"EXTENDING THE RIPLE-DESIGN PROCESS WITH QUALITY ATTRIBUTE VARIABILITY REALIZATION"

Por

Ricardo de Oliveira Cavalcanti

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, AGOSTO/2010

Universidade Federal de Pernambuco
Centro de Informática

Ricardo de Oliveira Cavalcