterprise Service Bus (ESB) . . . . .	28
3.8	Chapter Summary . . . . .	29
<b>4</b>	<b>A Systematic Review on SOA Design Methodologies</b>	<b>30</b>
4.1	Introduction . . . . .	30
4.2	The Systematic Review Process . . . . .	31
4.3	Planning the Review . . . . .	31
4.4	Conducting the review . . . . .	34
4.5	Data Synthesis . . . . .	37
4.6	Results of the Systematic Review . . . . .	40
4.6.1	Activities, Artifacts and Roles . . . . .	40
4.6.2	Quality Attributes . . . . .	46
4.6.3	Delivery Strategy . . . . .	47
4.6.4	Adaptation of Existing Processes . . . . .	48
4.7	Chapter Summary . . . . .	49
<b>5</b>	<b>An Approach to Design Service-Oriented Product Line Architectures</b>	<b>51</b>
5.1	Introduction . . . . .	51
5.2	Principles . . . . .	52
5.2.1	Component-Based Development (CBD) . . . . .	52
5.2.2	Feature-Oriented Development (FOD) . . . . .	52
5.2.3	Process-Oriented and Service-Oriented Development . . . . .	53
5.2.4	Separation of Concerns and Information Hiding . . . . .	53
5.2.5	Commonality and Variability . . . . .	53
5.2.6	Quality Attributes . . . . .	54
5.2.7	Cohesion and Granularity . . . . .	54
5.2.8	Top-Down and Proactive Development . . . . .	55
5.2.9	Systematic Sequence of Activities . . . . .	55
5.3	SOPLE-DE Overview . . . . .	55
5.3.1	Inputs and Outputs . . . . .	56
5.3.2	Roles . . . . .	57
5.3.3	Architectural Style . . . . .	58
5.3.4	Development Cycles . . . . .	60

---

---

5.3.5	Activities . . . . .	60
5.4	The SOPLE-DE Approach . . . . .	61
5.4.1	Architectural Elements Identification Activity . . . . .	63
5.4.2	Variability Analysis Activity . . . . .	74
5.4.3	Architecture Specification Activity . . . . .	79
5.4.4	Architectural Elements Specification Activity . . . . .	83
5.4.5	Design Decisions Documentation Activity . . . . .	85
5.5	Chapter Summary . . . . .	87
<b>6</b>	<b>A Preliminary Experiment</b>	<b>88</b>
6.1	Introduction . . . . .	88
6.2	Background Information . . . . .	89
6.3	The Experimental Study . . . . .	90
6.3.1	Definition . . . . .	91
6.3.2	Planning . . . . .	94
6.3.3	Operation . . . . .	101
6.3.4	Analysis and Interpretation . . . . .	102
6.4	Conclusions and Lessons Learned . . . . .	107
6.5	Chapter Summary . . . . .	109
<b>7</b>	<b>Conclusions</b>	<b>110</b>
7.1	Related Work . . . . .	111
7.2	Future Work . . . . .	112
7.3	Concluding Remarks . . . . .	114
	<b>Bibliography</b>	<b>115</b>
	<b>Appendices</b>	<b>124</b>
<b>A</b>	<b>Architecture Document Template</b>	<b>126</b>
<b>B</b>	<b>Instruments of the Experimental Study</b>	<b>131</b>

---

# List of Figures

1.1	RiSE Labs Influencing Areas . . . . .	5
1.2	RiSE Labs Projects . . . . .	7
2.1	The Essential Activities of Software Product Line Engineering . . . . .	14
2.2	Software Product Line Engineering Framework . . . . .	15
3.1	SOA Origins and Influences . . . . .	22
3.2	Distributed Object-Oriented Systems and SOA . . . . .	23
3.3	SOA Roles . . . . .	28
5.1	Architectural Style . . . . .	59
5.2	SOPLE-DE Activities . . . . .	61
5.3	The Project used as Example . . . . .	62
5.4	Architectural Elements Identification Activity . . . . .	63
5.5	Service Granularity . . . . .	66
5.6	Identification of Architectural Elements from Features . . . . .	68
5.7	Travel Reservation Feature Model . . . . .	69
5.8	Conceptual Service Model . . . . .	70
5.9	SOA Design Patterns . . . . .	71
5.10	Variability Analysis Activity . . . . .	75
5.11	Analyzing Cohesion . . . . .	76
5.12	Component Variability . . . . .	77
5.13	Variability Granularity . . . . .	78
5.14	Architecture Specification Activity . . . . .	80
5.15	Layer and Integration Views . . . . .	81
5.16	Interaction and Component Views . . . . .	82
5.17	Architectural Elements Specification Activity . . . . .	83
5.18	Service Interface . . . . .	84
5.19	Design Decisions Documentation Activity . . . . .	86
6.1	Experiment Variables . . . . .	89
6.2	Activities of the Experiment Process . . . . .	90
6.3	Service Coupling . . . . .	103
6.4	Service Coupling Mean . . . . .	104
6.5	Service Instability . . . . .	104

---

6.6	Service Instability Mean . . . . .	105
6.7	Cohesion of the Service Operations . . . . .	105

# List of Tables

4.1	Search Strings . . . . .	34
4.2	Electronic Databases . . . . .	34
4.3	Conferences and Journals . . . . .	35
4.4	Service Identification Techniques . . . . .	42
4.5	Service Specification Artifacts . . . . .	43
4.6	Roles Involved in the Process . . . . .	46
4.7	Service Delivery Strategies . . . . .	48
4.8	Summary of the Review . . . . .	49
5.1	Checklist of the Architectural Elements Identification Activity . . . . .	74
5.2	Checklist of the Variability Analysis Activity . . . . .	80
5.3	Checklist of the Architecture Specification Activity . . . . .	82
5.4	Checklist of the Architectural Elements Specification Activity . . . . .	86
5.5	Checklist of the Design Decisions Documentation Activity . . . . .	87
6.1	One Factor with Two Treatments Design . . . . .	98
6.2	The Profile of the Subjects . . . . .	102
A.1	Component Specification Template . . . . .	128
A.2	Service Specification Template . . . . .	128
A.3	Service Orchestration Specification Template . . . . .	129
A.4	Flow Specification Template . . . . .	130
B.1	Questionnaire for Subjects Background (Part 1) . . . . .	131
B.2	Questionnaire for Subjects Background (Part 2) . . . . .	132
B.3	Questionnaire for Subjects Feedback . . . . .	133

# Acronyms

**AOP** Aspect-Oriented Programming

**BPM** Business Process Modeling

**BPMN** Business Process Modeling Notation

**CBD** Component-Based Development

**EAI** Enterprise Application Integration

**ESB** Enterprise Service Bus

**FOD** Feature-Oriented Development

**GQM** Goal Question Metric

**MDD** Model Driven Development

**PLA** Product Line Architecture

**POD** Process-Oriented Development

**RUP** Rational Unified Process

**SLA** Service Level Agreement

**SOA** Service-Oriented Architecture

**SOAP** Simple Object Access Protocol

**SOD** Service-Oriented Development

**SO-PLA** Service-Oriented Product Line Architecture

**SPL** Software Product Line

**UDDI** Universal Description, Discovery, and Integration

**URI** Uniform Resource Identifier

**WSDL** Web Service Description Language



# 1

## Introduction

Software reuse is a key factor for enterprises interested in productivity gains, reduced development costs, improved time-to-market, and increased software quality ([Krueger, 1992](#)). In the context of software reuse, Software Product Line (SPL) and Service-Oriented Architecture (SOA) are two strategies that are getting a lot of attention in research and practice lately.

These strategies share common goals, i.e., they both encourage organizations to develop flexible and cost-effective software systems, and support the reuse of existing software and capabilities during the development of new systems ([Istoan, 2009](#)). However, at the same time, the main characteristics of SPL and SOA are quite different, and they also focus on distinct goals.

SOA enables assembly, orchestration and maintenance of enterprise solutions constructed based on a set of reusable, self-contained and business-aligned services with the main purpose of reacting to changes in the business requirements quickly ([Josuttis, 2007](#)). Conversely, SPL systematically captures and exploits the commonality among a set of related products, while manages variability to customize these products to specific customers and market segment needs ([Cohen and Krut, 2007](#)).

In addition, SOA usually proposes the development of large, heterogeneous and distributed systems that may be under control of different ownership domains ([MacKenzie et al., 2006](#)). In contrast, SPL focuses on the development of a set of similar products using a common platform, and often controlled by a single organization ([Pohl et al., 2005](#)). However, the differences between these reuse strategies, in some cases, may complement each other ([Helferich et al., 2007](#)).

This dissertation explores the combination of the concepts and characteristics of SPL and SOA in a single software engineering methodology. In particular, an approach to design service-oriented product line architectures is depicted in this work. In the proposed

solution, SPL concepts, such as managed variability and systematic planned reuse, are used to support the non-systematic reuse of SOA environments. On the other hand, SOA concepts, e.g., contract-based communication, dynamic service discoverability and the ability to compose services dynamically, are used to aid the development of flexible and dynamic software product lines (Lee *et al.*, 2008).

In this dissertation, a service-oriented product line is considered as a set of similar service-oriented systems that supports the business processes of a specific domain and can be developed from a common platform or core assets, such as requirements, diagrams, components and services (Clements and Northrop, 2001). In this way, service-oriented systems are developed as a software product line and can be customized to specific customers or market segment needs (Boffoli *et al.*, 2008).

The remainder of this chapter describes the focus and structure of this dissertation. Section 1.1 starts presenting its motivations, and a clear definition of the problem scope is depicted in Section 1.2. An overview of the proposed solution is presented in Section 1.3. Some related aspects that are not directly addressed by this work are shown in Section 1.4. In the Section 1.5, the main contributions of this work are discussed, and finally, Section 1.6 describes how this dissertation is organized.

## 1.1 Motivation

In the software development industry, organizations are always trying to reduce the costs, effort and time-to-market of software products (Linden *et al.*, 2007). Moreover, the development of flexible systems that can react to market changes quickly (Carter, 2007) and can be customized to specific customers (Pohl *et al.*, 2005) is essential in the current business environment. However, the complexity and size of the systems are increasing, which motivate the reuse of existing assets during the development of new systems, especially when assets developed in different languages can be integrated and reused in several contexts (Arsanjani *et al.*, 2008).

In this context, SOA is an approach that aids to solve integration and interoperability problems (Papazoglou and Heuvel, 2006) and align Information Technology (IT) and business goals, giving more flexibility and agility to the enterprises (Josuttis, 2007). On the other hand, SPL explores the commonality among a set of similar products and manages variability supporting the development of products that fit specific customers requirements or market segment needs (Clements and Northrop, 2001). In addition, SPL and SOA focus on reusing existing software and capabilities.

SOA lacks on support for high customization and systematic planned reuse, which is domain focused, based on repeatable processes, and concerned primarily with the reuse of high level artifacts such as specifications, designs and components (Frakes, 1994). In other words, despite of the natural way of achieving customization in service-oriented applications, i.e., changing service order or even the participants of service compositions, services are not designed with variability to be highly customizable and reusable in specific predefined contexts. In addition, service artifacts, e.g., specifications and models, are not produced with variability as well. Hence, a family of service-oriented applications cannot reuse these artifacts in a systematic manner (Helferich *et al.*, 2007).

Thus, SPL engineering, which has the principles of variability, customization and systematic planned reuse in its foundation, can be used to aid SOA to achieve these benefits. In this path, service-oriented applications that support a particular set of business processes can be developed as a SPL (Ye *et al.*, 2007; Boffoli *et al.*, 2008). Moreover, SOA can also sustain the development of flexible and dynamic SPLs due to its characteristics, e.g., loose coupling, contract-based communication, dynamic service discoverability, and the ability to compose services dynamically (Lee *et al.*, 2009).

The motivation to combine SPL and SOA is to achieve desired benefits, such as productivity gains, decreased development costs and effort, improved time-to-market, applications customized to specific customers or market segment needs, competitive advantage, flexibility and dynamic composition of software modules (Cohen and Krut, 2007). However, in order to gain these advantages, a well-defined process considering SPL and SOA concepts is essential. Without this process, the development team will develop in an ad-hoc manner, with success relying on the efforts of a few dedicated individual participants (Booch, 1995).

In this sense, this dissertation proposes an approach to design service-oriented product lines (SOPLE-DE) that combines SPL and SOA concepts with the intention of achieving the desired benefits mentioned before. The combination of SPL and SOA used in this work is explained and characterized in Chapter 5, which presents the SOPLE-DE in detail. Furthermore, Chapter 4 presents the state-of-the-art about existing processes and methodologies for SOA development, which were analyzed to create the proposed solution.

## 1.2 Problem Statement

Encouraged by the motivations depicted in the previous section and the lack of service-oriented product line processes, the goal of this dissertation can be stated as follows:

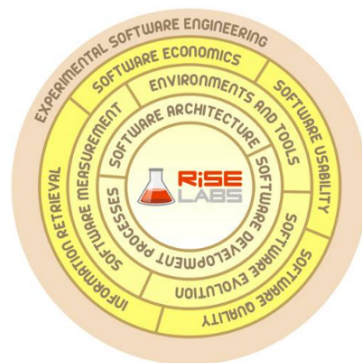
*“This work investigates the problem of developing software product lines using service-oriented architectures, characterizing it empirically to understand the impact of using such an architecture and software product line concepts together. It provides a systematic design approach with clearly defined activities, sub-activities, steps, inputs, outputs and roles to aid software architects to design service-oriented product line architectures. Moreover, the proposed approach is based on a set of software product line, service-oriented architecture, component-based development, and design principles”.*

## 1.3 Overview of the Proposed Solution

This section presents the context of this work and an overview of the proposed solution.

### 1.3.1 Context

This dissertation is part of the Reuse in Software Engineering (RiSE)<sup>1</sup> Labs, whose goal is to develop a robust framework for software reuse with the purpose of making the adoption of a reuse program easier (de Almeida *et al.*, 2004). However, the RiSE Labs is influenced by several areas as depicted in Figure 1.1.



**Figure 1.1** RiSE Labs Influencing Areas

---

<sup>1</sup><http://www.rise.com.br/research>

Based on these areas, the RiSE Labs is divided in several sub-projects, as shown in Figure 1.2. As it can be seen, this framework embraces several different projects related to software reuse and software engineering:

- **RiSE Framework:** Involves reuse processes, e.g., (de Almeida *et al.*, 2004; Nascimento, 2008), component certification (Alvaro *et al.*, 2006), and reuse adoption processes, e.g., (Garcia *et al.*, 2008);
- **RiSE Tools:** Research focused on software reuse tools, such as the Legacy Information retrieval Tool (LIFT) (Brito, 2007), the Admire Environment (Mascena, 2006), and the Basic Asset Retrieval Tool (B.A.R.T) (Santos *et al.*, 2006), which was enhanced, for example, with facets (Mendes, 2008) and data mining (Martins *et al.*, 2008);
- **RiPLE:** Development of a methodology for software product line engineering, which is divided in several smaller disciplines, such as requirements (Neiva, 2009), and design (Souza Filho *et al.*, 2009);
- **SOPLE:** Development of a methodology for software product lines based on services, which is also divided into smaller disciplines. This dissertation is part of this project, and it is concerned with the design of service-oriented product line architectures;
- **MATRIX:** Investigates the area of measurement in reuse and its impact on quality and productivity;
- **BTT:** Research focused on methods and tools for detection of duplicate bug reports, such as in Cavalcanti (2009);
- **Exploratory Research:** Investigates new research directions in software engineering and its impact on reuse;
- **CX-Ray:** Focused on understanding the Recife Center for Advanced Studies and Systems (C.E.S.A.R.)<sup>2</sup>, and its processes and practices in software development.

As mentioned previously, this work is part of the Service-Oriented Product Line Engineering (SOPLE) project, whose goal is to provide a well-defined process for the whole development of service-oriented product lines. However, in this dissertation, the design discipline of the SOPLE project is considered.

---

<sup>2</sup><http://www.cesar.org.br/>

---



**Figure 1.2** RiSE Labs Projects

### 1.3.2 Outline of the Proposal

The proposed solution consists of an approach to design service-oriented product line architectures that aids software architects during the identification, design and documentation of architectural elements, i.e., components, services and service orchestrations, and their communication flows. It also provides guidelines for the high-level and low-level design of these architectural elements.

This approach provides a set of sequential activities with clearly defined inputs and outputs, which are performed by a predefined set of roles with specific responsibilities. During the approach, essential software product line, component-based development, design and service-oriented concepts, such as cohesion, loose-coupling, granularity, commonality and variability are considered to provide a better design discipline.

The main goal of the approach is to aid software architects to produce a Service-Oriented Product Line Architecture (SO-PLA), which represents the common and variable architectural elements of a specific domain. The approach is called SOPLE-DE, and it combines feature-oriented ([Kang et al., 2002](#)), process-oriented ([Boffoli et al., 2009](#)), component-based ([Szyperski, 2003](#)) and service-oriented development ([Erl, 2005](#)). SO-PLA is related to service-oriented product line engineering and DE means design.

The SOPLE-DE receives the domain feature model, the business process models and the quality attribute scenarios of the domain as mandatory inputs. The domain use cases are also considered as inputs when available, i.e., the use cases are optional inputs. In the context of the SOPLE as a whole, the SOPLE-RE (requirements) process provides these inputs. However, the SOPLE-DE is flexible, and can be used with other service-oriented product line requirement process that fits the inputs required by it. The detailed description of the approach is presented in [Chapter 5](#).

## 1.4 Out of Scope

Some aspects that are related to this research will be left out of its scope due to the time constraints imposed on a master degree. This work can be seen as an initial climbing towards a full development process for service-oriented product lines. Thus, the following issues are not directly addressed by this work:

- **Other development disciplines:** Our solution only includes an approach to design service-oriented product line architectures, and other development disciplines will not be described in this work. Nevertheless, the other disciplines, e.g., scoping, requirement, implementation and testing, were envisioned since the initial definitions of the proposed approach. Hence, these disciplines can be added in the future with a few adjustments;
- **Re-engineering activities:** The proposed approach does not consider re-engineering aspects, such as the identification of services from existing legacy applications, nor redesign of existing services and components. However, it does not exclude the possibility to integrate existing components and services into the new architecture. Integration is one of the major benefits of SOA due to its interoperability characteristics, which allow legacy systems developed using different platforms and languages to be leveraged in the new service-oriented solution ([Arsanjani et al., 2008](#));
- **Extractive and reactive adoptions:** SPLs can be adopted using different strategies, e.g., the proactive, extractive or reactive adoption models ([Krueger, 2002](#)). This work uses a proactive adoption model, which concentrates on the development of a SPL from scratch. Thus, the adoption of SPL considering existing products (extractive) or the incremental development of products (reactive) are not being considered in this work. Chapter 2 discusses the adoption models in more detail;
- **Exposing services from components:** There are several ways to develop services, e.g., wrapping legacy applications, enterprise components or using integration tools that can expose services from database queries, files and so on ([Hewitt, 2009](#)). However, this work considers services that are exposed using enterprise components such as Enterprise Java Beans (EJB) ([Panda et al., 2007](#)). In this way, other ways to design and implement services are not being covered by this dissertation;

- **Architecture evaluation:** This work is not considering guidelines or methods for architecture evaluation. However, because the risk and impact of a SOA are distributed across applications and services, it is critical to perform an architecture evaluation early in the software life-cycle to avoid failures of quality attributes, e.g., security, performance, availability, and modifiability ([Bianco \*et al.\*, 2007](#)).

## 1.5 Statement of the Contributions

As a result of this dissertation, the following contributions can be highlighted:

- **A combination of SPL and SOA concepts:** It was conducted an extensive study about the development of software product lines using service-oriented architectures. In this study, it was considered different ways to combine SPL and SOA concepts with the purpose of finding out the combination that could get the benefits of both strategies without affecting the main characteristics of each other;
- **An analysis of the state-of-the-art of SOA design methodologies:** This work presents an overview of the works that have presented guidelines and activities for designing service-oriented systems;
- **An approach to design service-oriented product line architectures:** It provides a set of sequential activities, sub-activities and steps that aid software architects during the design of service-oriented product line architectures considering software product line, service-oriented architecture, component-based development, and design principles for a better design discipline;
- **An empirical study to validate the proposed solution:** This dissertation also presents a preliminary experimental study performed with the purpose of validating and refining the proposed approach.

In addition to the contributions mentioned, a paper presenting the findings of this dissertation was published and other two are under development.

- Medeiros, F. M., Almeida, E. S., and de Meira, S. R. L. (2009). **Towards an Approach for Service-Oriented Product Line Architectures**. In the 3rd Workshop on Service-Oriented Architectures and Software Product Lines (SOAPL), San Francisco, USA.



## 1.6 Organization of the Dissertation

The remainder of this dissertation is structured as follows:

- Chapter 2 presents an overview on software product line engineering, its principles, foundations, architecture and adoption models;
- Chapter 3 discusses the service-oriented architecture field, including its definitions, motivations, concepts, characteristics and roles;
- Chapter 4 depicts a systematic review on service-oriented architecture design methodologies representing the current state-of-the-art in the area;
- Chapter 5 presents the proposed approach to design service-oriented product line architectures (SOPLE-DE) with its foundations, roles, phases, activities, steps, inputs, outputs and guidelines;
- Chapter 6 describes the definition, planning, operation, analysis and interpretation of a preliminary experimental study for the proposed approach performed with the intention of evaluating and refining it;
- Chapter 7 presents some concluding remarks about this work, its related work, and directions for future work.

# 2

## Software Product Line (SPL): An Overview

### 2.1 Introduction

Software product line engineering has its principles based on automobile manufactures, which enable mass production cheaper than individual product creation. These manufactures use a common platform to derive products that can be customized to specific customers or market segments needs ([Clements and Northrop, 2001](#)). In the context of software engineering, the combination of mass customization, large-scale production, and the use of a common platform to derive products results in the software product line engineering paradigm ([Pohl \*et al.\*, 2005](#)).

A product line can be defined as a set of similar software intensive systems that share a collection of common features satisfying the needs of specific customers or market segments. This set of systems are developed from a set of core assets, which are documents, specifications, components, and other software artifacts that naturally become highly reusable during the development of each specific system in the product line ([Clements and Northrop, 2001](#)).

The goal of the software product line engineering is to exploit the commonality of a set of similar systems, while managing variability among these systems with the purpose of developing a family of customized products faster and cheaper than creating each individual product separately ([Gomaa, 2004](#)). In this sense, the software product line development paradigm uses a systematic and planned reuse strategy, i.e., first, the common characteristics among products are explored, reusable parts (core assets) are developed with variability, and then, the products are created and customized to specific customers reusing what has been built to be reused ([Pohl \*et al.\*, 2005](#)).

The remainder of this chapter is organized as follows: Section 2.2 presents the motivations to start a software product line approach, and the essential activities during the SPL engineering are described in Section 2.3; Section 2.4 depicts the adoption models that can be employed when starting a software product line; Section 2.5 presents the characteristics and peculiarities of software product line architectures, and Section 2.6 concludes this chapter with its summary.

## 2.2 Motivations of Software Product Line Engineering

There are several reasons to develop a family of related products using the software product line paradigm. According to Pohl *et al.* (2005), the main motivations are:

- **Reduction of development costs and time-to-market:** Since several products are developed from a set of common core assets (platform), the development costs and time-to-market of individual products are reduced significantly. However, it requires an upfront investment and takes a long time to develop the core assets that will be reused during the development of each individual product;
- **Enhancement of quality:** The core assets of the platform are reused in several products. In this way, they are reviewed, reused and tested several times and in different contexts, what makes it easier to find and correct problems;
- **Reduction of maintenance and evolution costs:** When artifacts of the platform are modified, or new artifacts are added into it, these changes propagate for all products derived from the platform. It makes maintenance and evolution simpler and cheaper than treating it separately for each product;
- **Customer satisfaction:** In a software product line, the products are customized to specific customers. Thus, they receive products that fit their individual needs and wishes, with lower prices and higher quality;
- **Improved cost estimation:** When the reusable core assets are developed, the cost estimations for products from the product line are straight forward and do not include many risks.

However, in order to gain these advantages, the management of variability among products is essential for the success of the product line (Pohl *et al.*, 2005). The variability management activity plays a key role in software product lines because different products

for specific customers or market segment needs are developed in terms of commonality and choices of variability. The software product line engineering paradigm requires specific development processes and activities for dealing with its peculiarities as described in the next section.

## 2.3 Software Product Line Engineering

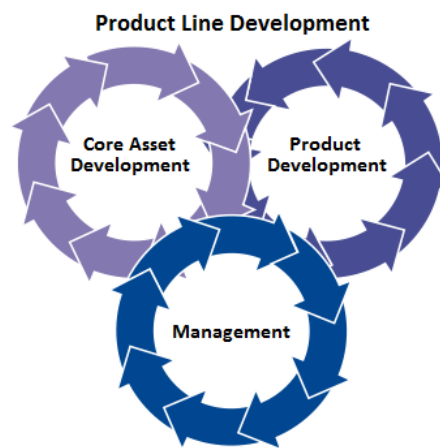
Basically, the software product line engineering is performed using two main phases. According to [Pohl \*et al.\* \(2005\)](#) and [Linden \*et al.\* \(2007\)](#), the product line engineering is divided into domain engineering and application engineering:

1. **Domain engineering:** It is responsible for establishing a reusable and customizable platform, and defining what is common among the products from the product line and what is variable;
2. **Application engineering:** In this phase, the products from the product line are built by reusing the artifacts (core assets) of the platform, and the product line variability is exploited to implement customizations.

In this sense, SPL engineering needs a special type of process that considers the development for reuse, i.e., domain engineering, and the development with reuse during the application engineering. Other researchers use different taxonomies for these phases, e.g., [Clements and Northrop \(2001\)](#) define three essential activities for the software product line engineering (see Figure 2.1):

- **Core asset development:** In this activity, a set of core assets, a product line scope and a production plan are produced. The core assets form the basis of the product line and its production capability. The core asset development receives as input the commonality and variability among the products from the product line, standards and requirements for the products, approaches to realize core assets, and sometimes an inventory of existing assets. This activity is equivalent to domain engineering;
- **The product development activity** receives as input the outputs of the core asset development, and a product-specific requirement. The product required is developed utilizing the core assets developed previously. Creating a product provides a strong feedback effect on the product line scope, production plan, and core assets. This activity is similar to application engineering;

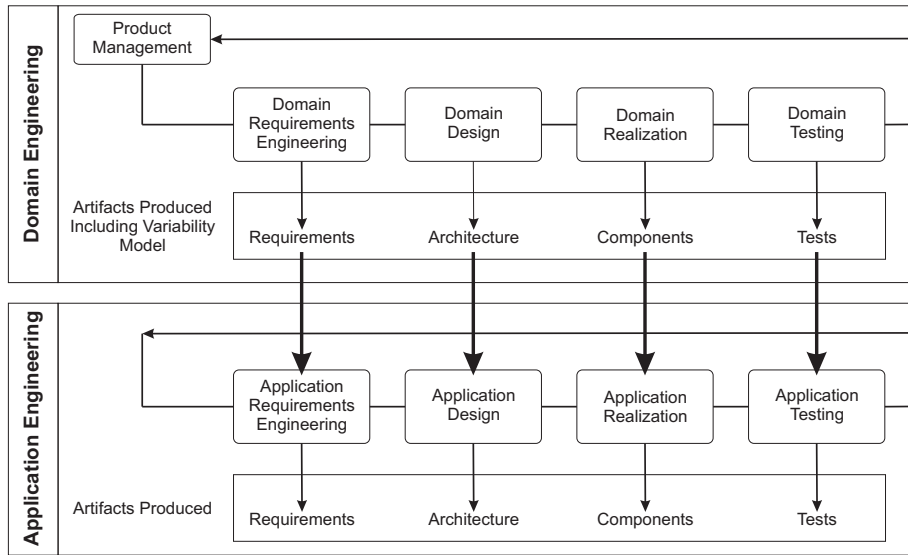
- **Management** is extremely necessary during the software product line engineering due to the fact that the core asset development and the product development activities are highly iterative, and this iteration must be carefully managed (Clements and Northrop, 2001). Management at the technical and organizational levels must be strongly committed for the success of the product line. Pohl *et al.* (2005) also emphasizes the importance of management, however, it is not considered as a separated activity or phase, and it is highlighted during the whole product line engineering.



**Figure 2.1** The Essential Activities of Software Product Line Engineering

The core asset and product development activities, or the domain and application engineering are composed by the traditional software development disciplines, e.g., requirements, design and testing. The difference is that during domain engineering or core asset development, the requirements, design and testing artifacts, for example, are developed with variability considering several products, while during application engineering or product development, these artifacts are customized for the specific product being constructed. Figure 2.2 presents a product line engineering framework representing the domain and application engineering phases and their development disciplines (Pohl *et al.*, 2005).

However, a software product line can be developed in different ways such as the traditional software development, e.g., using the waterfall approach or the spiral model (Boehm, 1988). The next section presents some of the adoption models that can be used to start a product line development effort.



**Figure 2.2** Software Product Line Engineering Framework

## 2.4 Software Product Line Adoption Models

Several reasons motivate organizations to introduce a software product line engineering approach. The adoption is usually strongly based on economic considerations, since software product lines support large-scale reuse during the software development, and the reuse rates can be around 90% of the overall software, what dramatically reduces the development costs and time-to-market of products (Linden *et al.*, 2007).

A software product line initiative requires a conscious and explicit effort from the organization interested in adopting a product line strategy. An organization can adopt product line engineering using some adoption models depending on its objectives, budget, time and requirements as described next (Krueger, 2002):

1. **Proactive:** This adoption model is like the waterfall approach to conventional software development. In this case, all product variations on the foreseeable horizon up front are analyzed, designed, and implemented. This model fits organizations that can predict their product line requirements well into the future, and have the time and resources for a long development cycle;
2. **Reactive:** This model is like the spiral or extreme programming approach to conventional software development. In this approach, product variations are analyzed, designed and implemented in an incremental way, i.e., incremented in each spiral or interaction. This adoption model fits organizations that cannot predict their product

line requirements well, and cannot stop their production and extend their deadlines during the product line adoption;

3. **Extractive:** This adoption model reuses existing products as the initial baseline for the product line. This approach fits organizations that intend to make the transition from conventional software development to product line development quickly.

Each adoption model has its associated risks and benefits. In general, the proactive approach involves more risks, since the development cycle is longer and requires a higher upfront investment. However, its returns on investment are higher compared with the reactive and extractive adoption models (Clements, 2002). On the other hand, the reactive and extractive adoption models can eliminate the adoption barrier, reduce risks and make the adoption faster, since several organizations cannot slow down or stop production during the transition (Krueger, 2002).

The adoption of software product line engineering needs an upfront investment, brings implications for the development process, and may require modifications on the organizational structure (Linden *et al.*, 2007). In addition, there are some essential prerequisites for the adoption of a product line approach, for instance, it needs an enabling technology to implement its concepts, well-defined processes, people who know their market customers in order to identify the commonality and variability among products, and a stable domain that does not change frequently to pay off the upfront investments (Pohl *et al.*, 2005). In this sense, each organization has to analyze its own timetables, budget and objectives before selecting a specific adoption model.

The next section presents an overview on product line architectures, which inherit the characteristics of common software architectures, but they have special peculiarities as described next.

## 2.5 Product Line Architecture (PLA)

A crucial point during the design of any large software system is the definition of the high-level organization of its computational elements and their interactions, i.e., the architecture definition (Garlan and Shaw, 1994). The architecture is an abstract view of the system, which concentrates on the behavior and interaction of elements in the system, and distills away details of implementation, algorithm, and data representation (Bass *et al.*, 2003).

The software architecture of a program or computer system is the structure or structures of the system, which comprises software elements, the external visible properties of those elements, and the relationships among these elements (Bass *et al.*, 2003).

Software product line architectures share this definition and characteristics with common software architectures. However, product line architectures play a more important role than architectures for conventional or single systems. It provides a high-level structure for all products from the product line, and it must reflect the scope of the line and supports its commonality and variability (Pohl *et al.*, 2005; Linden *et al.*, 2007). The software architecture is a factor of success or failure of a product line, and it also takes an important position in determining quality attributes (Bass *et al.*, 2003).

The product line architecture is developed during the domain engineering phase, specifically at the domain design activity (Pohl *et al.*, 2005). During the application engineering phase, the product line architecture is used as reference, and it is adapted and extended to generate the architecture for each specific product in the software product line (Linden *et al.*, 2007). The next section describes the benefits of a product line or common software architecture.

### 2.5.1 The Benefits of the Software Architecture

There is a diversity of aspects that makes the software architecture important and essential for any system as listed next (Bass *et al.*, 2003):

- The software architecture expresses a general abstraction of the system, and it can be used by different stakeholders as a common language for understanding, negotiation, consensus, and communication;
- It reveals the earliest design decisions, which are the most complicated to get right and the most difficult to change, since subsequent decisions are made based on these initial decisions and the stakeholders do not have a well understanding of the system in the beginning;
- The architecture permits more precise costs and schedules estimation, since stakeholders can understand modules of the system better and estimate than separately;
- It can serve as introduction to the system for new members;
- The architecture dictates the organization structure, especially when the organization divides groups of labor according to the architecture;



- It determines constraints on the implementation;
- The architecture restricts or supports specific quality attributes such as usability, modifiability and performance;
- And in the context of product line architectures, they contain variability and are used as reference to generate a specific architecture for each system in the product line.

However, software architectures are influenced by the context where it is being developed, the experience of the stakeholders involved in the project, and several other aspects as presented in the next section.

### 2.5.2 Factors that Influence the Architecture

As stated by [Bass \*et al.\* \(2003\)](#), there are several factors that influence the software architecture of a system:

- **The goals and concerns of the stakeholders involved:** The architect has to deal with conflicts among wishes of the stakeholders involved in the project, since each stakeholder may have its own opinion about the architecture of the system;
- **Skills, schedules and budgets:** The knowledge and skills of the stakeholders in specific technologies, the deadlines and the budgets of the project will also influence the way that the software architecture will be produced;
- **Background and experience of the architect:** He/she may have worked in other projects previously. Thus, the architect may want to reuse successful approaches, experiences and technologies;
- **The current environment:** Software engineering techniques and industry standard practices that are been extensively used by many organizations will also influence how the software architecture will be designed, and the selection of technologies that will be used.

It is important to note that software product line architectures inherit the characteristics of architectures for conventional systems. In this sense, the general aspects of software architectures discussed in this section are valid for product line architectures, which also have specific characteristics as commented.

## 2.6 Chapter Summary

This chapter depicted a brief overview on software product line concepts. It discussed the motivations to adopt the software product line engineering paradigm, highlighting its economical benefits that are achieved mainly due to the large scale, planned and systematic reuse that can be accomplished with the use of strategies for software product line engineering.

The essential activities during the software product line engineering were presented considering the different taxonomies that are currently being used in the literature. Next, some adoption models that can be used to start developing systems as software product lines were described, where it was discussed the risks, strengths and drawbacks of the adoption models.

Finally, the general characteristics of software architectures, and the specific characteristics of software product line architectures were detailed. The next chapter presents an overview on Service-Oriented Architecture (SOA) concepts.

# 3

## Service-Oriented Architecture (SOA): An Overview

### 3.1 Introduction

Service-Oriented Architecture (SOA) is a paradigm for developing distributed systems that deliver application functionality as a set of services, which are reused by end-user applications and other coarse-grained services (Endrei *et al.*, 2004). The adoption of SOA is efficient to solve integration and interoperability problems (Papazoglou and Heuvel, 2006), and provides a better Information Technology (IT) and business alignment, giving more flexibility to the enterprises (Josuttis, 2007).

In this dissertation, SOA is considered an architectural concept, which is based on a set of service-oriented principles that goes beyond the adoption of web services and its style of communication (Engels *et al.*, 2008). The Web Services<sup>1</sup> technology is only one of the existing environments that provides support for the development of flexible, loosely coupled and interoperable SOA systems. The Common Object Request Broker Architecture (CORBA)<sup>2</sup>, for example, is another platform that can be used to implement SOA, which is not dependent of any specific technology (McGovern *et al.*, 2003).

The key benefits of SOA are flexibility, interoperability, support for vendor diversity, extensibility, reusability and loose coupling (Erl, 2005). SOA generally leads to the development of large distributed systems that may be under control of different ownership domains (MacKenzie *et al.*, 2006). The main factors for the adoption of SOA are the use of standards, creation of a transition plan, understanding of performance and security requirements, planned governance and proximity with your SOA marketplace (Erl, 2006).

---

<sup>1</sup>For more information, go to see <http://www.w3.org/2002/ws/>

<sup>2</sup>For additional information, visit <http://www.corba.org/>

In addition, SOA requires activities, such as service identification, artifacts, e.g., service specifications, and roles, e.g., service designer, that are not presented in the traditional software engineering, e.g., object-oriented development. For instance, traditional methods do not address the three key elements of SOA: services, flows, and components realizing services. In other words, they do not provide techniques and processes required for the identification, specification and realization of services, their flows and composition, as well as the enterprise-scale components needed to realize and ensure the quality of the services ([Arsanjani, 2004](#)). Hence, SOA needs formal processes in order to ensure that services are modeled and designed consistently, incorporating accepted design principles, conventions, and standards ([Erl, 2007](#)).

The remainder of this chapter is organized as follows: Section 3.2 depicts the main characteristics of SOA systems, and Section 3.3 shows the SOA origins and influences; Section 3.4 presents the motivations to adopt SOA, and Section 3.5 describes the service-oriented principles, which SOA is based on; Section 3.6 shows the roles involved in SOA, and the Enterprise Service Bus (ESB), which is the key element of the SOA infrastructure, is discussed in Section 3.7; finally, Section 3.8 summarizes this chapter with its summary.

## 3.2 SOA Characteristics

SOA is not appropriated for all types of systems ([Josuttis, 2007](#)). However, SOA copes well with many difficult-to-handle characteristics of large systems as described next.

### 3.2.1 Distributed Systems

SOA systems are normally large and distributed. When business grows, it becomes more and more complex, and more organizations get involved in it. In this context, SOA solutions support integration of systems from different companies that may be developed in different platforms and programming languages.

In this sense, SOA is well suited for dealing with the complexity of large distributed systems. Thus, it facilitates interactions between service providers and service consumers due to its interoperability capacity, enabling the realization of business functionalities. In other words, SOA is paradigm for organizing and utilizing distributed capabilities ([MacKenzie \*et al.\*, 2006](#)).

### 3.2.2 Different Owners

Another characteristic of SOA systems is that beside large and distributed, different parts of the SOA solution may be under control of different ownership domains (MacKenzie et al., 2006). The presence of systems controlled by different owners is the key for certain properties of SOA, and the major reason why SOA is not only a technical concept (Josuttis, 2007).

SOA includes practices and processes that are based on the fact that parts of the distributed systems are not controlled by single owners (Josuttis, 2007). Thus, different teams, departments, or even different companies may manage different systems. For this reason, people who manage SOA solutions have to deal with different platforms, languages, schedules, priorities, and budgets.

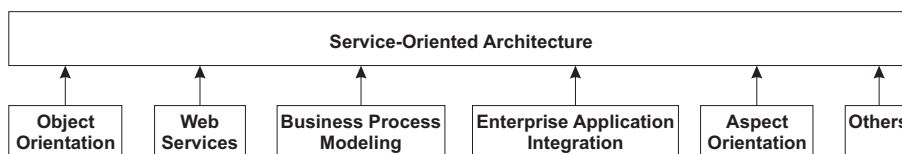
### 3.2.3 Heterogeneity

Large systems differ from small systems due to their lack of harmony. Large systems use different platforms and programming languages as already mentioned. They may be composed by mainframes, databases, Java applications, and so on. In other words, they are heterogeneous.

In the past, some approaches proposed to solve the problem of integrating distributed systems by eliminating heterogeneity. However, it did not work for some businesses (Josuttis, 2007). That is why SOA accepts heterogeneity, and it deals with large distributed systems by acknowledging and supporting this attribute.

## 3.3 SOA Origins and Influences

SOA is a relatively new paradigm that can represents the evolution of IT and has its roots in past paradigms and technologies. Figure 3.1 depicts the main paradigms and technologies that have influenced SOA (Erl, 2007). The ways they influence SOA are explained next.

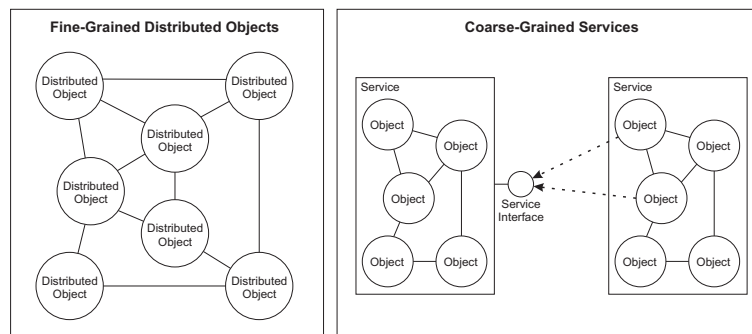


**Figure 3.1** SOA Origins and Influences

### 3.3.1 Object-Oriented Programming

Object Orientation (OO) became popular in the 1990s. This paradigm appeared with its own principles, which helped to ensure consistency among numerous environments (Erl, 2007). In the context of SOA, it is also based on a set of service-oriented principles as already discussed in Section 3.5.

In fact, SOA is more similar with the idea of distributed object-oriented systems than object orientation. However, in distributed object-oriented systems, fine-grained objects are distributed, which increases the number of remote calls among objects and causes performance and maintainability problems. Figure 3.2 compares the SOA and distributed object-oriented paradigms. In the context of SOA, services are more coarse-grained, which decreases the number of remote calls among service consumers and providers. Moreover, the interaction among objects inside each service is local, only the interactions among objects in a service and calls to other services are remote (McGovern *et al.*, 2003).



**Figure 3.2** Distributed Object-Oriented Systems and SOA

### 3.3.2 Web Services

The Web Services Platform is the most appropriated environment to implement SOA currently. The web service platform uses several concepts from the conceptual architecture (SOA) and vice-versa. Both SOA and Web Services are influencing the evolvement of each other (McGovern *et al.*, 2003). For instance, the web service platform has influenced and promoted several service-orientation principles, including service abstraction, service loose coupling, and service composability (Erl, 2007).

### 3.3.3 Business Process Modeling (BPM)

Business Process Modeling (BPM) in the context of software engineering is the activity of representing the processes of an enterprise. It focuses on business processes in order to improve the efficiency of the enterprises. Its main goals are to establish adaptable and extensible business processes that can respond to market changes quickly ([Havey, 2005](#)).

The primary goal of service-orientation is also to establish a highly agile automation environment fully capable of adapting to change. In the context of SOA, this goal is achieved by leaving the business processes in their own layer and using services to implement steps of the processes ([Erl, 2007](#)). Services provide a flexible way to expose discrete business functions, and therefore, an excellent way to develop applications that support business processes ([Brown et al., 2003](#)).

### 3.3.4 Enterprise Application Integration (EAI)

Enterprise Application Integration (EAI) became a primary focal point in the late 90's, and several enterprises started to use point-to-point integration channels to share data across systems, which results in well-known problems such as lack of stability, extensibility, and inadequate interoperability frameworks ([Erl, 2007](#)). In this context, SOA concepts and characteristics appear with their standardized environments that were based on several techniques used in EAI, and several interoperability problems are currently being solved with SOA ([Erl, 2007](#)).

### 3.3.5 Aspect-Oriented Programming (AOP)

Aspect-Oriented Programming (AOP) is a paradigm that encourages the separation of concerns considering functionality that are common in several applications, modules, components, etc. These concerns are classified as “*Cross-Cutting*”, and they naturally become reusable by several modules, components, or applications, e.g., logging and auditing functionality ([Laddad, 2003](#)).

SOA is related to AOP by the fact that it encourages the development of self-contained services that later will be reused to create several service-oriented systems ([Erl, 2007](#)). In the same way as aspects, services can become reusable in several systems that share some business processes.

## 3.4 SOA Motivations

Many organizations are going through the trouble of adopting a service-oriented computing platform, which is not an easy task, but the ideas behind it are very ambitious and attractive for organizations interested in improving the effectiveness of their IT enterprise (Erl, 2007). There are several reasons to adopt SOA, and develop software under the service-oriented development paradigm. The main motivations and goals of SOA are:

- **Reusability:** It has been the major objective of software engineering for years. In the context of SOA, the ability to compose new services reusing the existing ones provides several advantages to an organization, e.g., increased Return on Investments (ROI) and software quality, and decreased development costs and time-to-market (Elfatry and Layzell, 2004). There is an extensive list of modeling guidelines for promoting reuse in SOA (Erl, 2005), however, if reusability concerns are not taken into account deeply during the development, only opportunistic reuse will appear;
- **Flexibility and Business Agility:** The current business scenario requires organizations to continuously modify their systems according to market changes in order to accomplish new requirements, changes in the requirements and several other reasons (Carter, 2007). Hence, the enterprise architecture must be easily reconfigurable. The SOA characteristics enable a better IT and business alignment, integration of new business services and the reuse of existing services in order to respond to market changes quickly (Endrei *et al.*, 2004);
- **Integration and Interoperability:** Systems are no longer required to be developed and deployed in a systematic way. Services can be developed independently and then composed to create applications that can be adapted to market changes quickly (Arsanjani, 2004). In this context, SOA eases the integration among heterogeneous services due to its interoperability capacity. The more interoperable services are, the easier it is for them to exchange information (Erl, 2007);
- **Vendor Diversity:** SOA allows organizations to select among several vendor specific products, since it permits a heterogeneous environment when required and necessary. Thus, the organizations are always free to change, replace and add technologies (Erl, 2007). Vendor diversity is supported by the SOA standards and interoperability issues.



As already mentioned, SOA is normally defined using a set of service-oriented principles ([Erl, 2005](#)). In this sense, the next section depicts the principles that SOA is based on.

## 3.5 Service-Oriented Principles

SOA is based on a set of service-oriented principles that support its theories and characteristics. The set of principles that are directly related to SOA are described next:

- **Coupling:** It refers to the number of dependencies among services. In SOA, services maintain a relationship that minimizes dependencies and only requires that services retain an awareness of each other ([Erl, 2005](#)). Loose coupling is achieved through the use of standards and service contracts among consumers and providers that allow services to interact through well-defined interfaces. Coupling also affects other quality attributes, e.g., modifiability and reusability ([McGovern et al., 2003](#));
- **Service contract:** Services adhere to communication agreements, as defined collectively by one or more service descriptions and related documents. They define data formats, rules and characteristics of the services and their operations. These documents are defined using standards in order to be readable by the software elements of the architecture, e.g., XML, WSDL and XSD policy documents ([Erl, 2007](#));
- **Autonomy and Abstraction:** Services have control over the logic they encapsulate, i.e., they must be autonomous and self-contained. Moreover, beyond what is described in the service contract, services hide internal logic from the outside world. Services are like black boxes, and service consumers only depend on the provided interface ([Erl, 2005](#));
- **Stateless and Idempotent:** Services minimize retaining information specific to a customer's request. They should be as more stateless as possible in order to increase reusability and scalability ([Erl, 2005](#)). Moreover, services should also be idempotent, which is the ability to redo an operation without cause problems, e.g., duplicated data ([Josuttis, 2007](#));
- **Discoverability and Dynamic Binding:** Services are designed to be apparently descriptive so that they can be found and assessed via available discovery mecha-

nisms, e.g., Universal Description, Discovery, and Integration (UDDI)<sup>3</sup> (Erl, 2005). The use of a directory is not obligatory but a service should be discoverable. Service discoverability is usually achieved through a third-party entity that implements a discovery mechanism, e.g., a service registry (McGovern *et al.*, 2003);

- **Coarse-Grained Interfaces:** Services are abstractions that support the separation of concerns and information hiding. However, they slower down performance due to the remote calls (Josuttis, 2007). For this reason, services should provide coarse-grained operations that transfer all the necessary data all together instead of having several fine-grained calls. The requirements of the service consumers should be taken into account when deciding the right granularity for the whole services as well as their operations in order to avoid unnecessary data transfers and performance problems (McGovern *et al.*, 2003).

The next section presents the main roles involved in a SOA solution. Service consumers and providers are the two essential entities. However, other specific roles may also be used as described next.

## 3.6 SOA Roles

In the context of SOA, service consumers may use a third-party entity with the purpose of finding service providers and avoiding tight coupling (McGovern *et al.*, 2003). Figure 3.1 depicts the roles of SOA and their relationships as explained next:

- **Service Consumer:** It is an application, service or other software entity that needs a service. It can either use the Uniform Resource Identifier (URI) for the service description directly or it can find the service description of the provider in the service registry (Arsanjani, 2004);
- **Service Provider:** It is the entity that receives and executes requests from service consumers. It can be a legacy application, a component or other software entity that exposed its interface as a network-addressable and interoperable service, e.g., using the Web Service Description Language (WSDL)<sup>4</sup> (Josuttis, 2007). Providers publish their description in the service registry (McGovern *et al.*, 2003);

---

<sup>3</sup>For more information, visit <http://uddi.xml.org/>

<sup>4</sup> For more information, go to see [www.w3.org/TR/wsdl](http://www.w3.org/TR/wsdl)

- **Service Registry:** It is a third-party entity responsible to maintain the service descriptions (contracts) documents. It can be used to implement some quality attributes, e.g., availability and modifiability, since in some SOA implementations the service consumers have no information about the existing service providers before accessing the registry ([Arsanjani, 2004](#));
- **Service Contract:** It specifies the way the service consumers will interact with the service providers. This contract specifies the message formats, pre-conditions and post-conditions to use service providers, and it may also describe the non-functional requirements (quality attributes) of the providers in order to define Service Level Agreements (SLA) ([McGovern et al., 2003](#); [Erl, 2005](#)).

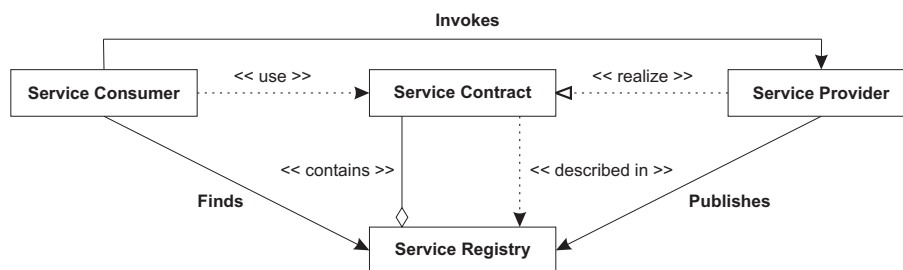


Figure 3.3 SOA Roles

The next section describes an essential part of the SOA infrastructure, i.e., the Enterprise Service Bus (ESB), which is responsible for several tasks as described next.

### 3.7 Enterprise Service Bus (ESB)

The ESB can be considered the backbone of a SOA infrastructure. It is responsible to enable service consumers to call providers. This communication seems simple, but it may involve several tasks, such as providing connectivity, data transformation, routing, dealing with security and reliability, monitoring and logging ([Josuttis, 2007](#)).

However, the main role of an ESB is to provide interoperability. Because it integrates different platforms and programming languages, a fundamental part of this role is data transformation. The usual approach to solve interoperability problems is to introduce a specific format to which all the individual platforms and API map. In the web service platform, for example, this format is usually the Simple Object Access Protocol (SOAP)<sup>5</sup> ([Chappell, 2004](#)).

<sup>5</sup>For more information, go to visit [www.w3.org/TR/soap/](http://www.w3.org/TR/soap/)

Another fundamental task of an ESB is routing. There must be some way of sending a service call from a consumer to a provider, and then sending an answer back from the provider to the consumer (Josuttis, 2007). Other responsibilities of an ESB are usually extensions of the core task of providing interoperability.

However, technically and conceptually, several enterprise service buses can differ widely. For example, a SOA solution might not involve any specific tool or piece of software as its ESB, i.e., just the use of a communication protocol might be enough. In this case, the ESB would delegate a lot of tasks to the service providers and consumers. On the other hand, an ESB might consist of several tools and programs that are used by service designers, implementers, and operators (Josuttis, 2007). Hence, the responsibility of an ESB will depend on the requirements of each specific SOA solution.

## 3.8 Chapter Summary

SOA is a type of software architecture that has special characteristics. This chapter started discussing about the SOA field describing its definitions, characteristics and motivations. The web services technology, which is the most appropriated environment to develop SOA currently, was also mentioned. However, it was emphasized that the web services technology is not the only possible way to develop SOA concepts, which are technology independent.

The origins of SOA and its influencing technologies and paradigms, e.g., enterprise application integration and distributed objects, were detailed. Next, the service-oriented principles that SOA is based on were depicted, e.g., coupling, discoverability and coarse-grained interfaces. Finally, the main roles of SOA, i.e., service consumer, provider, service registry and contract, were described.

The next chapter presents a systematic review on SOA methods for designing service-oriented applications representing the state-of-the-art in the area.

# 4

## A Systematic Review on SOA Design Methodologies

### 4.1 Introduction

Service-Oriented Architecture (SOA) needs a well-defined process to be implanted efficiently and reduce its associated risks ([Erl, 2007](#)). Moreover, without a well-defined process, the development team will develop in an ad-hoc manner, with success relying on the efforts of a few dedicated individual participants ([Booch, 1995](#)). Although existing processes and practices can be reused for service-oriented development, novel techniques are required to address unique SOA requirements ([Ramollari \*et al.\*, 2007](#)).

Hence, enterprises and researchers that desire to adopt a service-oriented solution or start with service-oriented development have to identify the most appropriated method to use or adapt, and the main activities to be considered. Without relying on proven processes and practices, a service-oriented adoption initiative can turn into a high-risk venture and bring unexpected results ([Erl, 2007](#)). Thus, the analysis and comparison of existing methodologies are crucial before launching a service-oriented computing effort.

In this sense, a systematic review is a way to analyze the state-of-the-art of existing methods. It can be used with the purpose of identifying, evaluating and interpreting the available research relevant to a particular research area ([Kitchenham and Charters, 2007](#)). Thus, this chapter presents a deep analysis and summary of the current state-of-the-art of methodologies and processes that provide guidelines and directions for the service-oriented design discipline. The results of this review can be used as background information for architects, companies and researchers that use service-oriented development, are planning to adopt it, or are investigating service-oriented methodologies.

This review aims to identify the essential activities of the service-oriented design discipline, the techniques used for service identification and realization, the notations and diagrams used for architecture documentation, the concerns with reusability and other quality attributes, the strategies used to deliver services and the adaptation of existing processes.

The remainder of this chapter is organized as follows: Section 4.2 presents the process used to perform the systematic review; the planning of the review is depicted in Section 4.3; Section 4.4 shows how the review was conducted; Section 4.5 presents the SOA methods included in the systematic review, and Section 4.6 details the results obtained with the systematic review; Section 4.7 depicts a summary of the review and concludes this chapter.

## 4.2 The Systematic Review Process

The systematic review depicted in this work was performed following the guidelines of (Kitchenham and Charters, 2007), in which a well-defined set of steps for undertaking systematic literature reviews in the context of software engineering is presented. The guidelines are divided in three main phases: planning, conducting and reporting.

The goal of the planning phase is to develop a review protocol that specifies the plan that the systematic review will follow to identify, assess, and gather evidences. The conducting phase is responsible for executing the protocol planned in the previous phase, and the purpose of the reporting phase is to relate the review steps and results to the community.

Each of the phases listed above contains a sequence of stages, as described in the next sections, that may appear to be executed sequentially, but the execution of the overall process involves iteration, feedback, and refinement of the defined process (Kitchenham, 2004). The next sections present the main stages during the planning and conducting phases. The reporting phase is fulfilled with this report.

## 4.3 Planning the Review

The main activity during the planning phase is the definition of the research questions that will be answered with the systematic review. The next sub-sections detail the research questions that guide this systematic review.

## Research Questions

**Q1. Which roles, activities and artifacts are defined in the methods?** The first objective of this review is to analyze which activities, roles and artifacts are used in the service-oriented design methods. In a well-defined process, each phase of the design is composed by activities that may use and produce different types of intermediate artifacts, such as diagrams and checklists. Moreover, each activity or task is normally performed by a predefined set of roles with clear responsibilities ([Kruchten, 2003](#)).

This question is important because service-oriented development requires activities, roles and artifacts, such as the service identification activity and the service description documents, that are not presented in the traditional software engineering, e.g., object-oriented development ([Zimmermann \*et al.\*, 2004](#); [Endrei \*et al.\*, 2004](#)). Thus, the main motivation of this analysis is to identify the essential activities, roles and artifacts that should be used during the service-oriented design.

As this question is related to many aspects of the service-oriented development, in order to improve the clarity of the review, it was divided into sub-questions as described next:

- **SQ1. How the methods identify services?** The identification of services is a challenging task during the service-oriented analysis and design ([Azevedo \*et al.\*, 2009](#)). Potential services can be identified and derived from different sources, e.g., business processes and legacy systems ([Arsanjani \*et al.\*, 2008](#)). Thus, this sub-question aims to identify the techniques used to identify services;
- **SQ2. How the methods specify services?** The idea of this sub-question is to analyze the specification of services, i.e., which models and notations are used during this activity;
- **SQ3. How the service-oriented architecture is documented?** Considering that at the end of the design process a structured document should be produced, the purpose of this analysis is to identify how the architecture documentation is structured and which models and notations are used. The intention here is to analyze structured artifacts produced to represent the architecture ([Bass \*et al.\*, 2003](#));
- **SQ4. Which are the strategies for service realization used in the methods?** Services can be designed and implemented in different ways, e.g., using existing legacy systems and components ([Erradi \*et al.\*, 2006](#)). Thus, this sub-question aims to analyze the realization strategies used by the service-oriented methodologies.

**Q2. How do the methods deal with reusability and other quality attributes?** The purpose of this analysis is to assess how the methodologies handle quality attributes and whether they consider the impact of design decisions at architecture level. This question also aims to identify the reuse strategies used and how the methods take into account reusability among elements, such as services logic, processes, operations and other units of work.

The reason to analyze this aspect is that service orientation is considered an efficient way to increase reuse, flexibility and productivity (Erl, 2005; Carter, 2007). However, rely on service-oriented principles, e.g., coupling, cohesion and granularity, without providing guidelines during the process may be not sufficient to satisfy some quality attributes. Thus, this question aims to identify additional guidelines provided by the methodologies to achieve quality attributes in the SOA development (Papazoglou and Heuvel, 2006; Erradi *et al.*, 2006).

**Q3. How do the methods deal with the delivery of services?** There are several delivery strategies that can be employed to build services (Erl, 2005). The bottom-up strategy, for example, focuses on the fulfillment of immediate business requirements. On the other side, the top-down strategy advocates the completion of an inventory analysis prior to the physical design and delivery of services.

This analysis aims to identify the adopted delivery strategy, that according to Erl (2005), depending on the maturity, goals and budget of the organization, one delivery strategy can be more adequate than other. For instance, while the bottom-up strategy avoids the initial extra cost, effort and time to produce the service inventory, it causes increased governance as bottom-up delivered services tend to have shorter life-spans and require more frequent maintenance, refactoring, and versioning.

The rationale to include this question is to analyze the delivery strategies used by the service-oriented methodologies in order to aid researchers and companies, interested in starting a SOA development, to select the most appropriated method to use based on their goals, maturity, deadlines and budget.

**Q4. Do the methods use or are based on an existing process or methodology?** The idea of this question is to analyze which methodologies adapt existing processes, e.g., the Rational Unified Process (RUP), and which ones build an entirely original one. The reason to add this question is that it can ease the adoption process of a service-oriented methodology, since the organization can adapt its existing processes, instead of bringing something totally new.



**Table 4.1** Search Strings

Term ID	Key Words	String ID	Key Words
T1	Service OR SOA	S1	(T1) AND (T2) AND (T3)
T2	Approach OR method OR process OR framework	S2	(T1) AND (T2)
T3	Analysis OR design OR development OR identification OR discovery OR specification OR realization OR modeling OR identify OR specify OR implement OR engineering OR architect	S3	(T1) AND (T3)

**Table 4.2** Electronic Databases

Electronic Databases
ACM Library - <a href="http://portal.acm.org/">http://portal.acm.org/</a>
IEEE Computer Society - <a href="http://www.computer.org/">http://www.computer.org/</a>
Citeseer library - <a href="http://citeseerx.ist.psu.edu/">http://citeseerx.ist.psu.edu/</a>
IEEE Computer Society - <a href="http://www.computer.org/">http://www.computer.org/</a>
Google Scholar - <a href="http://scholar.google.com">http://scholar.google.com</a>
Science Direct (Elsevier) - <a href="http://www.sciencedirect.com/">http://www.sciencedirect.com/</a>
Scirus - <a href="http://www.scirus.com/">http://www.scirus.com/</a>
Springer Link - <a href="http://www.springerlink.com/">http://www.springerlink.com/</a>
Wiley Inter Science - <a href="http://www.interscience.wiley.com/">http://www.interscience.wiley.com/</a>

## 4.4 Conducting the review

The steps referred to how the review was conducted are detailed in this section. It involves the search strategy, the studies selection and the data analysis, extraction and synthesis.

### Search Strategy

From the research questions, it was extracted some keywords used to search the primary study sources. The initial set of keywords extracted was: service, service-oriented, service orientation, SOA, approach, methodology, process, architecture, analysis, design, identification, specification, realization and development. Based on the refinement of these keywords, the Table 4.1 presents the search terms and the sophisticated search strings that could then be constructed using the Boolean operators (AND) and (OR).

The data sources of the review were conference proceedings, journals, thesis and books. The search process included web search engines and manual searches of conference proceedings and journals. The search for primary studies on web search engines was performed in the digital libraries of the most famous publishers in software engineering as presented in Table 4.2.

**Table 4.3** Conferences and Journals

<b>Conferences and Journals</b>
Asia-Pacific Software Engineering Conference (APSEC)
Congress on Services (SERVICES)
European Conference on Software Architecture (ECSA)
Fundamental Approaches to Software Engineering (FASE)
IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)
IEEE International Symposium on Service-Oriented System Engineering (SOSE)
International Conference on Services Computing (SCC)
International Conference on Software Engineering (ICSE)
International Conference on Web Engineering (ICWE)
International/European Conference on Web Services (ICWS)
Working IEEE/IFIP Conference on Software Architecture (WICSA)
Workshop on Service-Oriented Architectures and Software Product Lines (SOAPL)
Communications of the ACM (CACM)
International Journal of Web Engineering and Technology (IJWET)
IEEE ACM Sigsoft Software Engineering Notes
IEEE Transactions on Services Computing
IEEE Software
IEEE Transactions on Software Engineering (TSE)
ACM Transactions on Software Engineering and Methodology (TOSEM)

In addition, we also searched 12 conference proceedings and 7 journals considering the topic areas about service orientation, service-oriented architecture, software engineering and software architecture. The conferences and journals depicted in Table 4.3 were searched.

## Primary Studies Selection

Once the potentially relevant primary studies have been obtained, they need to be assessed for their actual relevance. To achieve that, the criteria for inclusion and exclusion of the objects in this review were defined. The year of publication was not taken into consideration as inclusion or exclusion criteria.

## Inclusion Criteria

The definition of inclusion and exclusion criteria is important to establish the reasons for each study to be included or excluded of the review. The inclusion criterion used was: Analysis and/or design approaches, methodologies and processes for service-oriented development, also including the ones which deal with other aspects besides design.

##### **Exclusion Criteria**

The following exclusion criteria were used to discard studies of the systematic review on service-oriented design methods:

- Study is not an approach, methodology or process. It treats only concepts or issues related to service-oriented analysis and design, but does not deeply describes a set of sequential activities with clearly defined inputs and outputs;
- Methods with insufficient information about service-oriented analysis and design, or duplicated studies published with different names.

##### **Data Analysis**

During the analysis of the methods included in our review, they were categorized to facilitate the tabulation and further comparative analysis. The studies containing any exclusion criteria were listed, where it was described the exclusion reason. According to the comparisons and analysis of the studies, some aspects were raised as described next:

- The method strengths and weaknesses;
- Challenges and gaps in the area to be addressed;
- Best practices of service-oriented analysis and design.

One problem faced during the analysis was that, sometimes, even methods considering the same scope of functionality treated different aspects of the design discipline. For this reason, the comparative analysis was difficult and the categorization of the studies was essential to obtain the results of the review.

##### **Data Extraction**

In this step, data extraction forms were designed to collect all the information needed to address the review questions and some relevant information to characterize the method, such as a general description, authors and the references used to analyze the methodologies. The data extracted from each approach were:

- The study title, year of publication, authors and source, such as conference or journal;

- Scope, e.g., full life-cycle process, architecture and design approach, service identification method, etc;
- Research questions answers;
- Summary, a brief critical analysis of the method and overview of its characteristics, key points and drawbacks.

The data were extracted by one researcher of the three main researchers performing the review, one Ph.D. and two M.Sc. students, and checked by the research group. Each methodology included in the review was documented in a form template. In this form, all extracted data were documented with the reviewer's name and date of the review.

## 4.5 Data Synthesis

Based on the search results and on the inclusion and exclusion criteria, a set of primary studies were selected as detailed next.

### Service-Oriented Modeling and Architecture (SOMA)

It was developed by IBM. It is a methodology for the full service-oriented life-cycle based on RUP. Its analysis was based on [Endrei \*et al.\* \(2004\)](#), [Arsanjani \(2004\)](#), [Arsanjani and Allam \(2006\)](#) and [Arsanjani \*et al.\* \(2008\)](#).

### Service-Oriented Analysis and Design (Erl's Methodology)

It defines concepts and fundamentals about service-oriented analysis and design. It also defines a method based on a set of principles, such as loose coupling and reusability. Its analysis was based on the books: [Erl \(2005\)](#) and [Erl \(2007\)](#).

### A Methodology for Service Architectures (OASIS)

This method was developed by advancing open standards for the information society<sup>1</sup>. It focuses mainly on the exercise of service discovery. Its analysis was based on [Jones and Morris \(2005\)](#).

---

<sup>1</sup>[www.oasis-open.org/](http://www.oasis-open.org/)

### **Service-Oriented Software Engineering (SOSE) Framework**

This framework proposes a combination of service-oriented and component-based development with the purpose of developing service-oriented applications with increased quality and profitability. Its analysis was based on [Karhunen \*et al.\* \(2005\)](#).

### **Service-Oriented Development Methodology (Papazoglou's Work)**

It is a full life-cycle development methodology for service-oriented systems that range from planning to deploying and monitoring service-oriented applications. Its analysis was based on [Papazoglou and Heuvel \(2006\)](#).

### **Service-Oriented Architecture Framework (SOAF)**

It is a systematic and architecture centric framework to ease definition, design, evaluation and realization of SOA. It incorporates several techniques and guidelines to identify services systematically, to decide service granularity and model services while integrating existing legacy systems. Its analysis was based on [Erradi \*et al.\* \(2006\)](#).

### **Service-Oriented Development with Midas (SOD-M)**

It is an extension of the model-driven development framework for web information systems, called MIDAS<sup>2</sup>. It proposes the development of a service-oriented solution in an architecture centric way. The proposal presents an UML profile that maps the key principles and concepts of service-oriented development making it possible to develop service-oriented solutions model-driven. Its analysis was based on the papers: [de Castro \*et al.\* \(2006\)](#), [Acuña and Marcos \(2006\)](#) and [López-Sanz \*et al.\* \(2007\)](#).

### **An Approach to Develop Adaptable Services (Chang's Work)**

The method is based on SOMA and extends it to deal with service variability and mismatch. Three types of variability in service design (workflow, service composition, and logic) and three types of service mismatch (interface, functional, and non-functional) are used by the approach. Its analysis was based on [Chang \(2007\)](#) and [Chang and Kim \(2007\)](#).

---

<sup>2</sup><http://www.lia.deis.unibo.it/research/MIDAS/>

**Service Identification Using Goal and Scenario in SOA (Kim's Work)**

It presents a service identification method based on business goals, and uses scenarios to describe them. It also proposes a conceptual framework to elicit possible business changes in order to identify services that accomplish these changes as well. Its analysis was based on [Kim \*et al.\* \(2008\)](#).

**A Method for Engineering a True SOA (Engels' Method)**

This method defines the term (*true SOA*) and aims in providing concrete engineering methods to construct such an architecture. A true SOA is considered more than web services technology and its style of communication, and it means that changes in the business can be supported by the enterprise without major restructuring of its IT architecture. Its analysis was based on [Engels \*et al.\* \(2008\)](#).

**A SOA-based Software Architecture Process (RiSE Process)**

This process was developed at the RiSE Labs. The process provides a quality attribute driven and view oriented process for SOA-based architectures. The process uses a design by contract strategy to design services based on service-orientation principles, e.g., service cohesion and loose coupling. Its analysis was based on [Dias Jr. \(2008\)](#).

**Building Service-oriented Systems on the Web (Lamparter's Method)**

The methodology provides a comprehensive method that integrates service, market and ontology engineering processes to one service-oriented engineering methodology. Its analysis was based on [Lamparter and Sure \(2008\)](#).

**A Method for Service Identification (Azevedo's Method)**

It provides a method for the identification of service candidates from business process models. It presents a set of well-defined heuristics for service identification that were collected in a real industrial case study. Its analysis was based on [Azevedo \*et al.\* \(2009\)](#).

## 4.6 Results of the Systematic Review

In this section, we present the results of this review that was conducted with the objective of analyzing the current state-of-the-art of existing SOA design methodologies. The next sub-sections present the analysis of the questions investigated by the systematic review as well as a critical discussion of the methods, their activities, roles, inputs, outputs, concerns with quality attributes, delivery strategies and adaptation of existing processes.

### 4.6.1 Activities, Artifacts and Roles

Analyzing the methodologies, we could identify four activities that are commonly used by most of the service-oriented methods analyzed: service identification, service specification, architecture documentation and service realization. We also observed that very few methods define artifacts and roles during the process. The next sections present the activities, artifacts and roles identified during the analysis of the service-oriented methodologies included in the review.

#### Service Identification Activity

The identification of services is a challenging and critical task of the service-oriented development ([Azevedo et al., 2009](#)). Identify services having the right level of granularity can have a broad influence on the whole system and quality attributes, e.g., reusability and performance ([Erradi et al., 2006](#); [Kim et al., 2008](#)). In addition, well-defined and right-grained services are critical for achieving business agility and flexibility ([Carter, 2007](#)).

In this way, it is important to use complementary techniques to identify service candidates from different sources in order to avoid the identification of an incomplete set of services ([Arsanjani et al., 2008](#)). Analyzing the methods of the review, four different techniques for service identification were detected:

- **Business process modeling:** In this activity, the service candidates are identified from the business process models;
- **Existing assets analysis:** It focuses on the identification of services from existing legacy applications;
- **Goal-service modeling:** It consists of an analysis of the goals of a SOA project in order to identify service candidates that accomplish these goals;

- **Use Cases:** In this activity, the service candidates are identified from the use case descriptions.

We could observe that business process modeling is the most common activity for service identification, and it is presented in all the methods included in the review. The Azevedo's approach presents the most complete set of guidelines and heuristics for the identification of service candidates from business process models. These guidelines and heuristics were collected in a real industrial case study.

Only SOMA, OASIS, Chang's and Kim's work define activities for goal-service modeling. Kim's work is interesting because it uses scenarios to describe the business goals and considers the impact of business changes during the service identification activity. In other words, it identifies services to accomplish goals that may appear due to changes in the business environment.

All the methodologies included in the review, except Engel's and SOD-M, describe activities for analyzing existing assets in order to leverage legacy systems in the new service-oriented solution. This is an important issue since this is one of the major benefits of service orientation due to its interoperability capabilities ([Arsanjani et al., 2008](#)).

Only SOSE and Kim's work mention the identification of service candidates from use cases. However, none of them provide guidelines and steps to perform the service identification activity using the use cases. A systematic identification of services from use cases is missing in the methods selected for this review. A summary of the service identification techniques used in the methods as well as their usage percentage are presented in Table 4.4.

An important point to be considered during the service identification is the traceability among the services identified and the artifact that was used for their identification. The traceability link is essential to measure the impact of modifying an artifact, e.g., a business process or business goal, and its impact on the services identified. In the Kim's work, the traceability link among the business goals, business changes and the services identified are maintained explicitly.

### **Service Specification Activity**

During the service specification, three sub-activities were identified: service specification, component specification and decision modeling. All the methods analyzed outline activities for service specification. Only the Engel's work and SOMA describe activities for specifying the components that will implement the services. Chang's method is the only one concerned with decision modeling and variability.



**Table 4.4** Service Identification Techniques

Source	Methodologies	Usage
Business Process Models	All the methods analyzed in the survey.	100%
Existing Systems	All the methodologies analyzed, except Engel's and SOD-M.	85%
Business Goals	SOMA, OASIS, Chang's and Kim's work.	31%

In the methodology of Papazoglou, WSDL and policies are used to specify and document services using three different specifications:

1. **Structural:** The service types, messages and operations are documented;
2. **Behavioral:** Effects and side effects of service communication is defined;
3. **Policy:** Policy assertions and constraints are described.

SOSE uses Unified Modeling Language (UML) use case diagrams to specify services. Every business process is described as a use case. The behavior of each service is also modeled more specifically using scenarios. In SOAF, service description documents are used for service specification, where the service contract, service data model, service usage interface, service usage policy, service inputs and outputs, pre-conditions, effects and non-functional requirements are specified.

As a model-driven approach, SOD-M introduces some models to the MIDAS framework, and documents services using UML use case models and activity diagrams. The service discovery process in OASIS is documented using its specific notation. A template for service definition is provided consisting of the following documentation:

- Business owner, its roles and responsibilities over the business process;
- Actors, their roles and responsibilities;
- Tasks with their description, pre-condition, post-condition and invariants.

In SOMA, the detailed design of the services is elaborated during the service modeling activity. A service model is elaborated in terms of service dependencies, flows and compositions, events, rules and policies, operations, messages, non-functional requirements, and state management decisions. Exposed services and service operations are mapped into IT implementation as WSDL. It also uses UML-based diagrams to specify

**Table 4.5** Service Specification Artifacts

Document	Methodologies	Usage
WSDL	Papazoglou's, Erl's, SOMA and Chang's method.	31%
BPEL	Chang's work.	8%
UML	SOMA, SOSE and the Engel's framework.	23%

subsystems and functional components in order to model the static and dynamic behavior of the services that expose such subsystems and components.

The Erl's methodology specifies services using web services standards, such as WSDL and policies. In Chang's work, services are specified firstly in a technology independent form, later it suggests the use of WSDL. In addition, Business Process Execution Language (BPEL) and Business Process Language Notation (BPMN) are also used to specify service orchestrations and business processes.

Engel's method specifies services using UML use case diagrams and tabular prose description, which can be structured with the fields: service name, external view, service provider and consumer, pre/post conditions, course of action, trigger, result, non-functional requirements and internal view.

The methodologies use different techniques to specify and design services. However, many of them use WSDL to describe services, which emerges as the standard as it can be seen in Table 4.5.

### Architecture Documentation Activity

Analyzing the methodologies, we could notice that several methods do not suggest any model to document the service-oriented architecture. This is a strong drawback identified in the majority of the works analyzed. However, SOMA and the RiSE process provide a template to document the architecture as discussed next.

SOMA suggests a general template for the service-oriented architecture documentation. The template has seven sections, basically one section for each layer of the reference architecture defined in the methodology. SOMA uses the following layers:

1. **Operational system:** This section consists of descriptions of existing custom built applications, also called legacy systems;
2. **Enterprise components:** In this section, the enterprise components that are responsible for realizing the functionalities of the services are described;

3. **Services:** The services identified are described in this section;
4. **Business process choreography:** Compositions and choreographies of the services exposed are described in this section;
5. **Access or presentation layer:** Describes the way services will be accessed;
6. **Quality of Service:** This section describes the capabilities required to monitor, manage, and maintain the quality of service, such as security, performance, and availability of the SOA;
7. **Integration:** This section describes the integration capabilities, such as intelligent routing, protocol mediation, and other transformation mechanisms, often described as the Enterprise Service Bus (ESB).

The RiSE process also suggests a specific template to document the service-oriented architecture consisting of the following sections:

- **Documentation overview:** Shows how the architecture document is organized, i.e., describes each section of the document briefly;
- **SOA overview:** Depicts the description of the SOA, the purpose of the SOA solution and the quality attributes of the service-oriented architecture;
- **Services:** Describes the services identified, their service contract, i.e., WSDL and Service Level Agreement (SLA), and architectural views in the service level;
- **SOA views:** Contains the architectural views in the enterprise level produced to address the quality attributes of the SOA;
- **Glossary and acronym list:** Describes the main words and acronyms with their meaning in the document.

Both the RiSE process and SOMA present a well-structured and focused template for the architecture documentation. This characteristic seems appropriated, since service-oriented architectures should be well documented using different views in order to address the concerns of the different stakeholders involved in the project and the quality attributes of the SOA ([Ibrahim and Misie, 2006](#)).

### Service Realization Activity

In the realization activity, another three sub-activities were recognized: realization strategy selection, non-functional analysis and the design of an adaptation layer. SOMA, SOAF, RiSE, Lamparter's and the Papazoglou's methodology define a step in their processes where the realization strategy should be chosen. The same methods define also steps to analyze the impact of non-functional requirements in the service-oriented architecture. The method of Chang is the only one to delineate a step to design an adaptation layer to deal with variability in the service-oriented architecture.

Analyzing the methodologies, we could observe that very few methods explore and describe the realization strategies supported. Only SOMA and SOAF describe their strategies briefly. The realization strategies found in the service-oriented methodologies analyzed are:

- **Reuse of existing assets:** It involves selecting integration and transformation strategies such as creating adapters and wrappers of legacy functionalities to realize the desired services. It can be made also exploiting design patterns such as mediators, facade and factories ([Erradi et al., 2006](#));
- **Building from scratch:** In this strategy, services are developed from scratch using a bottom-up or top-down manner. In the top-down strategy, the contract documents (e.g., WSDL) are created, and then, the service implementation is developed to realize the contract. Conversely, in the bottom-up strategy, the service implementation is developed followed by the creation of the contract documents.

### Artifacts Produced During the Process

As mentioned before, few methodologies define specific artifacts during the process. OASIS defines a central model for service definition where the SOA is based on. SOMA and the RiSE process define a specific architecture document, and the Chang's work defines an acquisition plan in one of its steps.

Papazoglou's, Erl's, SOMA and Chang's work define WSDL documents for service specification, while SOMA, SOSE and the Engel's framework use UML diagrams. Chang's work also uses BPEL and BPMN for business process modeling.

SOD-M uses an extended use case model, represented with a use case model at a lower granularity; and the service process model, which is a kind of activity diagram used to represent the service processes. SOAF, Engel's and OASIS produce a service description document to describe the services identified.

**Table 4.6** Roles Involved in the Process

Role	Description
Enterprise Manager	Person who knows the enterprise business as a whole and manages the integration of business area teams.
SOA Architect	Specialist that knows about SOA concepts and technologies.
Application Architect	System architect of some internal or external area of enterprise that will provide or consume services.
Business Analyst	Person responsible for business processes of the some internal or external area of enterprise that will provide or consume services.
Business Manager	Person who manages an enterprise business area.

The other methods do not define any specific artifact to use during analysis and design of services. The use of intermediary models and diagrams for business and service modeling are implicit.

### **Roles Involved in the Methodologies**

Hardly any of the methods define the roles involved in the activities. OASIS discusses about the project manager, lead architect, lead business analyst, administrative assistance, technical support and an event coordinator interacting during the business process modeling activity. The RiSE process defines the enterprise manager, SOA architect, application architect, business analyst and business manager.

Lamarter's work discusses the importance of stakeholders representing service consumers and producers during certain steps in the process. However, most of the methods do not detail the responsibilities of the roles during the process. Table 4.6 presents the characteristics and responsibilities of the roles that were described in the methodologies.

#### **4.6.2 Quality Attributes**

Only a few of the methodologies analyzed pay attention to architecture quality attributes. Some, for the nature of its purpose mention that arbitrarily will not consider other problems than service-oriented related. That is the case of the OASIS, that focuses exclusively on service-orientation, leaving gaps intentionally to be covered by another design discipline that must be used together to obtain a full blueprint.

SOMA, Erl, Lamarter and the RiSE process provide guidelines to deal with the accomplishment of quality attributes during architecture design. In SOMA, a separate

layer in the architecture is designated to monitor, manage and maintain QoS requirements such as security, performance and availability. In the Erl's methodology, some advice about maintainability and reusability can be found under the names of service-oriented principles. The RiSE process has a phase in architecture development to identify quality attributes and another phase to address those attributes.

Most of the methods do not provide proper guidelines to promote reuse among services. The RiSE, OASIS and the Erl methodology provide some advice and instruct the designer to look for reuse among services as a way of optimizing the design. The Erl's methodology also illustrates different situations where there can be reuse in a service-oriented architecture: cross-process, intra-process and cross-application reuse.

Reusability is taken into account in Kim's work during the service identification activity. It identifies services from business goals and advocates that services providing implementation for specific business goals can be reused by other services that implement more coarse-grained business goals. Likewise in Erl's, in which reusability is considered by analyzing business activities or group of activities that are common for several business process. In the Azevedo's work, it is suggested an integrated view of the business processes, in which dependencies and commonalities among processes are represented explicitly and can be useful to identify reusable building blocks of business activities.

Chang's work appears a step further and apply the implementation of variability mechanisms among services with the purpose of increasing reuse. The lack of guidelines for reusability and other quality attributes can be seen as a gap to be fulfilled.

The ideas of SOMA, Erl's and the Chang's work seem a nice path to follow with the purpose of obtaining a more complete service-oriented design discipline.

### 4.6.3 Delivery Strategy

Analyzing the methods selected for the review, we could detect some delivery strategies that are currently being used to build (deliver) services. The following delivery strategies were identified ([Erl, 2005](#); [Papazoglou and Heuvel, 2006](#)):

- **Top-down:** It advocates the completion of an inventory analysis prior to the physical design, development, and delivery of services;
- **Bottom-up:** It is tactically focused on the fulfillment of immediate business requirements as priority and the prime objective of the project;
- **Meet-in-the-middle:** It combines the top-down and bottom-up strategies in order to get the benefits and minimize the risks of both.

**Table 4.7** Service Delivery Strategies

Strategy	Methodologies	Usage
Bottom-Up	The SOSE framework.	8%
Top-Down	OASIS, SOD-M, Engel's and the Lamparter's work.	31%
Meet in the Middle	SOMA, Erl's, Papazoglou's, Chang's and the RiSE process.	38%

The top-down strategy increases the initial cost, effort and time because of an inventory blueprint that is established prior to the delivery of services. Conversely, the bottom-up strategy requires subsequence governance with increased costs, effort and time, as the services are delivered based on immediate project requirements (Erl, 2005, 2007).

The meet-in-the-middle strategy is present in most of the methods selected as shown in Table 4.7, which presents the delivery strategies used in each method as well as their usage percentage. We see this as a natural tendency, once this delivery strategy fits well in many different contexts.

#### 4.6.4 Adaptation of Existing Processes

Some methodologies try to minimize the business changes in the beginning of the adoption producing service-oriented architectures and adapting existing processes. This is a good point since it minimizes the companies cost and effort during the process adaptation. Other methods specify a totally new methodology for developing service-oriented architectures requiring a bigger effort to adopt it.

SOMA and the methodology of Papazoglou are based on the Rational Unified Process (RUP). The methodology of Papazoglou and the SOSE framework combine Service-Oriented Development (SOD) and Component-Based Development (CBD). In this combination, enterprise components, such as Enterprise Java Beans, are used to implement the functionalities that will be exposed by the services.

The work of Chang is based on SOMA, and it uses concepts of Software Product Lines (SPL), such as variability, in order to develop service-oriented applications that can be adapted to different contexts. The work of Lamparter proposes an integrated method that merges three engineering methodologies: web service/software engineering, market engineering and ontology engineering. And finally, SOD-M is based on the MIDAS, a framework for Model-Driven Development (MDD), making it possible to develop service-oriented applications in a model-driven way.

**Table 4.8** Summary of the Review

	SOMA	Erl's Methodology	OASIS	SOSE	Papazoglou	SOAF	SOD-M
Delivery Strategy	Meet in the middle	Top-Down, Bottom-Up and Meet in the middle	Top-Down	Top-Down	Top-Down, Bottom-Up and Meet in the middle	Meet in the middle	Top-Down
Adaptation of Existing Processes	RUP				RUP		MIDAS
Architecture Documentation	Specific Template						
Experimentation in Industry	Different projects and domains	Yes	Yes	Yes		Yes	Yes
Experimentation in Academy		Yes					
Guidelines for Quality Attributes	Specific layer to deal with quality attributes	Method based on service-oriented principles	Concerns with reusability		Concerns with reusability		

	Chang	Kim	Engels	RiSE	Lamparter	Azevedo
Delivery Strategy	Meet in the middle		Top-Down	Meet in the middle		
Adaptation of Existing Processes	SOMA				Marketing and Ontology Engineering	
Architecture Documentation				Specific Template		
Experimentation in Industry	Yes		Yes		Yes	Yes
Experimentation in Academy				Yes		Yes
Guidelines for Quality Attributes	Concerns with quality attributes	Concerns with quality attributes		Specific activity to deal with quality attributes		Concerns with quality attributes and directions for commonality and variability analysis

Relating to ease of adoption, the SOMA seems worth looking as it fits well with the RUP philosophy, which is worldwide spread. Moreover, both SOMA and RUP are currently being controlled by IBM.

## 4.7 Chapter Summary

This chapter presented a systematic review on SOA design methods. A deep analysis of the methodologies considering the analysis of their weaknesses and strengths, and gaps in the area was performed. Table 4.8 summarizes the review. A survey on SPL processes can be found in (de Almeida, 2007) and a systematic review is presented in (Souza Filho *et al.*, 2008).

By analyzing the activities, artifacts, architecture documentation, concerns with reusability and other quality attributes, delivery of services and adaptations of existing processes or methodologies, we could observe that a set of good practices, such as goal-driven and business process modeling, are being used in the existing methods.



We could notice, though, a certain lack of concerns with reusability and other quality attributes because the guidelines provided are not well detailed. Several approaches just comment it briefly and no guidelines are given to achieve quality attributes. We believe that the absence of these guidelines is a gap that needs to be addressed and an interesting research topic.

The same thing occurs during architecture documentation, several methodologies do not suggest any model or notation, or even comment about the documentation of the architecture. However, few approaches suggested their specific template for documentation.

As we could observe, some of the methods are adapted from an existing process or methodology. We consider it as a strong point because it can ease the adoption of the process for companies that already use the adapted process, such as RUP. However, it is strongly recommended that the method does not change the main characteristics of the adapted process.

In our point of view, the maturity of the methods can be determined by its experimentation in academy or industry. Thus, we trust that methods that have not been validated largely in industrial environments and different domains are not mature, and must be deeply and carefully analyzed to ensure their quality and consistency for real industrial projects usage.

In this path, we consider the SOMA to be a very mature and comprehensive methodology. Other methods that are undoubtedly worth seeing are: Erl's, for its primary, yet somewhat vague guidelines; and Chang's work for its concern about variability and commonality.

The next chapter presents an approach to design service-oriented product line architectures (SOPLE-DE), as well as its activities, sub-activities, roles, inputs, outputs and guidelines.

# An Approach to Design Service-Oriented Product Line Architectures

## 5.1 Introduction

The SOPLE-DE (Medeiros *et al.*, 2009) is part of a bigger process, called Service-Oriented Product Line Engineering (SOPLE), which is concerned with the full life-cycle of service-oriented product line engineering, including, scope, requirement, design, implementation and testing. In this dissertation, the SOPLE-DE, which deals with the architectural aspects of the service-oriented product line development, will be described.

The main goal of the SOPLE-DE is to produce a Service-Oriented Product Line Architecture (SO-PLA), which represents the common and variable architectural elements of a specific domain. In order to define a SO-PLA, a process is essential to provide guidance to the team, specify the artifacts to be produced, and associate activities with specific roles and the team as a whole (Booch, 1995).

The SOPLE-DE receives the feature model, business process models and the quality attribute<sup>1</sup> scenarios as mandatory inputs. The domain use cases are received as inputs when available, i.e., they are optional inputs. In the context of the SOPLE process, the SOPLE-Requirement process provides these inputs. However, the SOPLE-DE is flexible, and can be used with other service-oriented product line requirement process that fits the inputs required by it.

---

<sup>1</sup>In this work, the term quality attribute will be adopted instead of non-functional requirement. It was decided because this term is better to represent the essential characteristics of an architecture, e.g., this architecture is flexible, or it is modifiable.

The remainder of this chapter is organized as follows: Section 5.2 presents the principles used as basis to define the SOPLE-DE; In Section 5.3, an overview of the SOPLE-DE is described; Section 5.4 discusses the activities, tasks, inputs, outputs and roles of SOPLE-DE in details; finally, Section 5.5 concludes this chapter with its summary.

## 5.2 Principles

In order to provide a systematic way to perform the service-oriented domain design, SOPLE-DE is based on a set of design and product line principles, and it combines feature-oriented, component-based, process-oriented and service-oriented development. The principles used by the SOPLE-DE are discussed next.

### 5.2.1 Component-Based Development (CBD)

A component can be defined as an encapsulated part of a system, nearly independent, replaceable, reusable, and subject to third-party compositions (Szyperski, 2003). Components have well-defined (provided and required) interfaces, and fulfill a clear function of the system (Kruchten, 2003). An example of component can be an Enterprise Java Bean (EJB) (Panda *et al.*, 2007).

In this way, components are used as the basic architectural elements of the service-oriented product line architecture. In other words, the architectural components provide the functionalities that will be exposed by the services (Arsanjani *et al.*, 2008). Moreover, CBD can be used as a variability mechanism. Thus, service-oriented product line variability can be implemented changing architectural components (van Gurp *et al.*, 2000).

### 5.2.2 Feature-Oriented Development (FOD)

SOPLE-DE considers a feature as a logical unit of behavior that is specified by a set of related functional or non-functional requirements and represents a valuable aspect to the customer (Bosch, 2000). FOD is used in this approach motivated by the fact that features are a natural and intuitive way to express commonality and variability among products. Moreover, features are abstractions that both customers and developers can understand easily (Kang *et al.*, 2002).

In this sense, the SOPLE-DE approach uses feature models to represent the common and variable functionalities of the service-oriented product line in an exploitable way (Lee *et al.*, 2008). Thus, different customers, architects and developers can visualize what is common and what varies in the product line using a single model, i.e., the feature model. In this way, customers can select the desired features for their products, and architects and developers can understand, design and implement the variation points of the product line easier than without this general view.

### 5.2.3 Process-Oriented and Service-Oriented Development

Service orientation is an emergent paradigm in which application functionality is exposed using a set of reusable services. In the context of service-oriented development, services provide a flexible way to expose discrete business functions, and therefore, a viable way to develop applications that support business processes (Brown *et al.*, 2003).

Thus, the SOPLE-DE uses Process-Oriented Development (POD), and analyzes the domain business processes to identify architectural services (Boffoli *et al.*, 2009). The functionality exposed by these services will be provided by the architectural components as already mentioned (Arsanjani *et al.*, 2008). In addition, service-oriented product line variability can also be implemented changing architectural services.

### 5.2.4 Separation of Concerns and Information Hiding

The use of components and services allows the separation of concerns in specific self-contained and replaceable building blocks. However, the use of CBD and SOD is not enough to separate concerns efficiently. In this way, SOPLE-DE considers the separation of concerns during the design and provides guidelines for it. Thus, features and business process activities can be encapsulated into components and services.

The use of these architectural elements, i.e., components and services, also encourages information hiding due to the fact that they expose functionality through well-defined interfaces and maintain their internal logic hidden (Szyperski, 2003; Erl, 2005).

### 5.2.5 Commonality and Variability

SPL explores the commonality and variability among a set of related products (Pohl *et al.*, 2005). In this context, SOPLE-DE allows the commonality and variability, presented in the feature model, business process models, use cases and quality attribute scenarios, to be represented and mapped in the SO-PLA.

SOPLE-DE considers variability at the component and service levels. It extends the notion of variability into the service-oriented development activities, which is not considered by several existing service-oriented methodologies, e.g., [Erl \(2005\)](#) and [Papazoglou and Heuvel \(2006\)](#). Thus, the services produced in the SOPLE-DE can be developed with variability, i.e., a service can be customized to specific contexts based on customer requirements through variation points. Moreover, the artifacts produced, e.g., architectural views and other diagrams, can also contain variability.

### 5.2.6 Quality Attributes

The accomplishment of architectural quality attributes is crucial for certain types of systems. In these cases, merely satisfy the functional requirements are not enough ([Barbacci et al., 1995](#)). For instance, critical systems in general, must satisfy security, safety, dependability, performance, and other similar quality attributes as well.

In the context of software product line architectures, some quality attributes, e.g., flexibility and evolvability are essential ([Pohl et al., 2005](#)). Moreover, SPL as a systematic and planned reuse strategy explicitly considers reusability due to the use of two development cycles ([Clements and Northrop, 2001](#)). In the core asset development, the artifacts are produced generically with variability, and later, they are reused and customized to specific contexts during product development.

In this sense, SOPLE-DE uses the separation of concerns, information hiding, managed variability, CBD and SOD in a systematic way with the purpose of improving flexibility and evolvability. Thus, features and business process activities, when encapsulated in specific components or services, can be improved or modified locally without affecting other elements of the architecture. In addition, SOPLE-DE provides guidelines to satisfy other quality attributes, e.g., reusability and performance, during the process. The use of SOA concepts also encourages interoperability due to the characteristics of services, flexibility and business agility ([Erl, 2007](#); [Carter, 2007](#)).

### 5.2.7 Cohesion and Granularity

Cohesion and granularity are concepts that directly impact some architectural quality attributes, e.g., performance and reusability ([Erradi et al., 2006](#)). Cohesion is the degree of functional relatedness of the operations contained in the interface of a service or component, while granularity refers to the scope of functionality exposed by a service or component ([Papazoglou and Heuvel, 2006](#)).

For instance, if services are too coarse-grained, the size of the exchanged messages grows and sometimes might carry more data than needed. On the other hand, if services are too fine-grained, multiple message exchanges may be required to perform specific functionalities causing performance problems (Erradi *et al.*, 2006). In this sense, SOPLE-DE provides guidelines to consider cohesion and granularity during the architecture design in order to avoid performance problems and increase reusability.

### 5.2.8 Top-Down and Proactive Development

The SOPLE-DE uses a top-down way to identify and design services. This strategy advocates the completion of an inventory analysis prior to the physical design, development, and delivery of services (Erl, 2005). In this way, SOPLE-DE does not consider the bottom-up strategy, which is concerned with the fulfillment of immediate business requirements and integration of legacy applications (Arsanjani *et al.*, 2008).

In the context of SPL engineering, the SOPLE-DE uses a proactive adoption model, in which all product variations on the foreseeable horizon up front are analyzed, architected and designed. This adoption model suits organizations that can predict their product line requirements well into the future and that have the time and resources for a long development cycle (Krueger, 2002). The extractive and reactive adoption models, which are concerned with existing products and incremental development respectively, are not considered in SOPLE-DE.

### 5.2.9 Systematic Sequence of Activities

The last principle, and not less important, is to provide a systematic sequence of activities, which are divided into a set of tasks with well-defined inputs and outputs, and performed by a predefined set of roles with clear responsibilities. The purpose of this systematization is to ease the use and adoption of the SOPLE-DE in practice.

## 5.3 SOPLE-DE Overview

The SOPLE-DE is a top-down approach for the systematic identification, design and documentation of service-oriented core assets supporting the non-systematic reuse of service-oriented environments. It also uses SOA concepts, e.g., dynamic service discoverability, to aid the development of flexible and dynamic software product lines.

Next sub-sections present an overview of the inputs, outputs, roles, architectural style, development cycles, and activities of the SOPLE-DE.

### 5.3.1 Inputs and Outputs

The SOPLE-DE receives the feature model, business process models and quality attribute scenarios as mandatory inputs. The domain use cases are also considered as inputs when available. The feature model explicitly represents the commonality and variability of the service-oriented product line, i.e., common and variable parts of the domain. Thus, it provides the basis for designing, developing and configuring reusable core assets ([Kang et al., 2002](#)).

In the context of service-oriented product lines, the business process models can contain variability, i.e., business process activities can be marked as mandatory, optional or alternative ([Ye et al., 2007](#); [Boffoli et al., 2008](#)). The models represent the business processes for a specific domain that should be automated by the service-oriented product line. These models are usually modeled using Business Process Modeling Notation (BPMN) or UML activity diagrams ([Razavian and Khosravi, 2008](#)).

The quality attribute scenarios represent the non-functional requirements that should be satisfied at the architecture level, and SOPLE-DE requires them to be prioritized previously according to their importance to the product line architecture. In this way, architectural quality attribute trade-offs can be solved accordingly. In addition, SOPLE-DE considers that specific products in the line may require different quality attributes. Thus, quality attributes can be considered as a variation point of the service-oriented product line ([Etxeberria et al., 2007](#)).

However, it is important to note that a quality attribute, e.g., flexibility or evolvability, does not say exactly what it means. Thus, SOPLE-DE requires quality attributes to be specified using scenarios. Quality attribute scenarios can be described using the following parts ([Bass et al., 2003](#)):

- **Source of stimulus:** It is an entity that generates a stimulus, e.g., a human or a computer system;
- **Stimulus:** It is an action, generated by the source of stimulus, that arrives at a system and has to be considered;
- **Environment:** It is the condition of the system when the stimulus occurs. For instance, the system may be in an overload condition or may be running normally when the stimulus occurs;

- **Artifact:** Some artifact is stimulated, it may be the whole system or pieces of it;
- **Response:** The response is the activity undertaken after the arrival of the stimulus;
- **Response measure:** When the response occurs, it should be measurable in some way to test the quality attribute.

A use case represents the behavior of a system when responding to an action originated by an actor, i.e., source of stimulus (Kruchten, 2003). In the context of service-oriented product lines, use cases can also contain variability. The SOPLE-DE encourages use cases to be described using the following parts: actor, pre/post conditions, main flow, alternative flow and invariants.

The use cases are not used in the service-oriented development normally, in which the business processes represent the requirements and the focus is on business processes rather than use cases (Abu-Matar, 2007). However, SOPLE-DE considers the use cases as optional inputs because they provide detailed (additional) information that may not be presented in the business processes.

The SOPLE-DE encourages the quality attribute scenarios and use cases to be described as presented previously. However, as commented before, the approach is flexible and can be used together with others requirement processes that provide these inputs, even using different formats. It will not impact the SOPLE-DE, but the use cases and quality attribute scenarios should be detailed carefully in order to ensure a good understanding of the requirements and quality attributes by the stakeholders.

The output of the SOPLE-DE is an architecture document describing the SO-PLA, with traceability links among the architectural models, e.g., architectural views, and the requirement models such as the feature model and business process models. Appendix A provides an architecture document template to document the SO-PLA.

### 5.3.2 Roles

A role defines the behavior and responsibilities of an individual or group of people working together as a team. The behavior is expressed in terms of tasks the role performs, and each role is associated with a set of cohesive tasks (Kruchten, 2003). The SOPLE-DE uses the following roles:

- **Business Analyst:** Business specialist responsible for the business processes of the domain. The business analyst makes sure that all the business logic is represented in the design;



- **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;
- **SOA Architect:** Specialist in SOA modeling, and responsible for the design of the service-oriented architecture as whole, i.e., identification, design and documentation of the architectural elements considering the quality attributes that should be satisfied;
- **Domain Designer:** Person responsible for defining the responsibilities, operations, attributes, and relationships of classes and determining how they should be adjusted to the implementation environment;
- **Service Designer:** Specialist in service modeling, and responsible for defining the responsibilities, quality attributes and general documentation of services and service compositions.

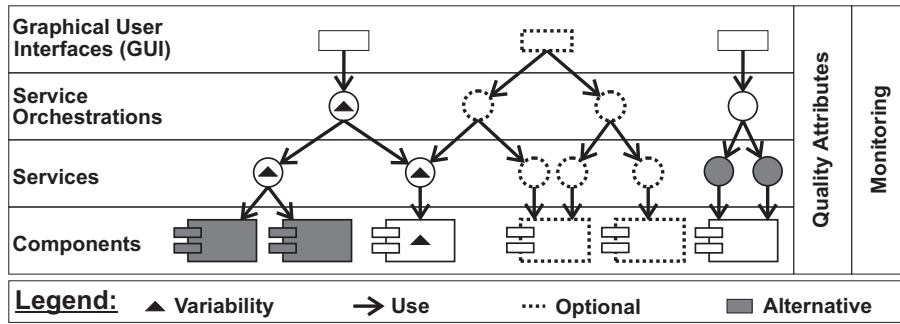
All the roles of SOPLE-DE are mandatory. However, the same person can play more than one role in a specific project, e.g., the same person can be the SOA architect and service designer, since the activities performed by these roles share common characteristics and they are performed in different times.

### 5.3.3 Architectural Style

The SOPLE-DE considers the architectural style shown in Figure 5.1. This architectural style presents the layers that are commonly used in SOA (Arsanjani, 2004). However, other authors use different taxonomies and some define additional layers as described next.

For instance, Erl (2005) divides the service layer into task, entity and utility service layers, while Papazoglou and Heuvel (2006) uses the term business processes layer instead of orchestration service layer, and divides the service layer into business and infrastructure service layers. The work of Arsanjani *et al.* (2007) uses additional layers, e.g., operational system layer, since it considers the integration of legacy applications that is not taken into account in SOPLE-DE.

The interface layer is composed of Graphical User Interfaces (GUI) components. Only service-oriented product lines that require visual interfaces to interact with the services and service orchestrations may use this layer. In addition, the visual interface



**Figure 5.1** Architectural Style

components may be specific for each system, in this way, they will not be considered as core assets in some product lines.

The orchestration service layer consists of composite services, which implement coarse-grained business activities, or even an entire business process, that need the participation and interaction of several fine-grained services. The service layer is composed of self-contained and business-aligned services, which implement fine-grained business activities. The component layer consists of a set of components that provide functionality for the services and maintain their Quality of Service (QoS).

The quality attribute layer consists of additional architectural elements responsible for satisfying specific quality attributes, e.g., performance and security. Finally, the monitoring layer, which is responsible for monitoring the health of the SOA solution, e.g., monitoring the number of service calls, the response time of services and number of service errors.

It is important to note that the architectural elements of these layers are developed taking variability into account, and they can be mandatory, optional or alternative. In addition, specific metrics can be monitored in different systems and each system may be customized to satisfy specific quality attributes.

The SOPLE-DE considers that service-oriented product line architectures support two variability levels as described next (Ye *et al.*, 2007; Boffoli *et al.*, 2008):

1. **Configuration variability:** Architectural elements are selected from the core assets in order to obtain the target system, i.e., optional and alternative architectural elements are selected or excluded from the architecture;
2. **Customization variability:** Architectural elements already selected for a system are customized according to the requirements of the specific system, i.e., architectural elements with variability are customized internally.

### 5.3.4 Development Cycles

The SOPLE-DE is divided in two life cycles as software product line engineering: core asset development and product development. The core asset development aims to provide guidelines and steps to identify, design, document and implement generic architectural elements, i.e., components, services and service orchestrations, with variability. During the product development cycle, these architectural elements are specialized to a particular context according to specific customer requirements or market segments needs ([Clements and Northrop, 2001](#); [Pohl et al., 2005](#)).

However, the focus of this dissertation is on the core asset development cycle, in particular, on the design of service-oriented product line architectures. Thus, the SOPLE-DE provides guidelines and steps for the identification, design and documentation of components, services, service orchestrations and their flows. In addition, SOPLE-DE considers variability in these architectural elements and in their communication, e.g., different protocols of communication, and synchronous or asynchronous message exchange.

### 5.3.5 Activities

The SOPLE-DE starts with the architectural elements identification activity. It receives the domain feature model, the business process models and the quality attribute scenarios as mandatory inputs. The domain use cases are optional inputs for this activity. It produces a list of components, services and service orchestration candidates for the product line architecture. Moreover, the communication flows among these elements are also identified.

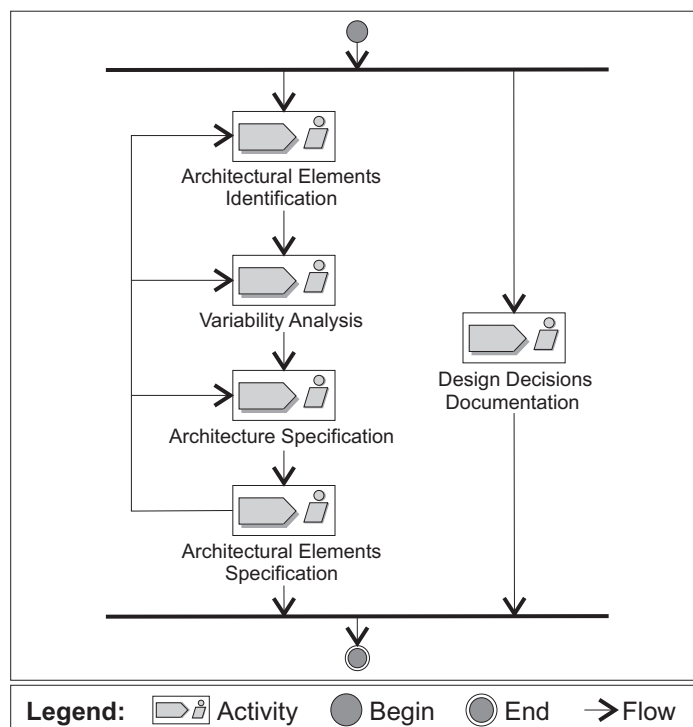
Subsequently, there is a variability analysis activity. It receives the list of components, services, service orchestrations and their communication flows identified previously, and defines and documents key architectural decisions regarding variability. In this activity, it is defined how the variability, presented in the feature model, business process models, use cases and quality attribute scenarios, will be implemented. The variability analysis activity refines the architectural elements identified previously.

Architecture specification is the next activity of the SOPLE-DE. In this activity, the architecture is documented using different views in order to represent the concerns of the different stakeholders involved in the project ([Bass et al., 2003](#)). An architecture is a complex entity that should be represented and documented upon several views.

The architectural elements specification activity is performed after the specification of the architectural views. During the architectural elements specification activity, the

low-level design of components and services is realized. SOPLE-DE suggests some UML diagrams to document the internal behavior of the architectural elements.

In parallel with these four activities described, the design decisions documentation activity is performed concurrently. In this activity, important design decisions, e.g., selection of technologies and variability mechanisms, are documented. Figure 5.2 shows the activities of the SOPLE-DE. As it can be seen, its activities are executed interactively, i.e., the activities do not have to be performed in a sequential order until their conclusion, and the execution flow can go back to previous activities when necessary (Kruchten, 2003).



**Figure 5.2** SOPLE-DE Activities

## 5.4 The SOPLE-DE Approach

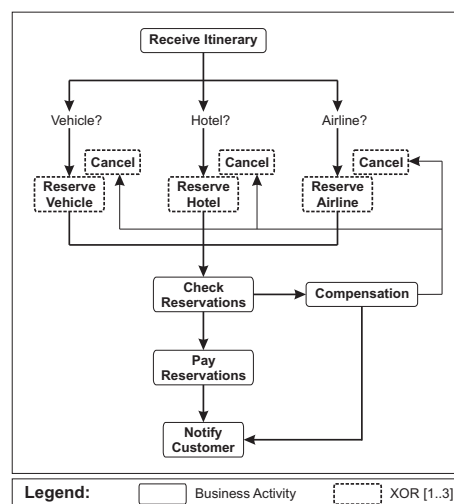
In this section, the activities, tasks, inputs, outputs and roles of the SOPLE-DE are presented in more details. In order to clarify its activities and tasks, an example on the *Travel Reservation* domain will be used.

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 the key goals through the development and deployment of its services (Snell, 2002):

1. The product line should allow customers to submit travel itineraries and payment information to the product line services using a Web interface;
2. The travel agency services should automatically obtain and reserve the appropriate services for the airline, hotel or vehicle according to the customer itineraries;
3. Its services should perform compensation operations for canceling itinerary failures;
4. 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 general business process that should be supported by the systems in the product line is shown in Figure 5.3.

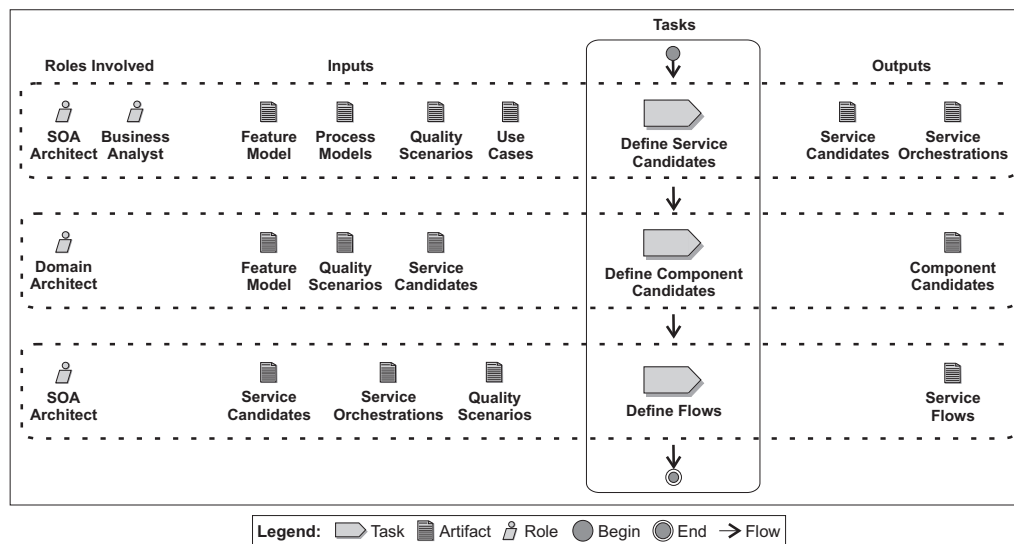


**Figure 5.3** The Project used as Example

### 5.4.1 Architectural Elements Identification Activity

In this activity, the architectural elements of the service-oriented product line (components, services and service orchestrations) and their flows will be identified. The architectural elements identification activity receives the feature model, business process models and quality attributes scenarios as inputs. The domain use cases are also received as inputs when available. It produces a list of components, services, orchestrations and their associated flows as output, and this list is documented in the architecture document.

The architectural elements identification activity is divided in three tasks: define service candidates, define component candidates and define flows. Figure 5.4 shows the inputs, outputs, tasks and roles of this activity, and the next sub-sections present its tasks in detail. All the activities of the SOPLE-DE are described using this notation, which was created based on (Kruchten, 2003).



**Figure 5.4** Architectural Elements Identification Activity

#### 5.4.1.1 Define Service Candidates

The identification of service candidates is a challenging and crucial task of service-oriented computing (Lee *et al.*, 2008). In the context of service-oriented product lines, the service identification task is even harder due to concerns with commonality and variability.

In the service identification task, a set of services and service orchestrations that support the business processes is defined. Thus, it is reasonable to identify these services

considering the business process models (Erl, 2005; Erradi *et al.*, 2006). However, according to Arsanjani *et al.* (2008), the identification of service candidates should be performed using complimentary techniques to avoid the identification of an incomplete set of services.

In particular, SOPLE-DE combines complimentary service identification techniques that use different sources to identify services, e.g., business process models, use cases, feature models and quality attribute scenarios. This task starts with an analysis of the business process models, feature model and quality attribute scenarios received as inputs. The service identification from the use cases is optional.

The service identification task produces as output a list of service and service orchestration candidates for the architecture. A service candidate is defined as an abstract, not implemented service, which during the design phase of a service life-cycle model, can be designed, and then implemented as a service or discarded (Erl, 2005).

The role responsible to perform the service identification task is the SOA architect. However, the business analyst should review the service and service orchestration candidates in order to ensure an accurate representation of the business logic (Erl, 2007).

The identification of services using each technique can be executed concurrently. However, at the end of the service identification task, the service candidates identified should be consolidated, e.g., duplicated services are discarded. Next sub-sections present the service identification techniques of SOPLE-DE.

### **Define Services from Business Processes**

The steps that should be performed in order to define the architectural services and service orchestrations from the business processes are: *identify automatic business activities* and *analyze business activities interactions*.

In the first step, automatic and partially automatic business process activities will be identified from the business process models. Automatic business process activities are performed entirely by a system with no manual interference, while partially automatic activities are executed manually, but supported by a system (Azevedo *et al.*, 2009). Manual activities are ignored because they usually cannot be implemented by the services (Havey, 2005). Initially, each automatic and partially automatic business process activity is considered as a service candidate.

However, in the context of service-oriented product lines, the business processes can be modeled with variability. Thus, some business activities may be marked as mandatory, optional or alternative. Hence, the services identified from the business activities will

be classified according to the type of the activity they implement, i.e., a service that implements an optional business process activity will be considered as optional and will not be presented in all systems of the product line.

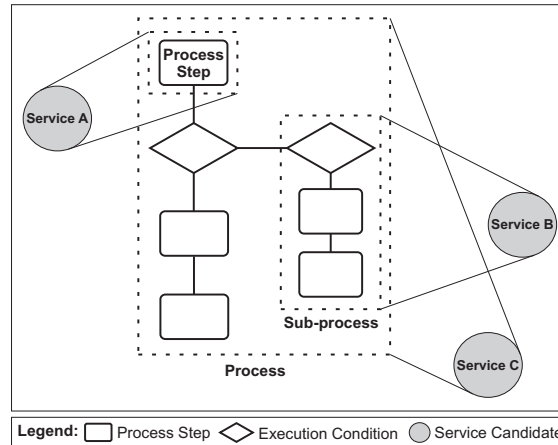
The first step towards the identification of service orchestrations is the analysis of interactions among business process activities. In this step, the list of automatic and partially automatic business activities produced previously will be analyzed to identify related activities that need to be executed in special conditions, such as the interaction patterns described next ([Lee and Kang, 2003](#); [Havey, 2005](#)):

- **Sequential:** Business process activities that must be executed in a pre-specified sequential order;
- **Concurrent:** Activities that must be executed concurrently in order to perform a functionality correctly;
- **Exclusive:** Business process activities that cannot be activated during the execution of other business activities;
- **Subordinate:** Activities that can be activated only if another business activity is being executed;
- **Loop:** Business process activities that must be executed until a condition becomes false, or executed a predefined number of times;
- **Optional:** A business process activity that will be executed according to conditions;
- **Alternative:** A set of business process activities in which only one will be executed depending on conditions.

These interaction patterns mentioned usually require service orchestrations in order to control and execute the related business process activities correctly. A business process as a whole, for instance, usually requires service orchestrations in order to execute the business activities as they are described in the business process model. A business process, commonly, may contain several combinations of sequential, concurrent, exclusive and other interaction patterns.

In Figure 5.5, [Erl \(2005\)](#) demonstrates that services can be defined with different levels of granularity, and they can be orchestrated in several ways. Thus, the SOA architect has to analyze each situation and decide a solution for its own project in order to satisfy the quality attributes required.





**Figure 5.5** Service Granularity

For instance, two examples are given with the purpose of providing guidance to satisfy quality attributes. In the first example, each service is used to implement a specific business process activity (process step), as Service A in Figure 5.5. In this way, each business activity can be modified without impact other activities, which is appropriated to achieve flexibility and evolvability. However, if the granularity of the business activities is low, the number of services and the message exchanged among them may increase dramatically impacting performance.

In the second example, it is assumed that the interaction pattern implemented by Service B (see Figure 5.5) appears in several processes. In this way, it can be explored in order to increase reusability. Thus, this interaction pattern can be isolated in a unique service to be reused in different contexts. Moreover, this service can be designed with variability to be adapted to specific contexts.

However, the activities of this interaction pattern can also be implemented by two or three services (one for each process step) and a service orchestration to control their execution. It is a design decision that should be made by the architect considering different factors, such as the cohesion and granularity of the business activities contained in the pattern.

The SOA architect has to decide the best way to encapsulate business process activities and interaction patterns into services and service orchestrations considering the quality attribute scenarios received as input. It is important to note that quality attribute trade-offs may occur, since satisfy one quality attribute might impact others. Thus, the SOA architect has to consider the priority of the quality attributes to solve these trade-offs.

Considering the business process depicted in Figure 5.3 as an example, different service candidates can be identified. For instance, a service for each type of reservation, i.e., airline, accommodation and vehicle, a service for payment issues, and a service to notify customers. In addition, the reservation activities need a service orchestration to execute these activities in parallel and control failures. In other words, if an airline reservation is performed successfully, but an accommodation is not found, it is necessary to cancel the airline reservation. Thus, this service orchestration is essential to control the reservation services and implement these functionalities correctly.

### Define Services from Features

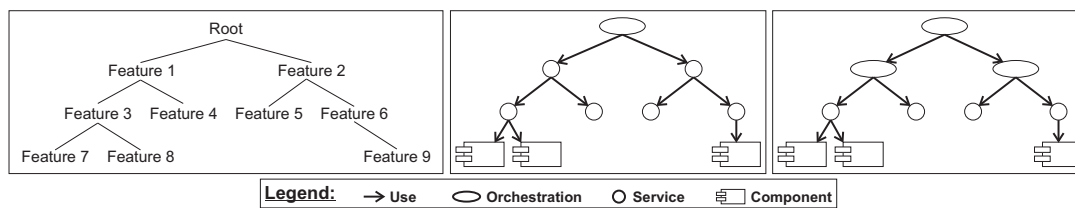
The identification of services and service orchestrations from the feature model should be performed using the concept of service feature, which is a major functionality of a specific domain that can be added or removed of systems, and configured independently of other features (Lee and Kang, 2003). Each service feature identified is considered as a service candidate or service orchestration depending on its characteristics, purposes and granularity.

The service features will dictate the granularity of the services identified. These services will be marked as mandatory, optional or alternative depending on the type of the service feature that originates them. They also may contain optional and alternative sub-features, which may impose variation on the service design. These sub-features may become the components that provide functionality for the services as described in the component identification task. However, service features considered as service candidates should share the following characteristics:

- **Stateless:** Service features should encapsulate functionalities that contain only stateless requirements. It is motivated because services should be as more stateless as possible in order to increase service reusability and scalability (Erl, 2005);
- **Autonomous:** Service features should encapsulate features that have control over their internal logic, since services should be autonomous and self-contained (Erl, 2005);
- **Coarse-grained:** Services slower down performance due to the remote calls (Joutsittis, 2007). For this reason, service features considered as service candidates should encapsulate coarse-grained functionalities to reduce the message exchanged among services;

- **Interoperable:** Service features encapsulating functionalities that will be reused by other services developed using different programming languages should be considered as service candidates.

For instance, Figure 5.6 presents two possible ways to use the feature model to identify services, orchestrations and components. As it can be seen, service orchestrations control the execution of services, while services use the functionalities that are provided by the architectural components. In this sense, orchestrations should be more coarse-grained than services, which should be more coarse-grained than components.



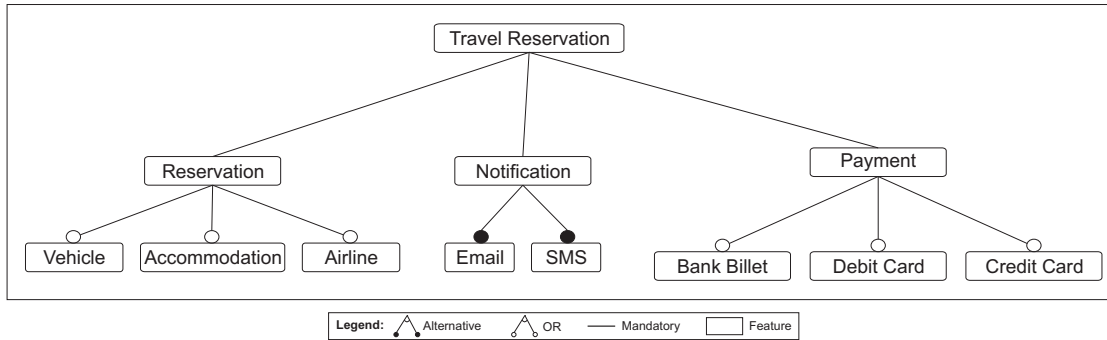
**Figure 5.6** Identification of Architectural Elements from Features

It is important to note that service features can be implemented by a service orchestration, service or component. However, the service features with the characteristics described before (e.g., interoperable and stateless), should be implemented by a service orchestration or service depending on their granularity and characteristics as already commented. For instance, service features that control the execution of other fine-grained service features should be considered as a service orchestration. Conversely, service features without those characteristics should be considered as architectural components as explained in the component identification task.

Figure 5.7 gives an example of a feature model for the travel reservation product line. In this sense, the following service features can be identified: airline, vehicle and accommodation features, notification and payment. Thus, these features can be considered as service candidates, while their sub-features can be considered as component candidates.

### Define Services from Use Cases

In this step, the key business entities of the domain should be identified using the use cases, when received as inputs. These entities are directly manipulated by several services and need specific services to implement their life-cycle operations, e.g., create, delete, update and retrieve (Erl, 2007). As the use cases are optional inputs, if they were not



**Figure 5.7** Travel Reservation Feature Model

provided, this identification will be realized from the business process and feature model, which may turn this step more difficult.

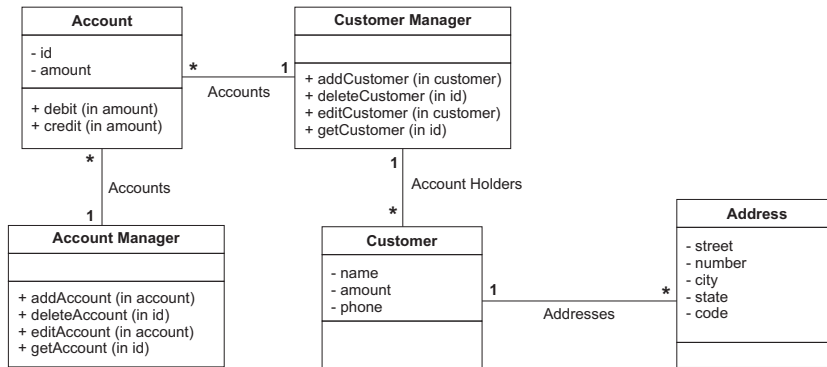
According to [Erl \(2005\)](#), entity services, which are the ones created to implement the life-cycle operations of the key business entities, are highly reusable because they are common to most parent business processes. The key business entities are described in the use cases usually using nouns, e.g., customer and account. However, other service candidates or their operations can be obtained from the verbs presented in the use cases, e.g., the users should **authenticate** themselves and the system should **restrict** access to specific pages, can be used to identify services to authenticate users and control user access respectively ([McGovern et al., 2003](#); [Larman, 2004](#)).

A conceptual service model can be defined from the use cases with the purpose of facilitating the identification of services. It consists of a model of the problem domain and it is created without regard for any application or technology. Figure 5.8 depicted a conceptual service model presented in [McGovern et al. \(2003\)](#). Each entity in the logical model is either a stateful entity or a stateless entity. In this sense, stateful entities, such as account, customer and address can be considered as components, while the managers are considered entity service candidates that manipulate these entities. The conceptual service model will also be useful during the definition of the service interfaces.

### Define Services from Quality Attributes

The SOPLE-DE considers a specific layer in its architectural style to deal with quality attributes. In this sense, service candidates can be identified using the quality attribute scenarios. The SOA architect is the role responsible to identify services in this task.

The SOA architect has to analyze the quality attribute scenarios in order to identify services that aid the accomplishment of architectural quality attributes. For instance,



**Figure 5.8** Conceptual Service Model

considering availability, well-known techniques can be used, such as ping/echo and heartbeat (Bass *et al.*, 2003). Thus, additional services can be identified to send messages to check the availability of other services. For security, services can be identified to authenticate users, control access to specific functionalities or limit access to services.

Table 5.9 presents some SOA design patterns that can be used to satisfy some quality attributes (Erl, 2009). In this sense, the quality attribute scenarios can be used as basis to select specific SOA patterns to achieve quality attributes.

It is important to note that some services identified may have already been identified in a previous service identification technique. In this way, at the end of the service identification task, the service candidates identified using all the available techniques should be consolidated, duplicated services should be removed and the complete list of services should be analyzed and reviewed by the business analyst with the purpose of producing an accurate list of services and service orchestrations.

After executing the complementary service identification techniques, an initial set of services and service orchestration candidates including drafts of their interface operations should be listed in the architecture document.

#### 5.4.1.2 Define Component Candidates

In this task, the components of the service-oriented product line architecture will be defined. The component identification task starts with an analysis of the feature model and service candidates identified previously with the purpose of identifying the architectural component candidates.

Each component identified in this activity is an architectural component candidate, which is an abstract, not implemented component that later in the approach will be considered for low-level design or discarded. The component identification task produces

Figure 5.9 SOA Design Patterns

	SOA Pattern	Problem	Solution
Reusability	Agnostic Capability	Service capabilities derived from specific concerns may not be useful to multiple service consumers, thereby reducing the reusability potential of the agnostic service.	Agnostic service logic is partitioned into a set of well-defined capabilities that address common concerns not specific to any one problem.
	Agnostic Context	Multi-purpose logic grouped together with single purpose logic results in programs with little or no reuse potential that introduce waste and redundancy into an enterprise.	Isolate logic that is not specific to one purpose into separate services with distinct agnostic contexts.
	Capability Recomposition	Using agnostic service logic to only solve a single problem is wasteful and does not leverage the logic's reuse potential.	Agnostic service capabilities can be designed to be repeatedly invoked in support of multiple compositions that solve multiple problems.
Performance	Asynchronous Queuing	When a service capability requires that consumers interact with it synchronously, it can inhibit performance and compromise reliability.	A service can exchange messages with its consumers via an intermediary buffer, allowing service and consumers to process messages independently by remaining temporally decoupled.
	Intermediate Routing	The larger and more complex a service composition is, the more difficult it is to anticipate and design for all possible runtime scenarios in advance, especially with asynchronous, messaging based communication.	Message paths can be dynamically determined through the use of intermediary routing logic.
	Redundant Implementation	A service that is being actively reused introduces a potential single point of failure that may jeopardize the reliability of all compositions in which it participates if an unexpected error condition occurs.	Reusable services can be deployed via redundant implementations or with failover support.
	Service Agent	Service compositions can become large and inefficient, especially when required to invoke granular capabilities across multiple services.	Event-driven logic can be deferred to event-driven programs that do not require explicit invocation, thereby reducing the size and performance strain of service compositions.
	Service Callback	When a service needs to respond to a consumer request through the issuance of multiple messages or when service message processing requires a large amount of time, it is often not possible to communicate synchronously.	A service can require that consumers communicate with it asynchronously and provide a callback address to which the service can send response messages.
Security	Brokered Authentication	Requiring the use of Direct Authentication can be impractical or even impossible when consumers and services do not trust each other or when consumers are required to access multiple services as part of the same runtime activity.	An authentication broker with a centralized identity store assumes the responsibility for authenticating the consumer and issuing a token that the consumer can use to access the service.
	Data Confidentiality	Within service compositions, data is often required to pass through one or more intermediaries. Point-to-point security protocols, such as those frequently used at the transport-layer, may allow messages containing sensitive information to be intercepted and viewed by such intermediaries.	The message contents are encrypted independently from the transport, ensuring that only intended recipients can access the protected data.
Flexibility	Concurrent Contracts	A service's contract may not be suitable for or applicable to all potential service consumers.	Multiple contracts can be created for a single service, each targeted at a specific type of consumer.
	Data Format Transformation	A service may be incompatible with resources it needs to access due to data format disparity. Furthermore, a service consumer that communicates using a data format different from a target service will be incompatible and therefore unable to invoke the service.	Intermediary data format transformation logic needs to be introduced in order to dynamically translate one data format into another.
	Data Model Transformation	Services may use incompatible schemas to represent the same data, hindering service interaction and composition.	A data transformation technology can be incorporated to convert data between disparate schema structures.
	Protocol Bridging	Services using different communication protocols or different versions of the same protocol cannot exchange data.	Bridging logic is introduced to enable communication between different communication protocols by dynamically converting one protocol to another at runtime.
	Event Driven Messaging	Events that occur within the functional boundary encapsulated by a service may be of relevance to service consumers, but without resorting to inefficient polling-based interaction, the consumer has no way of learning about these events.	The consumer establishes itself as a subscriber of the service. The service, in turn, automatically issues notifications of relevant events to this and any of its subscribers.
Modifiability	Compatible Change	Changing an already-published service contract can impact and invalidate existing consumer programs.	Some changes to the service contract can be backwards compatible, thereby avoiding negative consumer impacts.
	Service Refactoring	The logic or implementation technology of a service may become outdated or inadequate over time, but the service has become too entrenched to be replaced.	The service contract is preserved to maintain existing consumer dependencies, but the underlying service logic and/or implementation are refactored.

a list of architectural component candidates as output, which will be documented in the architecture document (SO-PLA). These components will provide the implementation for the operations exposed by the services.

The role responsible to perform the component identification task is the domain architect, who is responsible to analyze the feature model and define the features that will make part of each architectural component candidate considering the quality attributes that must be satisfied. The domain architect should solve the quality attribute trade-offs considering the priority of the attributes.

The components identified here will maintain the quality of the services in the architecture ([Arsanjani, 2004](#)). Thus, identify these components considering quality attributes, e.g., flexibility and evolvability, is necessary. However, some quality attributes of the SOA, e.g., security and performance, will be responsibility of the service platform selected as well ([Günther and Berger, 2008](#)).

For instance, suppose that each sub-feature of a service feature is implemented in a unique component. In this case, each sub-feature can evolve independently of each other, and they can also be modified without affect other features. However, the interfaces of the components implementing each sub-feature cannot change. Thus, this choice fits flexibility and evolvability, since features can be easily modified and the components encapsulating each feature can be combined in different ways to construct systems customized to specific customers.

Though, depending on the granularity of the feature model, implement one feature per component may cause the definition of many fine-grained components, which may cause performance problems and decrease reuse ([Erradi \*et al.\*, 2006](#)). In this case, variability techniques, e.g., aspect orientation or design patterns, can be used to group features into a component with variability. However, it is important to keep each feature in a specific class or aspect in order to still able to evolve and modify features independently. This issue will be discussed in the variability analysis activity in Section [5.4.2](#).

The components can be identified from use cases and quality attributes as well. However, it is important to note that components that will be used by several services, which may require interoperability issues should be considered as a service candidate. In this sense, components can implement use cases or quality attributes of a specific service. In order to make a specific component available for several services in the architecture, it needs to be exposed as a service. After concludes the component identification task, an initial set of component candidates including drafts of their interface operations should be listed in the architecture document.

### 5.4.1.3 Define Flows

In this task, the communication of services and service orchestrations will be defined. The flow identification task starts with an analysis of the services identified previously with the purpose of identifying the services communication flows.

The role responsible to perform the flow identification task is the SOA architect, who is responsible to analyze the services and define their communication protocols, e.g., SOAP or REST, and their communication types, e.g., synchronous or asynchronous, that will be used by the services to communicate with each other. The quality attributes should be considered in this task, since the protocol and type of communication impact some quality attributes, e.g., performance.

For instance, in the context of web services, SOAP-based design is more appropriated the RESTful web services in the following cases:

- A formal contract must be established to describe the interface that the web service offers, i.e., using WSDL documents, which describe the details such as messages, operations, bindings, and location of the web service;
- The architecture must address complex non-functional requirements, such as transactions, security, addressing, trust and coordination;
- The architecture needs to handle asynchronous processing and invocation.

On the other side, REST-based design can be used when services are completely stateless, a caching infrastructure can be leveraged for performance and service producers and service consumers have a mutual understanding of the context and content being passed along (Tyagi, 2006).

In this task, the integration mechanism that will be used in the SOA should be defined as well. For example, service consumers and providers can communicate directly without any broker such as the peer-to-peer communication pattern, or they can be mediated by a middleware, that in the context of SOA, it is known as the Enterprise Service Bus (ESB) (Josuttis, 2007; Bianco *et al.*, 2007).

It is important to note that in the context of service-oriented product lines, the communication protocol and type, and the integration mechanism can be treated as variation points. In other words, the same service can be accessed using several protocols in a synchronous or asynchronous way depending on the system of the product line (Segura *et al.*, 2007).



At this point, the information about service communications should be added to the list of architectural elements produced in the previous tasks. The next activity of SOPLE-DE, called variability analysis, explains how to refine the architectural elements identified in this activity considering the concepts of granularity and cohesion in order to select the best design choice to implement variability. Table 5.1 presents a checklist of the architectural elements identification activity.

**Table 5.1** Checklist of the Architectural Elements Identification Activity

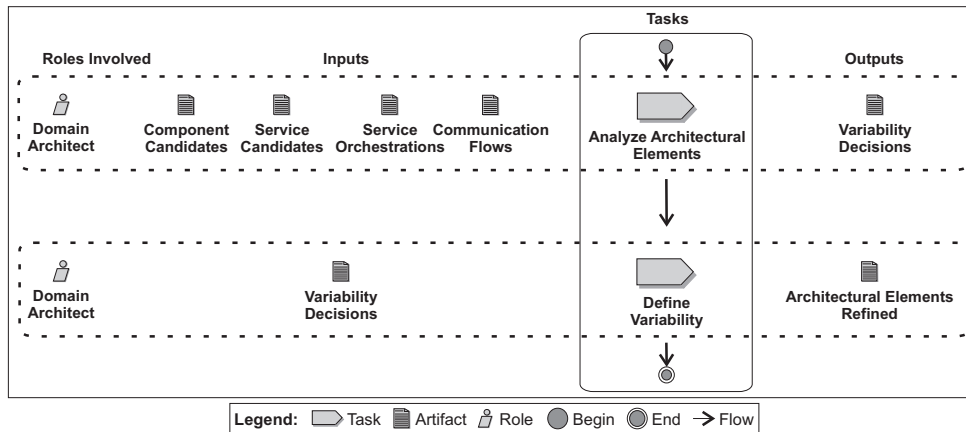
Checklist
Have you identified services from business processes, use cases (optional), feature model and quality attribute scenarios?
Have you considered the quality attribute scenarios to solve quality attributes trade-offs?
Have you defined the service and component interface operations?
Have you identified the service and component communication flows?
Have you defined if a service registry is going to be used?
Have you defined the service communication protocols and types?

### 5.4.2 Variability Analysis Activity

At this point, the components, services, service orchestrations and their communication flows were defined. During the variability analysis activity, it will be defined how the variability contained in the feature model, business processes, use cases and quality attributes will be implemented. This task is a refinement of the architectural elements identified previously.

The variability analysis activity receives as input the list of components, services, service orchestrations and their flows. Its output is a set of architectural decisions regarding variability, which defines where and how the variability will be modeled. These decisions are documented in the architecture document, and they basically consist of updating the list of architectural elements identified in the previous activity. After the variability analysis activity, the components and services are no longer candidates anymore.

The role responsible for this activity is the domain architect, and there are two tasks that should be performed in order to execute the variability analysis activity: *analyze architectural elements* and *define variability implementation technique*. Figure 5.10 shows the inputs, outputs, tasks and roles of the variability analysis activity, and its tasks are presented next.



**Figure 5.10** Variability Analysis Activity

#### 5.4.2.1 Analyze Architectural Elements

The analyze architectural elements task starts with an analysis of the component and service candidates identified previously. In this task, the cohesion and granularity of services and components should be analyzed with the purpose of reducing the number of candidates. The variability that cannot be isolated in specific components or services, i.e., crosscutting variability concerns, is also identified and analyzed here. This task is divided in three steps: *analyze cohesion*, *analyze granularity* and *analyze crosscutting concerns* as described next.

##### Analyze Cohesion

The objective of this step is to group together in a service or component the operations that are strongly related, e.g., operations related to a specific business entity. These operations may have been organized in different architectural elements during the previous activity.

In this sense, the operations contained in the component interfaces should be analyzed in order to identify related operations that can be joined in a unique component with the purpose of increasing cohesion and reducing the number of component candidates.

Thus, according to the analysis performed, related operations that were separated in different components, but have a high-level of cohesion, should be grouped in a unique component. Moreover, similar operations can also be merged into a single operation with variability. The same analysis and merging should be realized among the operations of the service candidates defined previously.

For instance, Figure 5.11 presents an example on the travel reservation product line. In this example, the operations of the services (Airline Reservation and Airline Cancellation) can be put in the same service, since these operations are related to each other and cohesion can be increased in this way. In addition, the operations to reserve airline of the Airline Reservation service can be grouped in a single operation with variability, i.e., the own service operation checks if go and return flights should be reserved, or only one way based on the itinerary information as depicted in the Airline Service.

Airline Reservation Service	Airline Cancellation Service	Airline Service
Void reserveAirline (Date, Time) Void reserveAirline (GoDate, GoTime, ReturnDate, ReturnTime)	Void cancelAirline (Id)	Void reserveAirline (Itinerary) Void cancelAirline (Id)

**Figure 5.11** Analyzing Cohesion

### Analyze Granularity

The granularity analysis task consists of analyzing the granularity of the variation points and variants of the service-oriented product line with the purpose of identifying fine-grained variability, which are variation points that can be implemented by changing a class attribute or method.

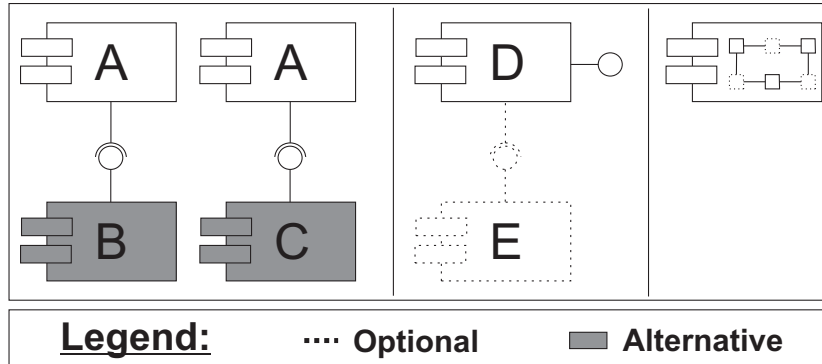
In this sense, after the granularity analysis, components that were identified to implement each variant of a variation point can be grouped into a single component with variability. Components implementing variants that require different classes, i.e., coarse-grained variability, should be left isolated. The same strategy should be performed with the services that were identified to implement fine-grained variability.

In order to implement variability with fine-grained granularity, well-known variability implementation techniques can be used, such as aspect-oriented programming, conditional compilation, configuration files and design patterns ([Gacek and Anastasopoulos, 2001](#)).

In the cases where the variability is more coarse-grained, Component-Based Development (CBD) can be used to implement the variability, i.e., each variant is implemented in a specific component. In this way, the components are not grouped together, they are maintained separated ([Kästner et al., 2008](#)).

Figure 5.12 depicts how component variability can be implemented. In the first three cases, CBD is used as the variability implementation mechanism. As it can be seen, components *A* and *D* implement variation points, components *B* and *C* implement alternative variants, and component *E* implements an optional sub-feature. However, in

the last case, CBD may be not appropriated because the variability granularity is low. Hence, other variability mechanisms, e.g., aspects, design patterns or configuration files, can be used to implement variability internally.



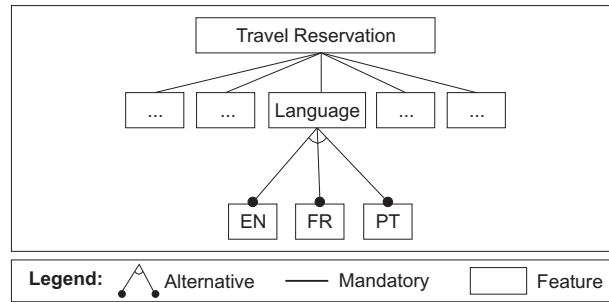
**Figure 5.12** Component Variability

In the context of service variability, service orientation can be used as a technique to implement variability, i.e., each variant can be implemented in a specific service. It is the way the current service-oriented applications are customized, in other words, changing the service order or even the participants of service compositions.

However, depending on the variability granularity, it may be insufficient. A variation point can be implemented changing a class attribute, or a class, a method or even an entire component or service. Thus, in some cases where the variability granularity is low, it is also necessary to introduce variability into services internally.

In order to implement service variability, i.e., a unique service that can be customized to different purposes, aspect-oriented programming, configuration files, parameterization and design patterns can be used with the purpose of changing service interfaces or modifying the service behavior according to the requirements of a specific system of the product line.

For instance, Figure 5.13 presents part of the feature model from the travel reservation product line. The language feature contains three alternatives variants: English, French and Portuguese. However, the difference among these variants is a file with the strings in the appropriated language. Thus, it does not make sense to use CBD as the implementation technique because the variability is low. It is implemented changing just a class attribute that stores the file name and location. The architect sometimes will not visualize the real differences among variants during design, for this reason, not only the activities of SOPLE-Design, but the activities of the SOPLE as whole must be executed interactively.

**Figure 5.13** Variability Granularity

### Analyze Crosscutting Concerns

In this step, crosscutting variability concerns should be analyzed. In some cases, a logging component for instance, has its code spread in several components. Thus, the domain architect should analyze these cases in order to decide how this variability is going to be implemented.

It is important to note that during product derivation, components and services will be selected or removed from the architecture. Hence, crosscutting variability concerns cannot affect the inclusion and exclusion of architectural elements of the architecture. The strategy selected to implement this type of variability should support this flexibility.

This step is essential because some features and business process activities cannot be isolated in a specific component or service. Thus, part of these features and activities (few lines of code) must be spread in several places (more than one component or service). The domain architect should identify these cases early in order to decide how it is going to be addressed.

#### 5.4.2.2 Define Variability Implementation Technique

In this step, the variability implementation technique will be defined for all types of variability presented in the product line. In the context of service-oriented product lines, there are some variation points that are specific to services. SOPLE-DE considers the following service-specific variation points:

- **Type of Communication:** Services can communicate in a synchronous or asynchronous way. In this sense, services can behave differently in specific systems;

- **Communication Protocol:** Services can use different protocols to communicate, such as SOAP and REST;
- **Integration Mechanism:** The way service interacts can be variable, e.g., the services can communicate directly with each other (peer-to-peer), or they can use an integration channel, such as an Enterprise Service Bus (ESB);
- **Service Discoverability:** Service consumers can use a service registry to find the service providers they need. This strategy supports dynamic compositions of services, i.e., during execution. On the other hand, consumers may call service providers that they already know at compile-time;
- **Monitoring:** The metrics used to monitor services can be specific for each service-oriented system of the product line, e.g., some systems may monitor the number of service calls, while others may monitor only the response time of services.

Besides variation points specific for service orientation, SOPLE-DE also considers variability that are not directly related to service orientation, such as the ones described next:

- **Functional variability:** The functionalities of services and components may vary depending on the system being considered;
- **Quality Attributes:** The architecture of different systems of the product line can be customized to satisfy different quality attributes, or different levels of quality attributes.

It is important to note that the binding time and the variability type of a variation point should be considered during this step because different variability mechanisms support specific types of variability and binding times ([Gacek and Anastasopoulos, 2001](#)). For instance, conditional compilation cannot be used to implement dynamic (runtime) binding times, since using this technique the selection of variants is realized at compile-time. Table [5.2](#) presents a checklist of the variability analysis activity.

### 5.4.3 Architecture Specification Activity

In the architecture specification activity, the high-level design of components, services, service orchestrations and their flows will be specified. In this activity, architectural views are produced, and they may contain variability as the artifacts of the core assets

---

**Table 5.2** Checklist of the Variability Analysis Activity

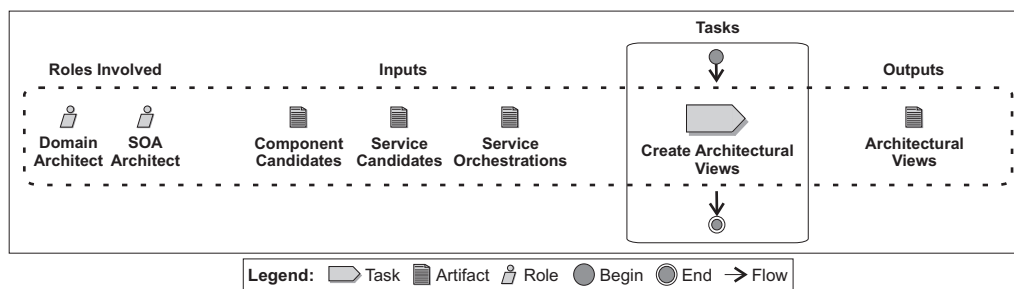
Checklist
Have you analyzed the cohesion of the component and service operations?
Have you analyzed the granularity of the variation points?
Have you grouped services and componets with low granularity?
Have you identified cross-cutting variability concerns?
Have you defined the variability implementation mechanisms for the variation points based on their granularity?

development cycle. Thus, architecture specification requires notations with support for variability representation, such as (Gomaa, 2004) and (Razavian and Khosravi, 2008).

The architecture specification activity receives the list of architectural elements (components, services and service orchestrations) and their flows identified in the previous activities as inputs, and produces the architectural views that will be documented in the architecture document. The domain and SOA architects should perform this activity.

Service-oriented product line architectures, as any other software architecture, are complex entities that cannot be represented in a simple one-dimensional fashion. Since there are different stakeholders involved in a product line project with particular concerns about the systems, it is important to use multiple views to represent the service-oriented product line architecture (Bass *et al.*, 2003). Moreover, the use of multiple architectural views are essential in order to handle separately the functional and non-functional requirements of the service-oriented product line (Kruchten, 1995).

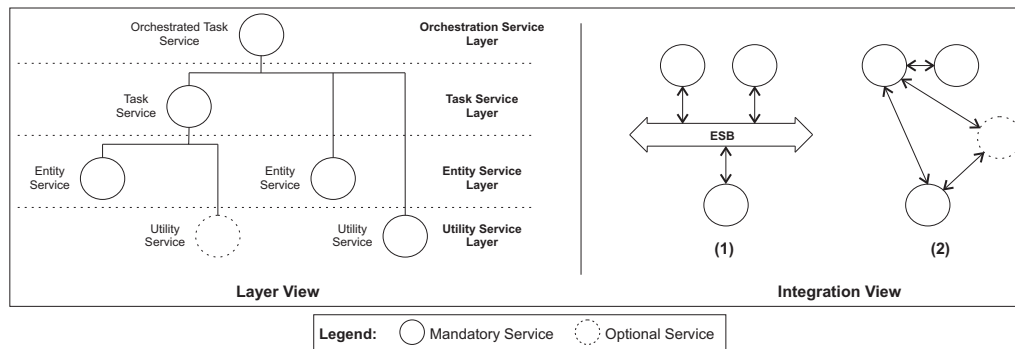
Figure 5.14 shows the inputs, outputs, tasks and roles of the architecture specification activity. The architectural views that can be produced to represent the concerns of the stakeholders involved in the project, and the quality attributes of the SO-PLA are presented next (Kruchten, 1995; Ibrahim and Misie, 2006).

**Figure 5.14** Architecture Specification Activity

**Layer View.** The objective of this viewpoint is to represent the layers of the SOA solution. Thus, in this perspective, the architectural elements identified in the previous

activities are represented in their respective layer. Figure 5.15 depicts a layer view example with some services categorized. In this figure, the service layer of the SOPLE-DE are further divided into task, entity and utility layers as used in Erl (2005) and Erl (2007). The lines indicate that the service of the upper layer uses the services of the layers below, and dashed circles represent optional services.

**Integration View.** The purpose of this view is to depict the integration mechanism that will be used in the SOA. Figure 5.15 shows two possible integration patterns that can be selected for a service-oriented architecture (Bianco *et al.*, 2007): (1) hub-and-spoke and (2) peer-to-peer. In the hub-and-spoke pattern, the interaction among service consumers and providers is mediated by a middleware. In the context of SOA, this middleware is known as Enterprise Service Bus (ESB) (Josuttis, 2007). In the second integration pattern, services communicate directly with other services without any broker.



**Figure 5.15** Layer and Integration Views

In most cases, the performance of the peer-to-peer integration pattern is better. However, other quality attributes, e.g., modifiability and flexibility, cannot be satisfied since the services are glued at compile-time, i.e., service calls are realized directly in the source code. On the other hand, the use of a service bus allows security checks, dynamic binding, message transformations and other benefits. In this way, the hub-and-spoke integration pattern is better to achieve several quality attributes, e.g., security, availability, flexibility and modifiability (Josuttis, 2007).

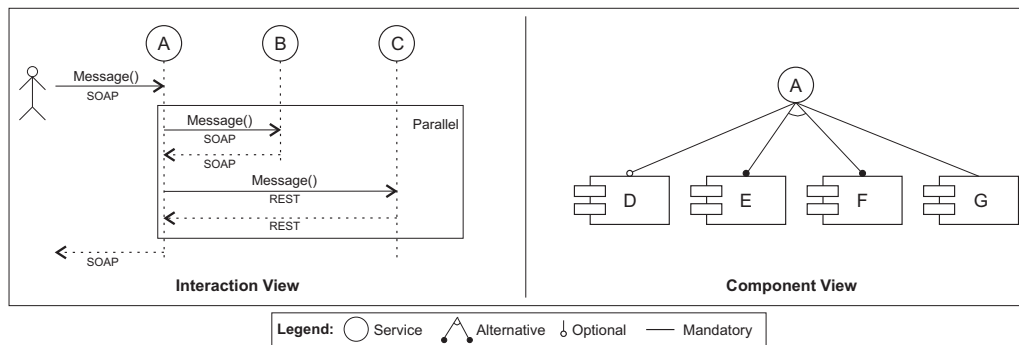
**Interaction View.** The main goal of this viewpoint is to show the communication protocols and the messages exchanged among service consumers and providers. It is also used to represent concurrency issues and depicts the dynamic behavior of service orchestrations.

In SOA systems, each interaction between a service consumer and a service provider can be implemented using different protocols, e.g., SOAP or REST. In the context of



service-oriented product lines, the protocol of communication can be a variation point. The protocol used impacts quality attributes of the product line, such as interoperability, performance and modifiability (Bianco *et al.*, 2007). Thus, quality attribute variability can be implemented changing the protocol of communication. Figure 5.16 shows an example of interaction view.

**Component View.** The purpose of this perspective is to represent the logical structure of the components that will be used by the services. From the perspective of service-oriented design, it is a best practice to create high-level and coarse-grained service interfaces that implement a complete business process or set of business activities. Thus, services should expose the functionality of several components (McGovern *et al.*, 2003). Services with variability that will be implemented using CBD should be represented here, i.e., components should be marked as optional and alternative. Figure 5.16 shows an example of component view.



**Figure 5.16** Interaction and Component Views

The domain and SOA architects have to decide which architectural views will be used to represent the service-oriented product line architecture depending on the stakeholders involved in the project, and the size and domain of the product line being constructed. The architects also decide the level of details of each view. Table 5.3 presents a checklist of the architecture specification activity.

**Table 5.3** Checklist of the Architecture Specification Activity

Checklist
Have you selected the architectural views to represent the architectural elements identified?
Have you defined the level of details of the architectural views selected?
Have you documented the architectural views?

### 5.4.4 Architectural Elements Specification Activity

In this activity, the low-level design and the detailed description of components and services will be defined and documented. The purpose of this activity is to design and document the internal behavior, pre-conditions, invariants, post-conditions and contracts (interfaces) of the services and components of the architecture.

The architectural elements specification activity receives the list of architectural elements identified in the previous activities, their flows and the architectural views produced as inputs. It produces some artifacts, e.g., UML diagrams and service descriptions, that should be documented in the architecture document.

There are two tasks that should be performed in order to realize the architectural elements specification activity: *specify services* and *specify components*. Figure 5.17 shows the inputs, outputs, tasks and roles of the architectural elements specification activity, and its tasks are presented next.

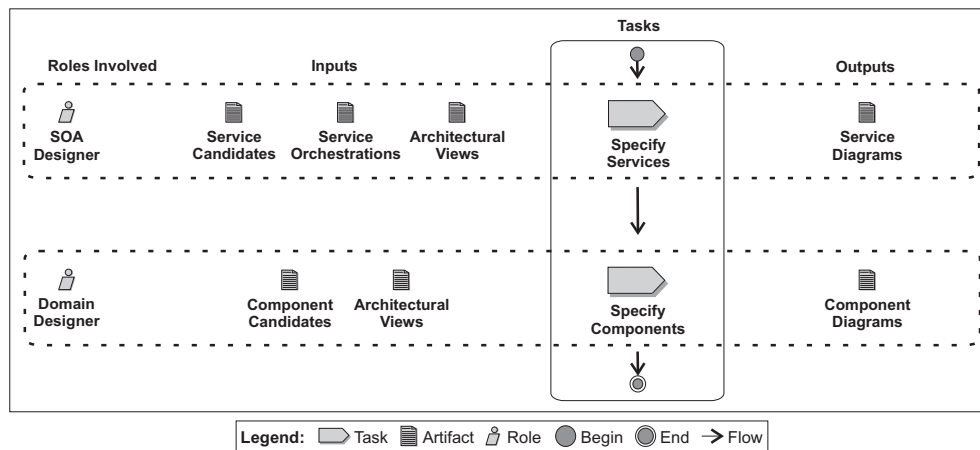


Figure 5.17 Architectural Elements Specification Activity

#### 5.4.4.1 Specify Services

In this task, the low-level design, the functionality provided by the service, quality attributes and the interfaces of the services will be defined and documented. The purpose of this task is to define the internal classes and create diagrams to represent the services internal behavior, and document detailed information about the services.

The service designer is the role responsible to perform this task. The following steps should be executed: *document service interfaces*, *create service descriptions* and *design service internal diagrams*. These steps are described next.

### Document Service Interfaces

In this step, the interfaces of the services of the service-oriented product line architecture are documented. SOPLE-DE advocates the interfaces to be described using Interface Description Language (IDL) ([Erl, 2007](#)). In this way, the interfaces are documented using a non-specific programming language notation. Stereotypes, e.g., optional and alternative, should be used to represent service interface variability as presented in [Figure 5.18](#).

Reservation Service
Void reserveAirline ( [in] Itinerary itinerary )
Void reserveVehicle ( [in] Itinerary itinerary ) << optional >>

**Figure 5.18** Service Interface

### Create Service Descriptions

In this step, the service description should be produced and documented in the architecture document. It should contain the following fields:

- **Description:** Describes the general purpose of the service, i.e., functional requirements;
- **Pre-conditions:** Lists the conditions that must be satisfied before using the service functionalities;
- **Invariants:** Describes the conditions that must be satisfied during the whole execution of the service operations, otherwise, the functionality being executed should stop as soon as the condition fails;
- **Post-conditions:** Lists the conditions that must be satisfied after the execution of the service operations;
- **Quality Attributes:** Describes the non-functional requirements (e.g., service level agreements) that are satisfied by the service, e.g., considering a performance attribute, the response time can be defined between 0.75 and 1.5 seconds.

### **Design Service Internal Diagrams**

In this step, the service designer should define the services internal diagrams. During this definition, the variability implementation mechanism defined in the variability analysis activity, and the components and services used by the service being designed must be considered. In this way, UML diagrams, e.g., class, state, sequence and activity diagrams, are created to represent the internal behavior of each service, when necessary.

#### **5.4.4.2 Specify Components**

In this task, the component interfaces will be documented, and the low-level design of components will be performed. The purpose of this task is to define the internal classes and create diagrams in order to represent the internal behavior of the components. The domain designer is the role responsible to perform this task, and the following steps should be executed: *document component interfaces* and *create internal diagrams*. These steps are presented next.

#### **Document Component Interfaces**

In this step, the interfaces of the components in the service-oriented product line architecture are documented. IDL should be used for this documentation. Components interface variability should be documented using stereotypes. For the components with dependency with other components, the required interfaces must be represented as well.

#### **Create Internal Diagrams**

In this step, the domain designer should define the components internal classes. In this definition, the component variability and the variability mechanism defined in the variability analysis activity must be considered. In this sense, UML class diagrams should be created to represent the components internal classes. The class diagrams will represent the internal static structure and variability of the components. Additional UML diagrams, e.g., state and sequence diagrams, may be produced if necessary. Table 5.4 presents a checklist of the architectural elements specification activity.

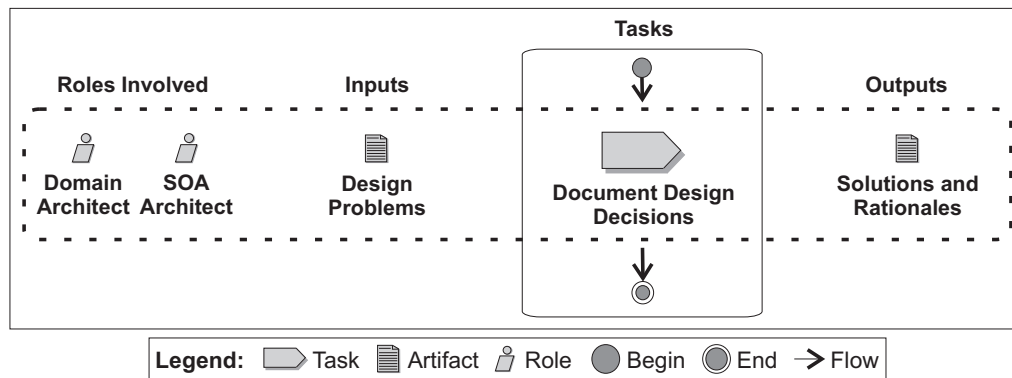
### **5.4.5 Design Decisions Documentation Activity**

Design decisions are very important parts of the design discipline (Jarczyk *et al.*, 1992). As it can be seen in Figure 5.2, these decisions can be made during the whole SOPLE-DE,

**Table 5.4** Checklist of the Architectural Elements Specification Activity

Checklist
Have you documented the service and component operations?
Have you defined the service contracts with the service level agreements, pre-conditions, post-conditions, etc.?
Have you modeled the UML diagrams necessary to represent the internal behavior of the services and components?
Have you put all this information in the architecture document?

since design decisions are made for problems<sup>2</sup> that the domain and SOA architects face during the project. Such decisions can be the selection of technologies that are going to be used, the variability technique that will be used to implement variability, and so on.

**Figure 5.19** Design Decisions Documentation Activity

This activity receives as input the design problems that appear during the SOPLE-DE. After identify these problems, the domain and SOA architects should find solutions for them. This activity produces as output a set of solutions for specific problems with the rationale used to perform each solution. Figure 5.19 shows the inputs, outputs, tasks and roles of the design decisions documentation activity.

The only step in order to perform this task is the identification of issues to be addressed during the design. This identification can happen during the whole design life-cycle. When an issue is identified, the domain and SOA architects should identify the possible solutions for the issue. Once the possible solutions are identified, the domain and SOA architects should decide which solution is going to be performed. For each decision, a rationale must be documented in order to allow other architects and developers to identify the reasons for the decisions (Jarczyk *et al.*, 1992). The solution and rationale should be

<sup>2</sup>In this context, problems mean specific situations that need a special attention during the design and must be discussed and documented.

documented in the architecture document. Table 5.5 presents a checklist of the design decisions documentation activity.

**Table 5.5** Checklist of the Design Decisions Documentation Activity

Checklist
Have you identified the key design problem?
Have you analyzed solutions for each key design problem?
Have you documented the solution taken and the rationale for each design problem?

## 5.5 Chapter Summary

This chapter presented an approach to design service-oriented product line architectures (SOPLE-DE) as well as its principles, roles, phases, activities, tasks, inputs and outputs (Medeiros *et al.*, 2009). The approach is divided in five phases: *architectural elements identification*, *variability analysis*, *architecture specification*, *architectural elements specification* and *design decisions documentation*.

During the architectural elements identification activity, it was proposed methods to identify services from feature models, business process models, quality attribute scenarios and use cases. A method for component identification using feature models was also proposed. The variability analysis activity presented guidelines on how to use the concepts of granularity and cohesion in order to introduce variability into components and services. Architecture specification activity described how the service-oriented product line architecture should be documented using a set of architectural views. The architectural elements specification activity presented how the low-level design of services and components should be realized and documented. And finally, the design decisions documentation discussed how important design decisions are documented.

In the next chapter, it will be presented a preliminary experimental study with the SOPLE-DE performed with the purpose of validating and refining it.

# 6

## A Preliminary Experiment

### 6.1 Introduction

Software has become part of several products nowadays, and it can be found in different kinds of products, such as toasters, televisions, automobiles and space shuttles. This means that much software has been developed and is being developed ([Wohlin \*et al.\*, 2000](#)). However, software development is a complicated and labor intensive task that can run into several problems, e.g., missing functionalities, cost overruns, poor quality and unacceptable performance ([Kruchten, 2003](#)). Hence, organizations focus on process improvements in the software development area with the intention of reducing costs, time and risks, and increasing software quality ([Chidamber and Kemerer, 1994](#)).

In the context of software engineering, empirical studies such as surveys, case studies and experiments, play an important role since the progress in any discipline depends on the ability of people to understand the basic units necessary to solve problems ([Basili, 1996](#)). In particular, experiments provide a systematic, disciplined, quantifiable, and controlled way to evaluate and test new theories and hypotheses ([Basili \*et al.\*, 1986](#)).

One of the main advantages of experiments is the control of, for example, subjects, objects and instruments. It ensures that we are able to draw more general conclusions than, for example, case studies ([Kitchenham \*et al.\*, 1995](#)). Other advantages include the ability to perform statistical analysis using hypothesis testing methods and opportunity for replication ([Wohlin \*et al.\*, 2000](#)). However, in order to impose full control, experiments are often small, which may be a problem when it scales from the laboratory to a real project ([Kitchenham \*et al.\*, 1995](#)).

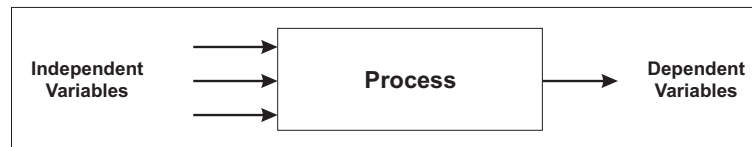
In this sense, this chapter presents a preliminary experimental study performed with the purpose of evaluating the efficacy, understanding and applicability of the SOPLE-DE in the context of service-oriented product line projects. In this experiment, the process

of Wohlin *et al.* (2000) was used to define, plan and execute the experimental study. In order to consider real SOA problems, the *Travel Reservation* domain, which is commonly used in SOA examples, contains enterprise integration requirements and other real SOA difficulties, was the project used in the experimental study (Snell, 2002; Segura *et al.*, 2007).

The remainder of this chapter is organized as follows: Section 6.2 presents essential information to understand experimental studies; Section 6.3 presents the definition, planning, operation, analysis and interpretation of the experimental study with the SOPLE-DE, and Section 6.4 presents the conclusions and the lessons learned with the experimental study; Finally, Section 6.5 concludes this chapter with its summary.

## 6.2 Background Information

The goal of an experiment is to study the outcome when we vary some of the input variables to a process (Wohlin *et al.*, 2000). There are two types of variables in experiments, dependent and independent variables, as illustrated in Figure 6.1.



**Figure 6.1** Experiment Variables

An experiment studies changing one or more **independent variables**, keeping other independent variables controlled at a fixed level, and analyzing the impact on the **dependent variables** (Wohlin *et al.*, 2000). For instance, it is necessary to study the effect of a new development method on the productivity of the personnel. The dependent variable in the experiment is the productivity. Independent variables are, for example, the development method, the experience of the personnel, tool support, and the environment.

As mentioned, an experiment studies the effect of changing independent variables, which are also called **factors**, and one particular value of a factor is called **treatment** (Wohlin *et al.*, 2000). For instance, in the example of changing the development method, two treatments are used for the factor: the old development method, and the new one.

The treatments are being applied to a combination of **objects** and **subjects**. An object can, for example, be a document that will be reviewed with different inspection



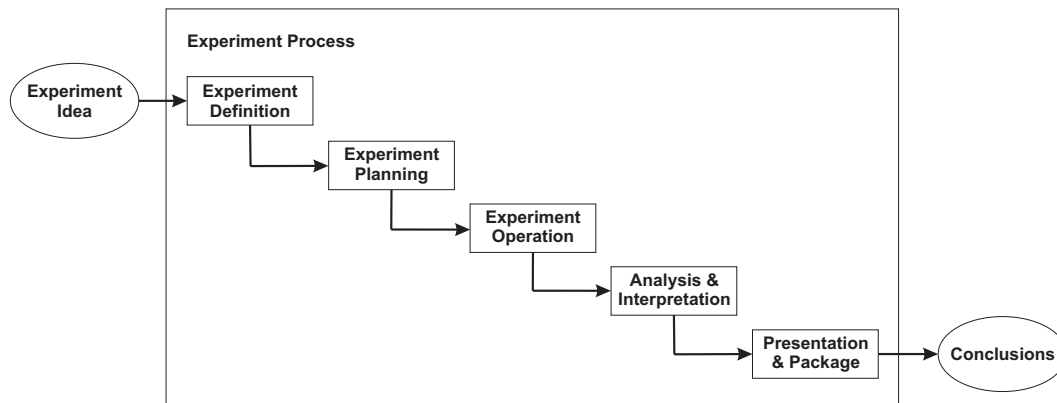
techniques. The people that apply the treatment are called subjects, but they can also be called participants. In the example of changing the development methods, the objects can be the programs to be developed and the subjects are the personnel.

At the end, an experiment consists of a set of tests where each **test** is a combination of treatment, subject and object. For instance, a test can be that person N (subject) uses the new development method (treatment) for developing program A (object).

## 6.3 The Experimental Study

The experimental study of the SOPLE-DE was performed following the process of [Wohlin et al. \(2000\)](#), which divides the experiment process in the following main activities, as depicted in Figure 6.2. An experiment process is necessary to make sure that the proper actions are taken to ensure a successful experiment, and it will provide support in setting up and conducting the experimental study.

The first step is the *definition*, in which the experiment is defined in terms of the problem, objective and goals. The *planning* activity comes next, where the design of the experiment is determined, the instrumentation is taken into account, and the threats of the experiment are evaluated. The *operation* activity is next, and it follows the design of the experiment determined previously. In this activity, the measurements are collected, and then analyzed during the *analysis and interpretation* activity. Finally, the results are published in the *presentation and package* activity.



**Figure 6.2** Activities of the Experiment Process

The next sections present the definition, planning, operation, analysis and interpretation of the experiment with the SOPLE-DE. The experiment presentation and package is represented with this chapter.

### 6.3.1 Definition

In this section, the foundation of the experiment is determined. If it is not properly defined, rework may be required, or even worse, the experiment will not be appropriated to study what was intended (Wohlin *et al.*, 2000). The purpose of this phase is to define the goals of the experiment according to a defining framework. In this experimental study, the Goal Question Metric (GQM) will be used for definition (Basili *et al.*, 1994).

The GQM is based on the assumption that an organization interested in measurements must first specify the goals for itself and its projects, trace those goals to the data that are intended to define those goals operationally, and finally provide a framework to interpreting the data with respect to the stated goals. The result of the application of GQM is the specification of a measurement system focusing on a set of particular issues, and a set of rules for interpreting the measured data. The resulting measurement model has three levels (Basili *et al.*, 1994):

- **Goal:** A goal is defined for an object, for a variety of reasons, with respect to various models of quality, from various points of view, relative to a particular environment;
- **Question:** A set of questions is used to characterize the way the assessment of a specific goal is going to be performed based on some characterizing model;
- **Metric:** A set of data is associated with every question in order to answer it in a quantitative way.

The next subsections present the goal, questions and metrics that were used in this experimental study.

#### Goal

**G1.** The goal of this experiment is *to analyze the SOPLE-DE for the purpose of evaluation with respect to its efficacy, understanding and applicability from the point of view of researcher in the context of service-oriented product line projects.*

#### Questions

- Q1.** Does the SOPLE-DE aid architects to generate services with low coupling?
- Q2.** Does the SOPLE-DE aid architects to generate services with low instability?

**Q3.** Does the SOPLE-DE aid architects to generate cohesive services?

**Q4.** Do the subjects have difficulties to understand the SOPLE-DE?

**Q5.** Do the subjects have difficulties to apply the SOPLE-DE in practice?

### Metrics

**M1. Service Coupling (SC):** Coupling is a measure of the extent to which interdependencies exist between software modules (Pereplechikov *et al.*, 2007). In this sense, two services are coupled if at least one of them acts upon the other.

According to Papazoglou and Heuvel (2006), one way of measuring the service design quality is coupling. Hence, the objective is to minimize coupling, which means to make self-contained services as independent as possible of other services. Low coupling between services indicates a well-partitioned system and avoids problems of service redundancy and duplication.

In addition, low coupling among service consumers and providers reduces service calls, which increases the performance of systems since service remote calls have a considerable time overhead (Erradi *et al.*, 2006). Moreover, the development of systems with unacceptable performance is also one of the most common causes for failures in software projects (Jones, 1995). Hence, this quality attribute needs a special attention during the design. In this sense, the following metric will be evaluated (Hofmeister and Wirtz, 2008):

$$SC(s) = \text{number of service providers used by a service consumer (s),}$$

where (s) is a service of a given system.

This coupling metric has range  $[0, n]$ , where  $n$  is the number of service providers different from (s) of a given system.  $SC = 0$  indicates a totally loosely coupled service, and  $SC = n$  indicates a maximally coupled service (Hofmeister and Wirtz, 2008).

**M2. Service Instability (SI):** The reason for a design rigid, fragile and difficult to reuse is the interdependency among its modules (Martin, 1994). A design is rigid if it cannot be changed easily, and a single change in a specific service causes a cascade of changes in several independent modules. In this sense, a change cannot be estimated because its impact on different modules of the design is not predictable by the architects and designers.

Thus, this metric will measure the instability of the services in order to assess it. The following metric will be evaluated (Quynh and Thang, 2009):

$SI(s) = P/(P + C)$ , where  $C$  is the number of service consumers that call service ( $s$ ), and  $P$  is the number of service providers that service ( $s$ ) uses.

The service instability metric has range  $[0, 1]$ , where  $SI = 0$  indicates a maximally stable service and  $SI = 1$  indicates a totally unstable service (Quynh and Thang, 2009).

**M3. Lack of Service Cohesion (LSC):** Cohesion is the degree of the strength of functional relatedness of operations within a service (Papazoglou and Heuvel, 2006). Highly cohesive service operations indicate good functionalities subdivision, and imply high reusability. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process (Rosenberg and Hyatt, 1998). Service operations with low cohesion could probably be subdivided into two or more services with increased cohesion.

In addition, high cohesion increases the clarity and comprehension of the design, simplifies maintenance and future enhancements, achieves service granularity at a fairly reasonable level, and often supports low coupling. Moreover, highly related functionality supports increased reuse potential, as highly cohesive service modules can be used for very specific purposes (Papazoglou and Heuvel, 2006).

This metric is based on the Lack of Cohesion of Methods (LOCM) for object orientation (Henderson-Sellers, 1995). The intuition underlying the LOCM metric is that cohesive methods in a class should access the same class attributes. We adapt this metric for service orientation, considering that cohesive service operations should access the same data abstractions, i.e., the same business entities (Pereplechikov *et al.*, 2007). In this sense, the following metric will be evaluated:

$LSC(s) = \text{Number of business entities accessed by the operations of service } (s).$

This lack of service cohesion metric has range  $[1, n]$ , where  $n$  is the number of business entities of a specific domain, and  $LSC = 1$  indicates totally cohesion among service operations, and  $LSC = n$  indicates maximally low cohesion. This metric assumes that the service operations of a specific service should access at least one business entity of the domain.

**M4. Misunderstanding Problems (MP):** This issue will be used to identify possible misunderstanding problems during the reading of the SOPLE-DE documentation. It is necessary to identify and analyze the difficulties found by the subjects learning the approach. It is important to note that the misunderstanding problems found will be mapped to the respective activity of the approach according to the information provided by the subjects. This mapping will be used to detect specific problems of the SOPLE-DE with the purpose of refining its documentation. This information will be provided by the subjects using a questionnaire. In this sense, the following metric will be evaluated:

$MP = \% \text{ of subjects that had difficulties to understand the SOPLE-DE.}$

**M5. Applicability Problems (AP):** This issue will be used to identify possible applicability problems during the execution of the SOPLE-DE. It is necessary to identify and analyze the difficulties found by the subjects applying the approach in practice. It is important to note that the applicability problems found will be mapped to the respective activity of the approach according to the information provided by the subjects. This mapping information will be used to detect specific problems with respect to the applicability of the SOPLE-DE in practice with the purpose of refining its activities. This information will be provided by the subjects using a questionnaire. In this sense, the following metric will be evaluated:

$AP = \% \text{ of subjects that had difficulties to apply the SOPLE-DE in practice.}$

### 6.3.2 Planning

The experiment definition determines the foundations for the experiment, i.e., why the experimental study will be conducted, while the experiment planning prepares for how the study will be conducted ([Wohlin et al., 2000](#)). As any other type of engineering activity, the experiment must be planned and the plans must be followed in order to control the experiment. The results of the experiment can be disturbed, or even destroyed if not planned properly.

#### Context

The objective of this experiment is to evaluate the efficacy, understanding and applicability of the SOPLE-DE in the context of service-oriented product line projects. The experiment will be conducted in a university laboratory with postgraduate students using a project on the *Travel Reservation* domain.

The experimental study will be conducted as a *Replicated Project*, which is characterized as being a study which examines object(s) across a set of teams, and a single project (Basili *et al.*, 1986).

The subjects of the study will be requested to act as the roles defined in the SOPLE-DE, i.e., domain architect and SOA architect. However, a subject can play more than one role during different activities and tasks of the SOPLE-DE. All the subjects will be trained to use the approach as discussed next.

#### **Training**

The subjects will be trained to use the approach at the university. The training will be divided in two steps: in the first one, concepts related to software reuse, variability, component-based development, domain engineering, software product lines, asset repository, software reuse metrics, and software reuse processes will be explained during ten lectures with two hours each at a postgraduate course at the Federal University of Pernambuco.

In the second step, independently of the course at the university, the experimenter will present the concepts and principles of Service-Oriented Architecture (SOA), such as service-oriented principles, Enterprise Service Bus (ESB), service registry, and SOA roles, during one class (two hours), and next, the SOPLE-DE will be discussed during two more lectures of one hour each. During the training, the subjects can interrupt to ask issues related to the lectures.

#### **Pilot Project**

Before performing the study, a pilot project will be conducted with the same structure defined in this planning. The pilot project will be performed by two subjects, who will be trained and will not participate of the real experiment. In the pilot project, the subjects will use the same material described in this planning, and will be observed by the responsible researcher. In this way, the pilot project will be a study based on observation, aiming to detect problems and improve the planned material before its use.

#### **Hypotheses**

In the context of experimental studies, there are two types of hypotheses: null and alternative hypotheses. The *null hypotheses* are the ones that the experimenter wants to

reject with as high as significance as possible, while the the *alternative hypotheses* are the ones in favor of which the null hypotheses are rejected (Wohlin *et al.*, 2000).

The following sub-sections present the null and alternative hypotheses of this experiment. The data collected during the course of the experiment will be used to, if possible, reject the null hypotheses.

### The Null Hypotheses

In this experimental study, the null hypotheses determine that the use of the SOPLE-DE in service-oriented product line projects does not produce benefits that justify its use and that the subjects will have difficulties to understand and apply the approach in practice. Thus, according to the selected criteria, the following null hypotheses were defined:

- H1.**  $\mu_{SC}$  of services without SOPLE-DE  $< \mu_{SC}$  of services with SOPLE-DE
- H2.**  $\mu_{SI}$  of services without SOPLE-DE  $< \mu_{SI}$  of services with SOPLE-DE
- H3.**  $\mu_{LSC}$  of service operations without SOPLE-DE  $< \mu_{LSC}$  of service operations with SOPLE-DE
- H4.**  $\mu_{More\ than\ 50\%}$  of the subjects will have difficulties to understand the SOPLE-DE
- H5.**  $\mu_{More\ than\ 50\%}$  of the subjects will have difficulties to apply the SOPLE-DE in practice

### The Alternative Hypotheses

In this experimental study, the alternative hypotheses determine that the use of the SOPLE-DE in service-oriented product line projects produces benefits that justify its use and that most of the subjects will not have difficulties to understand and apply the approach in practice. Thus, the following alternative hypotheses were defined:

- H1.**  $\mu_{SC}$  of services without SOPLE-DE  $\geq \mu_{SC}$  of services with SOPLE-DE
- H2.**  $\mu_{SI}$  of services without SOPLE-DE  $\geq \mu_{SI}$  of services with SOPLE-DE
- H3.**  $\mu_{LSC}$  of service operations without SOPLE-DE  $\geq \mu_{LSC}$  of service operations with SOPLE-DE
- H4.**  $\mu_{More\ than,\ or\ 50\%}$  of the subjects will not have difficulties to understand the SOPLE-DE
- H5.**  $\mu_{More\ than,\ or\ 50\%}$  of the subjects will not have difficulties to apply the SOPLE-DE in practice

### Variables Selection

In the variables selection, the independent and dependent variables of the experiment are selected. All the variables that are manipulated and controlled are called *independent variables*. These variables should have some effect on the dependent variables. In this study, the experience of the subjects and the use of the SOPLE-DE are the independent

---

variables considered. The experience of the subjects will be used to group subjects into blocks with similar profiles during the analysis of the results.

According to (Wohlin *et al.*, 2000), the choices of independent variables also include choosing the measurement scales, the range for the variables and the specific levels at which tests will be made. In this experiment, the experience of the subjects will be considered in two levels:

1. **Subjects with significant experience:** Subjects that have participated in at least three industrial and three academic software projects;
2. **Subjects without significant experience:** Subjects that have not participated in at least three industrial and three academic software projects.

The *dependent variables* are the ones that we want to study to see the effect of the changes in the independent variables. In this experimental study, the dependent variables considered are the quality of the service-oriented product line architecture produced, the understandability and the applicability of the SOPLE-DE.

In this context, the experience of the subjects and the use of SOPLE-DE will be manipulated with the purpose of measuring the effects on the quality of the architecture generated. In addition, the understandability of the SOPLE-DE documentation, and the applicability of the SOPLE-DE in practice will be analyzed considering the experience of the subjects.

### **Selection of Subjects**

The selection of subjects is closely related to the generalization of the results from the experiment. In order to generalize the results to the desired population, the selection must be representative for that population (Wohlin *et al.*, 2000). The selection of subjects is also called a sample from the population. Ideally, this selection should be performed randomly.

The subjects of the experimental study will act as domain architect and SOA architect as defined in the SOPLE-DE. In this experiment, the subjects will be selected using a *Convenience Sampling*, in which the nearest and most convenient people are selected, i.e., this selection is not totally randomized (Wohlin *et al.*, 2000).

The larger the sample is, the lower the errors become when generalizing the results. However, if there is a large variability in the population, a larger sample size is needed. In this experiment, the variability of the population is not very large, since all of the subjects



Factor	
The quality, i.e., <i>Coupling</i> and <i>Instability</i> , of the SO-PLA produced.	
Treatment 1	Treatment 2
Design the SO-PLA without a structured method.	Produce the SO-PLA following the SOPLE-DE.

**Table 6.1** One Factor with Two Treatments Design

have degree in computer science; they are all postgraduate students; and all have attended a similar set of disciplines in their postgraduate courses. However, the experience of the subjects may be significantly different because some of them have worked for different organizations.

### Experiment Design

A design of an experiment describes how the tests are organized and run. In this experiment, the *One Factor with Two Treatments* design will be used as illustrated in Table 6.1 (Wohlin *et al.*, 2000). In the context of experimentation, there are three general design principles that are frequently used in experimental studies:

1. **Randomization:** It is the most important design principle. It is used in the selection of the subjects and in the assignment of subjects to treatments. Ideally, the subjects must be selected randomly from a set of candidates, and they should be assigned to treatments randomly. In this experiment, the assignments of subjects to the treatments will be done randomly;
2. **Blocking:** It is used to systematically eliminate the undesired effect in the comparison among the treatments. In this experiment, the experience of subjects provided using a questionnaire may be used to group subjects with similar profiles;
3. **Balancing:** If we assign treatments so that each treatment has an equal number of subjects, we have a balanced design. Balancing is desirable because it both simplifies and strengthens the statistical analysis of the data. In this experiment, we will try to balance the experiment, i.e., select the same number of subjects per treatment. However, it will depend on the number of volunteers found to perform the experiment.

### Instrumentation

All the subjects will receive a questionnaire (QT1) about his/her education and experience. This questionnaire will be used to evaluate their educational background, participation in software development projects, and experience in SOA, software product lines and software reuse.

In order to guide the participants in the experiment, the complete description of the SOPLE-DE, with all supporting material, such as templates and guidelines will be provided by the experimenter. In addition, the requirements of the service-oriented product line on the *Travel Reservation* domain, i.e., the project that will be used in this experiment, will be given to the participants as well. The following documents will be provided:

1. **SOPLE-DE:** Complete description of the activities and tasks of the SOPLE-DE;
2. **Architecture Template:** A document template to document the service-oriented product line architecture;
3. **Travel Reservation Requirements:** The business processes, feature model, quality attribute scenarios and use cases explaining the requirements and the variability of the service-oriented product line.

In addition, the material also includes a second questionnaire (QT2) for the evaluation of the difficulties found by the participants when reading and using the approach in practice. This questionnaire will be used to identify possible misunderstandings and applicability problems during the execution of the SOPLE-DE. The architecture template can be seen in Appendix [A](#), and the questionnaires in Appendix [B](#).

### Validity Evaluation

A fundamental question concerning results from an experiment is how valid the results are. Adequate validity refers to that the results should be valid for the population of interest. In other words, the results are said to have adequate validity if they are valid for the population to which we would like to generalize ([Wohlin et al., 2000](#)). In this study, we consider four types of validity as described next.

**Conclusion validity:** Threats to the conclusion validity are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment. In this experiment, the following conclusion validities were considered:

- **Reliability of measures:** The validity of an experiment is highly dependent on the reliability of the measures. In this experiment, we have not found baseline values for the metrics used in the context of service-oriented development. Thus, this issue can be a problem since we do not have baselines used in service orientation with empirical evaluations in order to compare our finds;
- **Random heterogeneity of subjects:** There is always heterogeneity in a study group. If the group is very heterogeneous, there is a risk that the variation due to individual differences is larger than due to the treatment. In this sense, in the experiment we will try to reduce the group heterogeneity, and the experiment will be conducted with postgraduate students that do research in the same area and attended to a similar set of postgraduate disciplines;
- **Experience of the subjects:** Subjects without experience can also affect the experiment, since it is harder for them to understand the process. To mitigate the lack of experience, we will provide training.

**Internal validity:** Threats to internal validity are influences that can affect the independent variable with respect to causality, without the knowledge of the researcher. It is the capacity to replicate the experiment using the same subjects and objects. The following internal validity was used:

- **Maturation:** Subjects react differently when performing the experiment. Some participants can be affected negatively (tired or bored), while others positively (learning with practice). In this sense, the subjects performing the experiment will be volunteers, so they have at least some interest in the study.

**Construct validity:** It concerns the ability to generalize results of the experiment outside the experiment setting. In this experiment, the following construct validity was considered:

- **Mono-operation bias:** If the experiment includes a single independent variable, case, subject or treatment, the experiment may under-represent the construct and thus not give the full picture of the theory. In this sense, it would be better if we could analyze the SOPLE-DE comparing it with other service-oriented product line design approach. However, we could not find any other systematic and structured approach since the combination of SPL and SOA is still emerging. In this sense, we will compare the SOPLE-DE with an ad-hoc development.

**External validity:** Threats to external validity are conditions that limit our ability to generalize the results of our experiment to an industrial practice. In this experiment, the following external validity was considered:

- Interaction of setting and treatment: This is the effect of not having the experimental settings or material representative of, for example, industrial practices. In this experiment, we used the *Travel Reservation* domain that involves integration problems and other real SOA difficulties. This domain is commonly used in SOA works, e.g., ([Snell, 2002](#)), and represents a real and complex problem.

### 6.3.3 Operation

This section presents the details about the execution of the experimental study performed with the purpose of evaluating and refining the SOPLE-DE.

#### The Environment

The experimental study was conducted during 8 hours at the Federal University of Pernambuco (UFPE). The experimental study was composed of 8 subjects that performed the experiment in parallel. In the experiment, three service-oriented systems were designed as a service-oriented product line.

#### Training

The subjects were trained before the experimental study began. The training took 22 hours, divided into 10 lectures with two hours each, during the postgraduate course at the university, and 2 hours independently of the university course presented by the experimenter. In addition, the subjects who used the proposed approach were trained 2 hours more to use the SOPLE-DE.

As previously described, the study was performed in two steps: initially, the subjects were trained in several aspects of software reuse, SPL, SOA, reuse processes and SOPLE-DE, and after, they performed the service-oriented product line project in 8 hours.

#### Subjects

The subjects were four M.Sc. and four Ph.D. students from the Federal University of Pernambuco. However, three subjects were not considered during the analysis of the

ID	Academic Projects	Industrial Projects	SPL Projects	SOA Projects
1	(2) Low Complexity (2) Medium Complexity (1) High Complexity	(2) Low Complexity (3) Medium Complexity (1) High Complexity	(1) Academic	
2	(9) Low Complexity (3) Medium Complexity (1) High Complexity	(4) Low Complexity (6) Medium Complexity (4) High Complexity	(3) Academic	(1) Academic
3	(5) Low Complexity (5) Medium Complexity	(4) Medium Complexity (2) High Complexity	(2) Academic	
4	(1) Low Complexity (1) Medium Complexity	(1) Medium Complexity (1) High Complexity	(1) Academic	(1) Academic
5	(1) Low Complexity (1) High Complexity	(9) Low Complexity (3) Medium Complexity	(1) Academic	

**Table 6.2** The Profile of the Subjects

experiment because they did not have the necessary profile, i.e., participated as architect of an industrial project. In this sense, two M.Sc. and one Ph.D. students were removed.

All the subjects considered had industrial experience in software development, more than one year at least. Two subjects had participated in industrial projects involving some kind of reuse activity, for instance, component-based development, framework, or web services development. In addition, all the subjects had participated in SPL academic projects, and two subjects have taken part of an academic SOA project. Table 6.2 shows a summary of the profile of the subjects involved in this experiment.

### Costs

Since the subjects of the experimental study were students from the Federal University of Pernambuco and the environment for execution was the labs of the university, the cost for the study was basically planning and operation. The planning for the experimental study took about two months. During this period, it was developed three versions of the planning presented in this dissertation.

### 6.3.4 Analysis and Interpretation

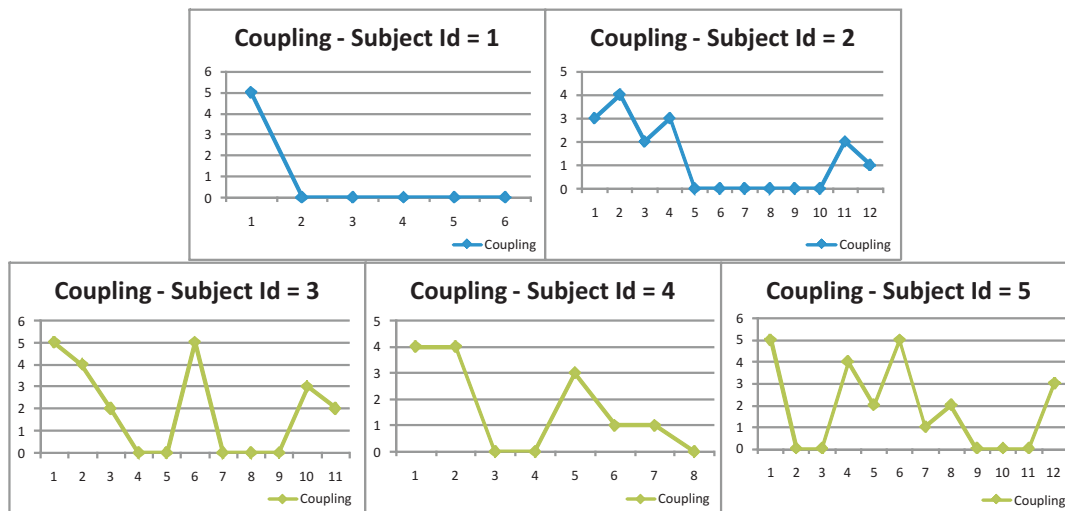
In this section, the results obtained with the experimental study are presented. This section is divided into quantitative and qualitative analysis.

#### Quantitative Analysis

The quantitative analysis was divided in five analyses: coupling and instability of the service-oriented product line architecture, service operations cohesion, difficulties found

to understand the SOPLE-DE, and the difficulties found during the use of the SOPLE-DE in practice.

**Coupling:** After collecting the information about the service coupling, the data collected was analyzed. Figure 6.3 shows the service coupling of the services identified by the subjects. In the graphics, the axis (X) shows the service identifiers<sup>1</sup>, and the axis (Y) represents the service coupling values. In addition, the subject with Id = 1 and 2 used the SOPLE-DE, while subjects with Id = 3, 4 and 5 designed the project without following a structured method (ad-hoc).



**Figure 6.3** Service Coupling

As it can be seen in figure, the coupling of the services generated using the SOPLE-DE is lower, when compared with the service coupling produced without using the structured method. Figure 6.4 compares the mean of all service coupling following the SOPLE-DE and without using any method. This aspect indicates that the null hypothesis ( $\mu_{SC}$  of services without SOPLE-DE  $<$   $\mu_{SC}$  of services with SOPLE-DE) can be rejected.

Through the analysis of empirical studies, Chidamber and Kemerer (1994) suggested that at least 50% of the classes in an object-oriented system should be totally independent of other classes, i.e., these classes should have Coupling = 0. However, we have not found any baseline for this metric in the context of service orientation. Thus, maybe we can consider that in a service-oriented system at least 50% of the services should be only providers, in other words, they should not depend on any other service. As illustrated in

<sup>1</sup>Each subject identified several services during the experiment. In the graphics, each of these services are represented with an identifier showed in the axis (X), i.e., the numbers (1, 2, ..., 12) were used as the identifiers.

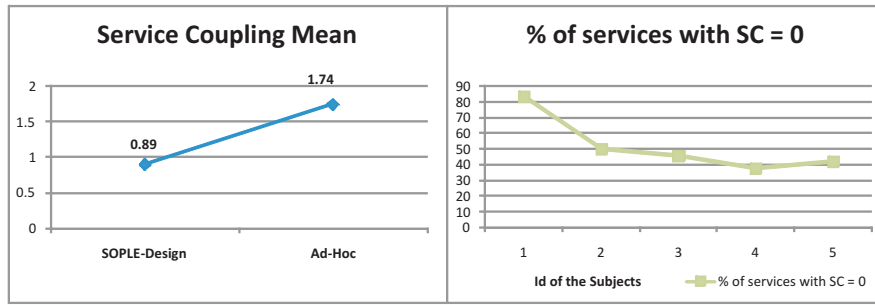


Figure 6.4 Service Coupling Mean

Figure 6.4, the architectures of the subjects with Id = 1 and 2 have 50% or more of their services with SC = 0.

**Instability:** After collecting the information about the service instability, the data collected was analyzed. Figure 6.5 shows the service instability data. In the graphics, the axis (X) shows the service identifiers, and the axis (Y) represents the service instability values. The average instability of the services generated using the SOPLE-DE is lower, when compared with the mean of the service instability produced without using the structured method (see Figure 6.6). This aspect indicates that the null hypothesis ( $\mu_{SI}$  of services without SOPLE-DE  $< \mu_{SI}$  of services with SOPLE-DE) can be rejected.

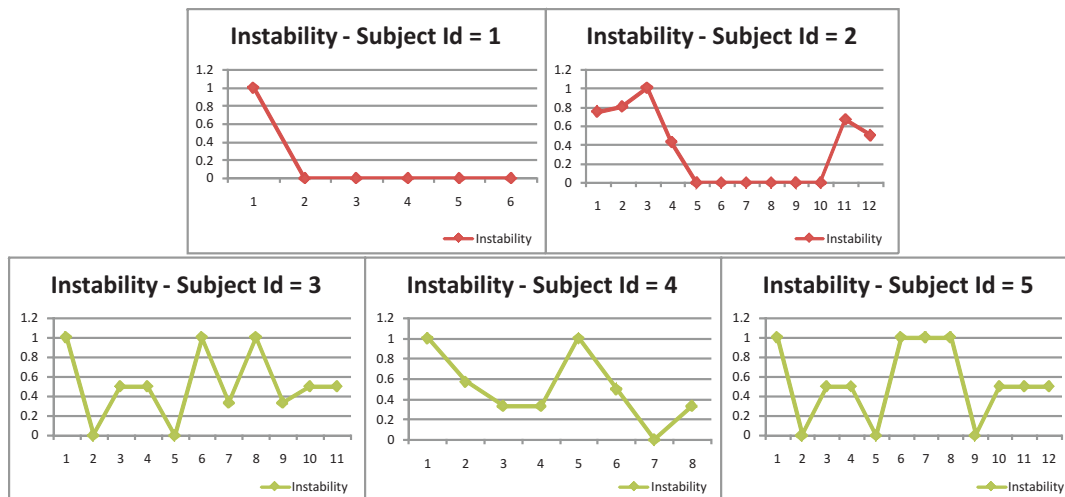


Figure 6.5 Service Instability

We have not found any baseline for the instability metric in the context of service-oriented development as well. Thus, we cannot say how good are the values obtained with the treatments. However, these values can be used in new experiments as baselines.

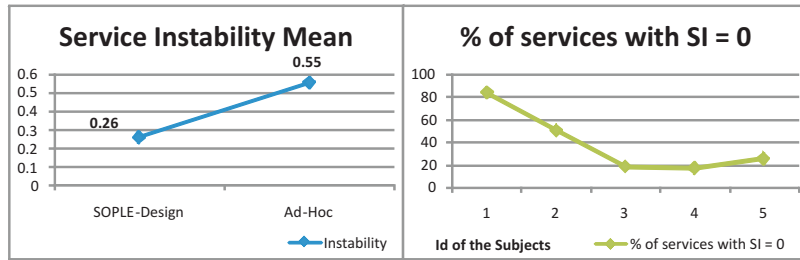


Figure 6.6 Service Instability Mean

It is important to observe that several services that the subjects (Id: 1 and 2) identified are totally stable, i.e., they have SI = 0 (see Figure 6.6).

**Lack of Service Cohesion:** After collecting the information about the service cohesion, we could observe that 16.7% of the services identified by the subjects using the SOPLE-DE have LSC > 1, i.e., the operations of these services access more than one business entity of the domain. On the other hand, 19.4% of the services identified by the subjects without use the SOPLE-DE had LSC > 1. Figure 6.7 presents the cohesion metric values. The hypothesis ( $\mu_{LSC}$  of service operations without SOPLE-DE <  $\mu_{LSC}$  of service operations with SOPLE-DE) could be rejected.

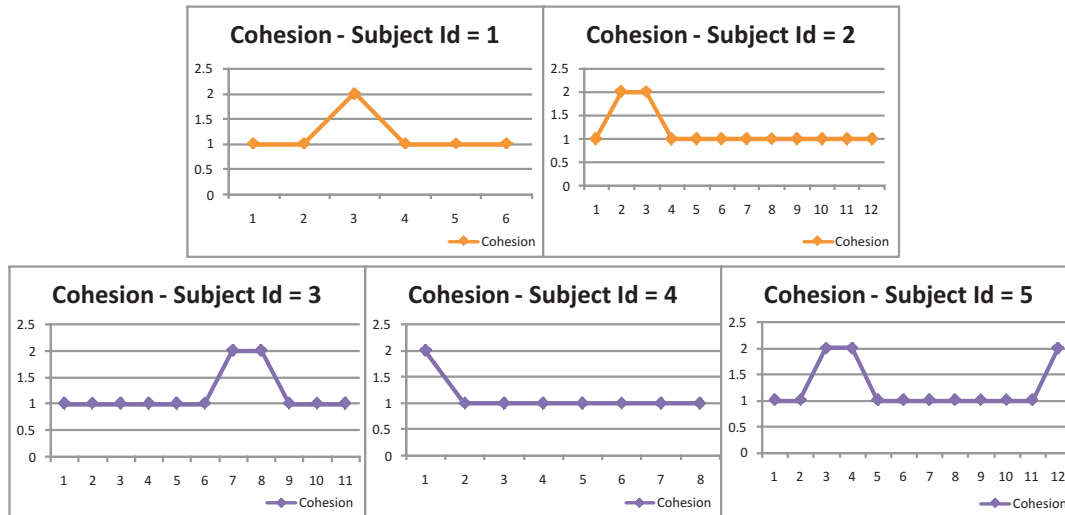


Figure 6.7 Cohesion of the Service Operations

**Difficulties to Understand the Activities of the SOPLE-DE:** Analyzing the answers of the subjects using the questionnaire (QT2) for the difficulties found to understand the SOPLE-DE activities, it was identified that the subjects (Id = 1 and 2) that used the SOPLE-DE did not have difficulties to understand the approach. Thus, this



aspect confirms that the null hypothesis (understanding problems > 50%) can be rejected. However, it is necessary to highlight that this value for the null hypothesis was defined without any previous data. Nevertheless, the next time the experiment is performed this value can be refined based on this experience, resulting in a more calibrated metric.

**Difficulties to Apply the SOPLE-DE in Practice:** Analyzing the answers of the subjects using the questionnaire (QT2) for the difficulties found to apply the SOPLE-DE in practice, it was identified that subjects (Id = 1 and 2) did not have difficulties to apply the approach in practice. Thus, this aspect indicates that the null hypothesis (applicability problems > 50%) can be rejected without much significance. However, it is necessary to highlight that the value for the null hypothesis was defined without any previous data. Thus, this value can be calibrated for new experiments.

### Qualitative Analysis

After concluding the quantitative analysis of the experiment, the qualitative analysis was performed. This analysis was based on the answers of the subjects defined in the questionnaire (QT2).

**Training Analysis:** The training was applied to all the subjects who participated of the experimental study and was composed of a set of slides involving topics related to software reuse, software product lines and service-oriented architectures. The training was performed in 24 hours. Two subjects considered the training very good (Id = 6 and 7) and three subjects classified it as good (Id = 1, 2 and 5). The scale defined was: very good, good, regular, and unsatisfactory.

Finally, the subjects (Id = 1 and 2) were trained to use the SOPLE-DE. The subjects (Id = 5, 6 and 7) were not trained to use the SOPLE-DE, since they designed the project without using a structured method (ad-hoc).

**Usefulness of the SOPLE-DE:** The four subjects that used the SOPLE-DE reported that the approach was useful to perform the service-oriented domain design. However, one subject (Id = 3) indicated some improvements in the architectural elements identification activity with the purpose of facilitating the identification of modules and their classification as components or services. All the issues raised during the experiment were considered, and the SOPLE-DE was refined. These issues were detailed during the discussion about the difficulties to understand and apply the SOPLE-DE in practice.

**Quality of the Documentation and Instruments:** One subject (Id = 2) complained about the lack of examples in the documentation of the SOPLE-DE to clarify the different activities of the approach. In this sense, an example was put in the approach

documentation to ease understanding.

Regarding the instruments of the experiment, two subjects (Id = 2 and 6) complained about the lack of details in the requirements of the service-oriented product line on the *Travel Reservation* domain. It was proposed to include more information for new experiments. The subjects (Id = 2 and 6) were the only two participants that had experience with SOA projects in academy. Thus, this experience may have influenced these subjects to complain about the lack of information in the requirements.

### **Quality of the Architecture Document Produced by the Subjects**

We used the following scale to measure the quality of the architecture documents produced by the subjects: very good, good, regular, and unsatisfactory. It was noticed that subjects (Id = 1 and 2) produced a well-structured document, since they followed the SOPLE-DE template strictly. In this sense, we classified the architecture documents produced by these subjects as good.

Considering the architecture documents produced by the subjects without use the SOPLE-DE, we could detect that subject (Id = 5 and 6) produced good architecture documents as well, however, with different sections, since they do not follow a template. Regarding subject (Id = 7), the architecture document was classified as regular, since it was not organized and well-structured as the documents of the other subjects.

## **6.4 Conclusions and Lessons Learned**

Even with the analysis not being conclusive, the experimental study indicates that the SOPLE-DE allows the architects to design service-oriented product line architectures with a good coupling and stability, and services with cohesive operations. Additionally, the aspects related to understanding and applicability of the SOPLE-DE in practice returned satisfactory results. Moreover, with the results identified in this experiment, the metric values can be calibrated in a more accurate way. It was also identified that the training on SOA and SOPLE-DE should be reconsidered, it should be longer, and the requirements of the experiment project should be more detailed to improve the results of future experiments.

We have not found correlations among the experience of the subjects and the quality of the architecture produced, i.e., the results of the experiment did not show that subjects with a significance experience produced the service-oriented product line architecture with better coupling, instability and cohesion. Considering the understanding and applicability

of the SOPLE-DE, this correlation could be analyzed as described previously. New experiments can provide more evidence for these correlations. In addition, we have not considered any value as outlier. In this sense, no value was removed when analyzing the metrics results ([Fenton, 1994](#)).

After concluding the experimental study, some aspects should be considered in order to repeat the experiment, since they were seen as limitations of the first execution. In this sense, the next sub-sections present the lessons learned with the experimental study performed.

### **Problem Description**

It is important to specify the system that will be used in the experimental study as more detailed as possible. One subject highlighted some problems related to the specifications of the use cases during the pilot project. During the real experiment, two subjects (Id = 2 and 6) asked about details that were not mentioned in the material provided by the experimenter. It was noted that a full example during the training can avoid doubts during the execution of the real experiment. A more detailed specification of the project used in the experiment is extremely necessary.

### **Single Roles**

During the experiment, the subjects were associated to two roles, i.e., domain architect and SOA architect. This could have a negative impact on the project, mainly related to overworking and lack of experience of the subjects in specific roles. This issue was already identified in ([de Almeida, 2007](#)), but we could not avoid it due to the reduced number of volunteers found to perform the experimental study. This issue could be solved if we could separate several groups of people, five subjects each in the context of the SOPLE-DE, to realize the experiment acting just as a single role.

### **Motivation**

As the project was performed independently of a university course, it was difficult to maintain the subjects motivated, and keep their attention and discipline during the whole execution of the experimental study. Thus, this aspect should be analyzed in order to try to control it. A possible solution can be to perform the experimental study in a university course.

### **Number of subjects**

It is very hard to find volunteers to perform the experimental study. This experiment was performed by a reduced number of subjects (8 participants), and the pilot project with 2 subjects. After this experimental study, the necessity to increase the number of subjects could be identified. The execution of the experiment in a university course may solve this issue as well.

## **6.5 Chapter Summary**

This Chapter presented the definition, planning, operation, analysis and interpretation of the experimental study that evaluated the efficacy, understanding and applicability of the SOPLE-DE. The study analyzed the possibility of subjects using the approach to design a service-oriented product line architecture with good stability and coupling, and services with cohesive operations. It was also analyzed the understanding and applicability of the SOPLE-DE in practice. The difficulties were categorized in the different activities of the SOPLE-DE with the intention of evaluating and refining its activities.

Even with the reduced number of subjects, the analysis has shown that the SOPLE-DE can be viable. It also identified some issues for improvements. However, two aspects should be considered: the repetition of the study in different contexts and new studies based on observation in order to identify more problems and new points for improvements.

Next chapter presents the conclusions of this work, its related work and directions for future work.

# 7

## Conclusions

The software industry is constantly searching for new ways to achieve productivity gains, reduced development costs, improved time-to-market, and increased software quality (Linden *et al.*, 2007). Organizations aim to accomplish these goals with the purpose of maintaining competitive in the current business environment.

In this scenario, software reuse is a key factor to achieve these goals (Krueger, 1992). In the context of software reuse, SPL and SOA are strategies that are getting a lot of attention in research and practice lately. These strategies share common goals and characteristics as presented in Chapter 2 and 3, which depicted an overview on SPL and SOA concepts respectively.

In this context, some works have considered the combination of SPL and SOA concepts in a unique development process, e.g., Lee *et al.* (2007), Helferich *et al.* (2007) and Naab (2009). However, there are few works that propose processes for service-oriented product lines, and they do not provide sequential activities, sub-activities, roles, inputs and outputs in a systematic manner.

Thus, in order to solve the problems and gaps identified in the service-oriented product line area, this dissertation presented the SOPLE-DE. It is an approach to design service-oriented product line architectures, which defines a systematic way to perform the architecture design based on a set of principles. The SOPLE-DE as well as its principles, guidelines, activities, sub-activities, inputs, outputs and roles are presented in Chapter 5.

The SOPLE-DE approach was based on an extensive review of existing service-oriented processes, considering their weak and strong points, and gaps in the research area. In the Chapter 4, the current state-of-the-art of SOA design methods is discussed.

An experimental study on the *Travel Reservation* domain is presented in Chapter 6. It was performed with the purpose of evaluating the SOPLE-DE approach and refining it considering the feedbacks received during the execution of the experiment and its results.

Finally, this chapter concludes this dissertation presenting its conclusions, and its related and future work. Next section presents the related work that have considered the combination of SPL and SOA, compared their concepts or presented information about their similarities and differences.

## 7.1 Related Work

In the literature, few works have considered the combination of SPL and SOA concepts and processes for service-oriented product lines. However, the key difference between the work described in this dissertation and the others is the systematization of the design process, which presented a well-defined sequence of activities, sub-activities and steps with clearly defined inputs and outputs, and performed by a predefined set of roles with clear responsibilities. This helped to reduce the gaps and lack of details among the few steps and activities provided by the existing approaches and processes for service-oriented product lines.

An approach for developing service-oriented product lines was published in [Lee et al. \(2007\)](#) and [Lee et al. \(2008\)](#). These works present an initial development process that provides methods for the identification and documentation of services and service compositions. They focus on the development of service-oriented product lines that are automatically configurable at runtime. In this case, SOA concepts, e.g., runtime service discoverability and binding, are used to aid the development of Dynamic Software Product Lines (DSPL).

Our work considers the identification of services from different sources, e.g., feature models, use cases, business processes and quality attribute scenarios. The related work mentioned does not use business processes to identify services. It is a negative point, since the business processes are the focus of the service-oriented development. In addition, our work provides guidelines to achieve some quality attributes, such as performance, flexibility and evolvability, that are not mentioned in the related work. Moreover, SOPLE-DE suggests a more detailed specification for services and service compositions, which include information such as service level agreements and non-functional requirements.

The concept of Business Process Lines (BPL) was used in [Ye et al. \(2007\)](#) and [Boffoli et al. \(2008\)](#). In this context, business processes are developed with variability in order to fit the requirements of several customers. Thus, the business process activities are customized, the optional and alternative ones are selected or excluded from the generic process, and the target business process is created. Afterwards, the SOA system for this

specific business process is developed.

SOPLE-DE considers variability in the business processes as well. However, it also uses feature models to represent variability, since the business process models can become polluted with too much information about variability. Other key difference of our work is that we consider reuse of services as well as artifacts produced during the process, which are developed with variability to be reused by several systems of the product line. The related work considers only reuse of services and puts all the variability information in the business processes.

In addition, the related work is technology specific, i.e., they use web services and BPEL. At least, the related work provides tool support. The work presented in this dissertation is intended to depict guidelines and activities during the process without restrict it to a specific technology such as web services.

In [Günther and Berger \(2008\)](#), it is presented an initial process for service-oriented product lines, but it basically compares SOA and SPL processes. It discusses about the implementation of service variability using code transformation tools in the *Web Store* domain. However, the related work provides no sequential activities and tasks to aid the development of service-oriented product lines, different from SOPLE-DE that is systematic.

## 7.2 Future Work

Due to the time constraints imposed on the master degree, this work can be seen as an initial climbing towards a process for service-oriented product lines, and interesting directions remain to improve what was started here and new routes can be explored in the future. Thus, the following issues should be investigated as future work:

- **Re-engineering activities:** The proposed approach does not consider reengineering aspects, such as the identification of services from existing legacy applications, nor redesign of existing services and components for the new service-oriented product line. However, it does not exclude the possibility to integrate existing components into the new architecture. Integration is one of the main benefits of SOA due to its interoperability characteristics that allow legacy systems developed using different platforms and languages to be leveraged in the new solution ([Arsanjani et al., 2008](#)). A deep analysis of re-engineering aspects during the design approach, i.e., the bottom-up strategy ([Erl, 2005](#)), is an important advancement that can be considered as future work;

- **Extractive and reactive adoption:** SPL can be adopted using different strategies, e.g., proactive, extractive and reactive adoption models. This work uses a proactive adoption model, which concentrates on the development of a SPL from scratch. Thus, the adoption of SPL considering existing products (extractive) or the incremental development of products (reactive) are not being considered in this work. In this sense, these adoption models can be considered as future work to provide a better design discipline;
- **Experimental Study:** This dissertation presented the definition, planning, operation, analysis and interpretation of an experimental study that was executed with the purpose of evaluating and refining the SOPLE-DE approach. However, new studies in different contexts, including more subjects and other domains are still necessary in order to calibrate the proposed approach;
- **Full Process:** In the Chapter 5, we presented an approach to design service-oriented product line architectures. However, a process that considers all the disciplines, e.g., requirements, design and implementation, during the development of product lines using service-oriented architectures is still missing in the literature.
- **Other Directions in Software Development:** The approach to design service-oriented product line architectures proposed in this dissertation is based on four solid concepts: service orientation, component-based, product line and design principles, e.g., granularity, cohesion, coupling and variability. However, new directions started in the software development field such as Model-Driven Development (MDD) and can be used in the development of service-oriented product lines. Some directions in this sense are being investigated in [Boffoli \*et al.\* \(2009\)](#);
- **Architecture Evaluation:** The software architecture is a key asset for any organization that builds complex software-intensive systems. In SPL, this issue is even more important, since the architecture should be used to create several products. In addition, service-oriented architectures are distributed, thus, it is critical to perform an architecture evaluation early in the software life-cycle to avoid failures of quality attributes, e.g., security, performance, availability, and modifiability ([Bianco \*et al.\*, 2007](#)). Even being very important in SPL and SOA processes, this aspect was not considered in this dissertation and can be considered as a future work.



## 7.3 Concluding Remarks

Software reuse is a key factor for organizations interested in improvements related to productivity, quality and cost reductions. In this context, this work presented the SOPLE-DE, which is an approach to design service-oriented product line architectures. It combines SPL and SOA concepts focusing on increasing reuse and productivity.

The motivation to combine SPL and SOA was to achieve desired benefits, such as productivity gains, decreased development costs and effort, improved time-to-market, applications customized to specific customers or market segment needs, competitive advantage, flexibility and dynamic composition of software modules (Cohen and Krut, 2007). In particular, this work provided contributions to develop service-oriented systems customizable to specific customers through increased flexibility and dynamic compositions of software modules.

The SOPLE-DE approach was based on an extensive review of the available service-oriented processes, their weak and strong points and gaps in the area. The SOPLE-DE can be seen as a systematic way to design service-oriented product line architectures through a well-defined sequence of activities, steps, inputs, outputs, and guidelines.

Additionally, the approach was evaluated in a service-oriented product line project through an experimental study on the *Travel Reservation* domain, which analyzed it both quantitatively and qualitatively. This experimental study presented findings that the SOPLE-DE can be viable to aid software architects during the design of service-oriented product line architectures.

Even it being an important contribution for the field, new routes need to be investigated in order to define a more complete process that consider all the software development disciplines, such as requirements, design and implementation, for product lines based on services.

# Bibliography

- Abu-Matar, M. (2007). Toward a service-oriented analysis and design methodology for software product lines. *IBM Developer Works*.
- Acuña, C. J. and Marcos, E. (2006). Modeling semantic web services: a case study. In *ICWE'06: Proceedings of the 6th International Conference on Web Engineering*, pages 32–39, New York, NY, USA. ACM.
- Alvaro, A., Almeida, E. S., and Meira, S. L. (2006). A software component quality model: A preliminary evaluation. In *Proceedings of the 32nd Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, pages 28–37, Washington, DC, USA. IEEE Computer Society.
- Arsanjani, A. (2004). Service-oriented modeling and architecture. Technical report, Service-Oriented Architecture and Web services Center of Excellence, IBM.
- Arsanjani, A. and Allam, A. (2006). Service-oriented modeling and architecture for realization of a SOA. In *SCC '06: Proceedings of the IEEE International Conference on Services Computing*, page 521, Washington, DC, USA. IEEE Computer Society.
- Arsanjani, A., Zhang, L.-J., Ellis, M., Allam, A., and Channabasavaiah, K. (2007). S3: A service-oriented reference architecture. *IT Professional*, **9**(3), 10–17.
- Arsanjani, A., Ghosh, S., Allam, A., Abdollah, T., Ganapathy, S., and Holley, K. (2008). SOMA: A method for developing service-oriented solutions. *IBM System Journal*, **47**(3), 377–396.
- Azevedo, L. G., Santoro, F., Baião, F., Souza, J., Revoredo, K., Pereira, V., and Herlain, I. (2009). A method for service identification from business process models in a SOA approach. In *10th International Workshops on Business Process Modeling, Development and Support (BPMDS)*, volume 29 of *Lecture Notes in Business Information Processing*, pages 99–112. Springer Berlin Heidelberg.
- Barbacci, M., Longstaff, T. H., Klein, M. H., and Weinstock, C. B. (1995). Quality attributes. Technical report, Software Engineering Institute.
- Basili, V., Caldiera, G., and Rombach, D. H. (1994). The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley.

- Basili, V. R. (1996). The role of experimentation in software engineering: Past, present and future. In *ICSE'96: 18th International Conference on Software Engineering*.
- Basili, V. R., Selby, R. W., and Hutchens, D. H. (1986). Experimentation in software engineering. *IEEE Transactions on Software Engineering*.
- Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Bianco, P., Kotermanski, R., and Merson, P. (2007). Evaluating a service-oriented architecture. Technical report, Software Engineering Institute.
- Boehm, B. W. (1988). A spiral model of software development and enhancement. *IEEE Computer Society*, **21**(5), 61–72.
- Boffoli, N., Caivano, D., Castelluccia, D., Maggi, F. M., and Visaggio, G. (2008). Business process lines to develop service-oriented architectures through the software product lines paradigm. In *SOAPL'08: 2nd Workshop on Service-Oriented Architectures and Software Product Lines*, pages 143–147.
- Boffoli, N., Cimitile, M., Maggi, F. M., and Visaggio, G. (2009). Managing SOA system variation through business process lines and process-oriented development. In *SOAPL'09: 3rd Workshop on Service-Oriented Architectures and Software Product Lines*.
- Booch, G. (1995). *Object Solutions: Managing the Object-Oriented Project*. Addison-Wesley.
- Bosch, J. (2000). *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley.
- Brito, K. S. (2007). *LIFT: A Legacy InFormation retrieval Tool*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.
- Brown, A., Johnston, S., and Kelly, K. (2003). Using service-oriented architecture and component-based development to build web service applications. IBM Developer Works.
- Carter, S. (2007). *The New Language of Business: SOA & Web 2.0*. IBM Press.
-

- Cavalcanti, Y. C. (2009). *BAST: A Bug Report Analysis and Search Tool*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.
- Chang, S. H. (2007). A systematic analysis and design approach to develop adaptable services in service oriented computing. *IEEE Congress on Services*, pages 375–378.
- Chang, S. H. and Kim, S. D. (2007). A service-oriented analysis and design approach to developing adaptable services. In *SCC'07: Proceedings of the IEEE International Conference on Services Computing*.
- Chappell, D. (2004). *Enterprise Service Bus: Theory in Practice*. O'Reilly Media.
- Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*.
- Clements, P. (2002). Being proactive pays off. *IEEE Software*, **19**(4), 28–30.
- Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley.
- Cohen, S. and Krut, R., editors (2007). *Proceedings of the First Workshop on Service-Oriented Architectures and Software Product Lines, 11th International Software Product Line Conference*.
- de Almeida, E. S. (2007). *RiDE: The RiSE Process for Domain Engineering*. Ph.D. thesis, Federal University of Pernambuco.
- de Almeida, E. S., Alvaro, A., Lucredio, D., Garcia, V., and Meira, S. R. L. (2004). Rise project: Towards a robust framework for software reuse. In *IRI'04: International Conference on Information Reuse and Integration*.
- de Castro, V., Marcos, E., and López-Sanz, M. (2006). A model driven method for service composition modelling: a case study. *International Journal of Web Engineering and Technology*, **2**(4), 335–353.
- Dias Jr., J. J. L. (2008). *A Software Architecture Process for SOA-based Enterprise Applications*. Master's thesis, Federal University of Pernambuco, Brazil.
- Elfatratry, A. and Layzell, P. (2004). Negotiating in service-oriented environments. *Communications of the ACM*, **47**(8).
-

- Endrei, M., Ang, J., Arsanjani, A., Chua, S., Comte, P., Krogdahl, P., Luo, M., and Newling, T. (2004). *Patterns: Service-Oriented Architecture and Web Services*. IBM Redbooks.
- Engels, G., Hess, A., Humm, B., Juwig, O., Lohmann, M., Richter, J.-P., Voß, M., and Willkomm, J. (2008). A method for engineering a true service-oriented architecture. In *ICEIS'08: International Conference on Enterprise Information Systems*, pages 272–281, Barcelona, Spain.
- Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, Upper Saddle River, NJ, USA.
- Erl, T. (2006). Top 8 SOA Adoption Pitfalls. <http://www.infoq.com/articles/Top-8-SOA-Adoption-Pitfalls>.
- Erl, T. (2007). *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
- Erl, T. (2009). *SOA Design Patterns*. Prentice Hall PTR.
- Erradi, A., Anand, S., and Kulkarni, N. (2006). SOAF: An architectural framework for service definition and realization. In *SCC'06: International Conference on Services Computing*, pages 151–158. IEEE Computer Society.
- Etxeberria, L., Sagardui, G., and Belategi, L. (2007). Modelling variation in quality attributes. In *VaMoS'07: 1st International Workshop on Variability Modelling of Software-Intensive Systems*.
- Fenton, N. (1994). Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*.
- Frakes, W. (1994). Systematic software reuse: a paradigm shift. In *ICSR'94: 2nd International Conference on Software Reuse*.
- Gacek, C. and Anastasopoulos, M. (2001). Implementing product line variabilities. *SSR'01: Symposium on Software Reusability*, **26**(3), 109–117.
- Garcia, V. C., Lisboa, L. B., ao, F. A. D., Almeida, E. S., and Meira, S. R. L. (2008). A lightweight technology change management approach to facilitating reuse adoption.
-

- In *2nd Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS'08)*, Porto Alegre, Brazil.
- Garlan, D. and Shaw, M. (1994). An introduction to software architecture. Technical report, Software Engineering Institute.
- Gomaa, H. (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley.
- Günther, S. and Berger, T. (2008). Service-oriented product lines: Towards a development process and feature management model for web services. In *SOAPL'08: 2nd Workshop on Service-Oriented Architectures and Software Product Lines*, pages 131–136.
- Havey, M. (2005). *Essential Business Process Modeling*. O'Reilly.
- Helferich, A., Herzwurm, G., and Jesse, S. (2007). Software product lines and service-oriented architecture: A systematic comparison of two concepts. In *SPLC'07: 11th International Software Product Line Conference*. IEEE Computer Society.
- Henderson-Sellers, B. (1995). *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall PTR.
- Hewitt, E. (2009). *Java SOA Cookbook*. O'Reilly.
- Hofmeister, H. and Wirtz, G. (2008). Supporting service-oriented design with metrics. In *EDOC'08: 12th Enterprise Distributed Object Computing Conference*.
- Ibrahim, D. and Misie, V. B. (2006). Service views: a coherent view model of the SOA in the enterprise. In *SCC '06: Proceedings of the IEEE International Conference on Services Computing*.
- Istoan, P. (2009). *Software Product Lines for Creating Service-Oriented Applications*. Master's thesis, Irista Rennes Research Institute.
- Jarczyk, A. P. J., Löffler, P., and Shipman, F. M. (1992). Design rationale for software engineering: A survey. In *HICSS'92: 25th Hawaii International Conference on System Sciences*.
- Jones, C. (1995). *Patterns of Software System Failure and Success*. Intl Thomson Computer.
-

- Jones, S. and Morris, M. (2005). A methodology for service architectures. Technical report, Organization for the Advancement of Structured Information Standards (OASIS).
- Josuttis, N. M. (2007). *SOA in Practice*. O'Reilly.
- Kang, K. C., Lee, J., and Donohoe, P. (2002). Feature-oriented product line engineering. *IEEE Software*.
- Karhunen, H., Jantti, M., and Eerola, A. (2005). Service-Oriented Software Engineering (SOSE) framework. *ICSSSM'05: International Conference on Service Systems and Service Management*, **2**, 1199–1204 Vol. 2.
- Kästner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in software product lines. In *30th International Conference on Software Engineering (ICSE)*, pages 311–320.
- Kim, S., Kim, M., and Park, S. (2008). Service identification using goal and scenario in service oriented architecture. In *APSEC'08: 15th Asia-Pacific Software Engineering Conference*, pages 419–426. IEEE Computer Society.
- Kitchenham, B. (2004). Procedures for performing systematic reviews. Technical report, Joint Technical Report, Keele University TR/SE-0401 and NICTA 0400011T.1.
- Kitchenham, B. and Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report 2007-001, Keele University and Durham University Joint Report.
- Kitchenham, B., Pickard, L., and Pfleeger, S. L. (1995). Case studies for method and tools evaluation. *IEEE Software*.
- Kruchten, P. (1995). Architectural blueprints - the 4+1 view model of software architecture. *IEEE Software*.
- Kruchten, P. (2003). *The Rational Unified Process: An Introduction*. Addison Wesley, third edition.
- Krueger, C. (2002). Eliminating the adoption barrier. *IEEE Software*, **19**(4), 29–31.
- Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys*, **24**(2).
- Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications.
-

- Lamparter, S. and Sure, Y. (2008). An interdisciplinary methodology for building service-oriented systems on the web. In *SCC'08: Proceedings of the IEEE International Conference on Services Computing*.
- Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall.
- Lee, J. and Kang, K. C. (2003). Feature binding analysis for product line component development. In *PFE'03: 5th International Workshop on Software Product-Family Engineering*, pages 250–260.
- Lee, J., Muthig, D., and Naab, M. (2007). Identifying and specifying reusable services of service centric systems through product line technology. In *SPLC'07: 11th International Software Product Line Conference*. IEEE Computer Society.
- Lee, J., Muthig, D., and Naab, M. (2008). An approach for developing service-oriented product lines. In *SPLC'08: 12th International Software Product Line Conference*, pages 275–284. IEEE Computer Society.
- Lee, J., Kotonya, G., and Robinson, D. (2009). A negotiation framework for service-oriented product line development. In *ICSR'09: 11th International Conference on Software Reuse*.
- Linden, F. J. v. d., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- López-Sanz, M., Acuña, C. J., Cuesta, C. E., and Marcos, E. (2007). UML profile for the platform independent modelling of service-oriented architectures. In *ECSA'07: Proceedings of the 1st European conference on Software Architecture*, pages 304–307, Berlin, Heidelberg. Springer-Verlag.
- MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., and Metz, R. (2006). Reference model for service oriented architecture. Technical report, Organization for the Advancement of Structured Information Standards (OASIS).
- Martin, R. (1994). OO design quality metrics: An analysis of dependencies. <http://www.objectmentor.com/resources/articles/oodmetrc.pdf>.



- Martins, A. C., Garcia, V. C., Almeida, E. S., and Meira, S. R. L. (2008). Enhancing components search in a reuse environment using discovered knowledge techniques. In *2nd Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS'08)*, Porto Alegre, Brazil.
- Mascena, J. C. C. P. (2006). *ADMIRE: Asset Development Metric-based Integrated Reuse Environment*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.
- McGovern, J., Tyagi, S., Stevens, M., and Mathew, S. (2003). *Java Web Services Architecture*. Morgan Kaufmann.
- Medeiros, F. M., de Almeida, E. S., and Meira, S. R. L. (2009). Towards an approach for service-oriented product line architectures. In *SOAPL'09: 3rd Workshop on Service-Oriented Architectures and Software Product Lines*.
- Mendes, R. C. (2008). *Search and Retrieval of Reusable Source Code using Faceted Classification Approach*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.
- Naab, M. (2009). Achieving true flexibility of SOA-based information systems by adopting practices from product line engineering. In *Software Product Lines Doctoral Symposium - 13th International Software Product Line Conference*.
- Nascimento, L. M. (2008). *Core Assets Development in SPL - Towards a Practical Approach for the Mobile Game Domain*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.
- Neiva, D. F. S. (2009). *RiPLE-RE: A Requirements Engineering Process for Software Product Lines*. Master's thesis, Federal University of Pernambuco.
- Panda, D., Rahman, R., and Lane, D. (2007). *EJB 3 in Action*. Manning Publications.
- Papazoglou, M. P. and Heuvel, W.-J. V. D. (2006). Service-oriented design and development methodology. *International Journal of Web Engineering and Technology (IJWET)*, 2(4), 412–442.
- Pereplechikov, M., Ryan, C., Frampton, K., and Tari, Z. (2007). Coupling metrics for predicting maintainability in service-oriented designs. In *ASWEC'07: Australian Software Engineering Conference*.
-

- Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
- Quynh, P. T. and Thang, H. Q. (2009). Dynamic coupling metrics for service-oriented software. *IJCSE'09: International Journal of Computer Science and Engineering*.
- Ramollari, E., Dranidis, D., and Simons, A. J. H. (2007). A survey of service oriented development methodologies. *YR-SOC'07: 2nd European Young Researchers Workshop on Service Oriented Computing*.
- Razavian, M. and Khosravi, R. (2008). Modeling variability in business process models using uml. In *ITNG'08: 5th International Conference on Information Technology - New Generations*, pages 82–87.
- Rosenberg, L. H. and Hyatt, L. (1998). Applying and interpreting object oriented metrics. In *Proceedings of the Software Technology Conference*.
- Santos, E. C. R., ao, F. A. D., Martins, A. C., Mendes, R., Melo, C., Garcia, V. C., Almeida, E. S., and Meira, S. R. L. (2006). Towards an effective context-aware proactive asset search and retrieval tool. In *6th Workshop on Component-Based Development (WDBC'06)*, pages 105–112, Recife, Pernambuco, Brazil.
- Segura, S., Benavides, D., Ruiz-Cortés, A., and Trinidad, P. (2007). A taxonomy of variability in web service flows. In *SOAPL'07: 1st Workshop on Service-Oriented Architectures and Software Product Lines*.
- Snell, J. (2002). Automating business processes and transactions in web services. Technical report, IBM.
- Souza Filho, E. D., Cavalcanti, R. O., Neiva, D. F. S., Oliveira, T. H. B., Lisboa, L. B., de Almeida, E. S., and Meira, S. R. L. (2008). Evaluating domain design approaches using a systematic review. In *ECSA'08: 2nd European Conference on Software Architecture*, pages 50–65.
- Souza Filho, E. D., de Almeida, E. S., and Meira, S. R. L. (2009). Experimenting a process to design product line architectures. In *EASA'09: Workshop on Empirical Assessment in Software Architecture*.

- Szyperski, C. (2003). Component technology: what, where, and how? In *ICSE'03: 25th International Conference on Software Engineering*, pages 684–693. IEEE Computer Society.
- Tyagi, S. (2006). Restful web services. Technical report, Sun Microsystems.
- van Gorp, J., Bosch, J., and Svahnberg, M. (2000). Managing variability in software product lines. In *LAC'00: 2nd Landelijk Architecture Congress*.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslen, A. (2000). *Experimentation in Software Engineering: An Introduction*. Springer.
- Ye, E., Moon, M., Kim, Y., and Yeom, K. (2007). An approach to designing service-oriented product-line architecture for business process families. In *ICACT'07: 9th International conference on Advanced Computing Technologies*, pages 999–1002.
- Zimmermann, O., Krogdahl, P., and Gee, C. (2004). Elements of service-oriented analysis and design. IBM Developer Works.

# Appendices



# Architecture Document Template

As part of the SOPLE-DE, detailed in Chapter 5, an architecture document template was defined with the purpose of facilitating the documentation of the service-oriented product line architecture. The next sections list all the information that should be documented.

## 1. Introduction

This introduction provides an overview of the entire software architecture document. It includes the purpose, scope, definitions, acronyms, abbreviations, and an overview of the software architecture document.

## 2. References

This section describes references to other documents, if any.

- « Document 1 Name »: « Document general description »;
- « Document 2 Name »: « Document general description »;
- « Document 3 Name »: « Document general description »;
- « Document 4 Name »: « Document general description »;
- « Document 5 Name »: « Document general description ».

---

### 3. Technologies Description

In this section, it will be described the technologies that will be applied in the service-oriented product line development. The description should be written according to the following format:

- « Technology 1 Name »: « Rationale for the selection »;
- « Technology 2 Name »: « Rationale for the selection »;
- « Technology 3 Name »: « Rationale for the selection »;
- « Technology 4 Name »: « Rationale for the selection »;
- « Technology 5 Name »: « Rationale for the selection ».

### 4. Architectural Components

In this section, it will be described the architectural components that were identified from the feature model and service candidates. This section is responsible for representing the component specification. For each component, it is presented its description, use cases, features, workflow, class diagrams and interfaces.

#### 4.1 Component Name

Give a brief description of the architectural component emphasizing its purpose. For each component in the architecture, repeat this section and the next sub-sections with the information required.

**Classes:** This section presents the component internal structure with its classes and relationships. The UML class diagrams representing the component internal behavior should be here.

**Workflow:** This section describes the component workflow represented by UML sequence or activity diagrams when necessary.

**Provided and Required Interfaces** This section presents the component provided and required interfaces and its services offered.

**Quality Attributes:** This section presents the quality attributes that the component should satisfy, e.g, modifiability, performance, and so on.

**Traceability and Internal Variability:** The traceability links of components should be documented as in Table [A.1](#).

<b>Features:</b>	« Provide the features that will be implemented by this component. »
<b>Use Cases:</b>	« Provide the use cases that will be implemented by this component. »
<b>Business Activities:</b>	« Provide the business activities that will be implemented by this component. »
<b>Interface Operations:</b>	« Provide the operations that will be implemented by this component. »
<b>Variability:</b>	« If it contains variability, give a brief description of the variation point and its variants. » « A brief description of the rationale to include variability in this component. » « Describe the variability mechanism that will be used and the rationale to select it. »

**Table A.1** Component Specification Template

## 5. Architectural Services

In this section, it will be described the architectural services that were identified from the business process models, feature model, use cases and quality attribute scenarios. This section is responsible for representing the service specification. For each service, it is presented its description, use cases, features, workflow and interfaces.

### 5.1 Service Name

Give a brief description of the architectural service emphasizing its purpose. For each service in the architecture, repeat this section and the next sub-sections with the information required.

**Workflow:** This section describes the service workflow represented by UML sequence or activity diagrams when necessary.

**Provided and Required Interfaces:** This section presents the service provided and required interfaces and its services offered.

**Quality Attributes:** This section presents the quality attributes that the service should satisfy, e.g, modifiability, performance, and so on.

**Traceability and Internal Variability:** The traceability links of services should be documented as in Table [A.2](#).

<b>Components:</b>	« Provide the components that will provide the implementation functionality to this service. »
<b>Use Cases:</b>	« Provide the use cases that will be implemented by this service. »
<b>Business Activities:</b>	« Provide the business activities that will be implemented by this service. »
<b>Interface Operations:</b>	« Provide the operations that will be implemented by this service. »
<b>Variability:</b>	« If it contains variability, give a brief description of the variation point and its variants. » « A brief description of the rationale to include variability in this service. » « Describe the variability mechanism that will be used and the rationale to select it. »

**Table A.2** Service Specification Template

---

## 6. Architectural Service Orchestrations

In this section, it will be described the architectural service orchestrations that were identified from the business process models, feature model and use cases. This section is responsible for representing the service orchestration specification. For each service orchestration, it is presented its description, features, workflow and interfaces.

### 6.1 Orchestration Service Name

Give a brief description of the architectural service orchestration emphasizing its purpose. For each service orchestration in the architecture, repeat this section and the next subsections with the information required.

**Workflow:** This section describes the service orchestration workflow represented by UML sequence or activity diagrams.

**Provided and Required Interfaces:** This section presents the service orchestration provided and required interfaces and its services offered.

**Quality Attributes:** This section presents the quality attributes that the service orchestration should satisfy, e.g, modifiability, performance, and so on.

**Traceability and Internal Variability:** The traceability links of service orchestrations should be documented as in Table A.3.

<b>Services:</b>	« Provide the services that will be orchestrated by this service. »
<b>Business Activities:</b>	« Provide the business activities that will be implemented by this service orchestration. »
<b>Features:</b>	« Provide the features that will be implemented by this service orchestration. »
<b>Interface Operations:</b>	« Provide the operations that will be implemented by this service orchestration. »
<b>Variability:</b>	« If it contains variability, give a brief description of the variation point and its variants. » « A brief description of the rationale to include variability in this service. » « Describe the variability mechanism that will be used and the rationale to select it. »

**Table A.3** Service Orchestration Specification Template

## 7. Communication Flows

In this section, it will be described the communication flows that were identified from the services and service orchestrations. This section is responsible for representing the flow specifications. For each flow, its details should be described in Table A.4.



---

<b>Architectural Elements:</b>	« Provide the architectural elements involved in the communication flow. »
<b>Protocol of Communication:</b>	« Provide the protocol used in the flow, e.g., REST and SOAP. »
<b>Type of Communication:</b>	« Provide the type of communication that will be used, e.g., synchronous and asynchronous. »
<b>Integration Mechanims:</b>	« Provide the integration mechanism that will be used in the flow, e.g, peer-to-peer or mediated by an ESB. »
<b>Variability:</b>	« If it contains variability, give a brief description of the variation point and its variants. » « A brief description of the rationale to include the variability in this flow. » « Describe the variability mechanism that will be used and the rationale to select it. »

**Table A.4** Flow Specification Template

## 8. Architectural Views

Here, it will be described the architectural views that will be used to represent the service-oriented product line architecture. The description should be written according to the following format.

### 8.1 Architectural View Name

Give a brief description of this current view and the details that should be presented in it, e.g., services and components that will be described, if all the services and components of the architecture will be represented, or only the most important ones, and so on. The diagram of the architectural view should be documented here.

# B

## Instruments of the Experimental Study

As part of the experiment instrumentation, detailed in Chapter 6, two questionnaires were defined, and applied to the subjects. The next sections list all the questions of each questionnaire.

The first questionnaire (detailed in Table B.1 and B.2) was intended to collect data about the subjects background, and the second one (detailed in Table B.3) was created with the purpose of collecting information about the use of the SOPLE-DE.

Questionnaire for Subjects Background
<b>Degree:</b> <input type="checkbox"/> Graduation. <input type="checkbox"/> Specialization. <input type="checkbox"/> M.Sc. <input type="checkbox"/> Ph.D. <b>How many years since graduation?</b> <input type="checkbox"/> years.
<b>How many industrial software projects have you participated according to the following categories?</b>  <input type="checkbox"/> Low complexity (less than 6 months). <input type="checkbox"/> Medium complexity (more than 6 months and less than a year). <input type="checkbox"/> High complexity (more than a year).
<b>What were the roles that you played in the projects cited before, e.g., architect, designer, developer, tester...?</b>

**Table B.1** Questionnaire for Subjects Background (Part 1)

<p><b>How many academic software projects have you participated according to the following categories?</b></p> <p><input type="checkbox"/> Low complexity (less than 6 months).</p> <p><input type="checkbox"/> Medium complexity (more than 6 months and less than a year).</p> <p><input type="checkbox"/> High complexity (more than a year).</p>																
<p><b>What were the roles that you played in the projects cited before, e.g., architect, designer, developer, tester...?</b></p>																
<p><b>How many SPL projects have you participated?</b></p> <p><input type="checkbox"/> None.</p> <p><input type="checkbox"/> Academic.</p> <p><input type="checkbox"/> Industrial.</p>																
<p><b>How many SOA projects have you participated?</b></p> <p><input type="checkbox"/> None.</p> <p><input type="checkbox"/> Academic.</p> <p><input type="checkbox"/> Industrial.</p>																
<p><b>How do you define your experience with software reuse?</b></p> <table> <tr> <td><b>Industrial:</b></td> <td><input type="checkbox"/> None.</td> <td><b>Academic:</b></td> <td><input type="checkbox"/> None.</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Low.</td> <td></td> <td><input type="checkbox"/> Low.</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Medium.</td> <td></td> <td><input type="checkbox"/> Medium.</td> </tr> <tr> <td></td> <td><input type="checkbox"/> High.</td> <td></td> <td><input type="checkbox"/> High.</td> </tr> </table>	<b>Industrial:</b>	<input type="checkbox"/> None.	<b>Academic:</b>	<input type="checkbox"/> None.		<input type="checkbox"/> Low.		<input type="checkbox"/> Low.		<input type="checkbox"/> Medium.		<input type="checkbox"/> Medium.		<input type="checkbox"/> High.		<input type="checkbox"/> High.
<b>Industrial:</b>	<input type="checkbox"/> None.	<b>Academic:</b>	<input type="checkbox"/> None.													
	<input type="checkbox"/> Low.		<input type="checkbox"/> Low.													
	<input type="checkbox"/> Medium.		<input type="checkbox"/> Medium.													
	<input type="checkbox"/> High.		<input type="checkbox"/> High.													
<p><b>How do you define your experience with service orientation?</b></p> <table> <tr> <td><b>Industrial:</b></td> <td><input type="checkbox"/> None.</td> <td><b>Academic:</b></td> <td><input type="checkbox"/> None.</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Low.</td> <td></td> <td><input type="checkbox"/> Low.</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Medium.</td> <td></td> <td><input type="checkbox"/> Medium.</td> </tr> <tr> <td></td> <td><input type="checkbox"/> High.</td> <td></td> <td><input type="checkbox"/> High.</td> </tr> </table>	<b>Industrial:</b>	<input type="checkbox"/> None.	<b>Academic:</b>	<input type="checkbox"/> None.		<input type="checkbox"/> Low.		<input type="checkbox"/> Low.		<input type="checkbox"/> Medium.		<input type="checkbox"/> Medium.		<input type="checkbox"/> High.		<input type="checkbox"/> High.
<b>Industrial:</b>	<input type="checkbox"/> None.	<b>Academic:</b>	<input type="checkbox"/> None.													
	<input type="checkbox"/> Low.		<input type="checkbox"/> Low.													
	<input type="checkbox"/> Medium.		<input type="checkbox"/> Medium.													
	<input type="checkbox"/> High.		<input type="checkbox"/> High.													
<p><b>How do you define your experience with domain design?</b></p> <table> <tr> <td><b>Industrial:</b></td> <td><input type="checkbox"/> None.</td> <td><b>Academic:</b></td> <td><input type="checkbox"/> None.</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Low.</td> <td></td> <td><input type="checkbox"/> Low.</td> </tr> <tr> <td></td> <td><input type="checkbox"/> Medium.</td> <td></td> <td><input type="checkbox"/> Medium.</td> </tr> <tr> <td></td> <td><input type="checkbox"/> High.</td> <td></td> <td><input type="checkbox"/> High.</td> </tr> </table>	<b>Industrial:</b>	<input type="checkbox"/> None.	<b>Academic:</b>	<input type="checkbox"/> None.		<input type="checkbox"/> Low.		<input type="checkbox"/> Low.		<input type="checkbox"/> Medium.		<input type="checkbox"/> Medium.		<input type="checkbox"/> High.		<input type="checkbox"/> High.
<b>Industrial:</b>	<input type="checkbox"/> None.	<b>Academic:</b>	<input type="checkbox"/> None.													
	<input type="checkbox"/> Low.		<input type="checkbox"/> Low.													
	<input type="checkbox"/> Medium.		<input type="checkbox"/> Medium.													
	<input type="checkbox"/> High.		<input type="checkbox"/> High.													

**Table B.2** Questionnaire for Subjects Background (Part 2)

Questionnaire for Subjects Feedback
Did you have any difficulties to understand the inputs of the experiment? Which one(s)?
Did you have any difficulties to understand or apply the architectural element activity of the SOPLE-DE?
Did you have any difficulties to understand or apply the variability analysis activity of the SOPLE-DE?
Did you have any difficulties to understand or apply the architecture specification activity of the SOPLE-DE?
Did you have any difficulties to understand or apply the architectural elements specification activity of the SOPLE-DE?
Did you have any difficulties to understand or apply the design decisions documentation activity of the SOPLE-DE?
Do you think the SOPLE-DE training was efficacious? How do you classify it? [ ] Very Good. [ ] Good. [ ] Regular. [ ] Unsatisfactory.
Do you think the SOPLE-DE documentation is sufficient? Please justify.
Which improvements would you suggest for the SOPLE-DE?

**Table B.3** Questionnaire for Subjects Feedback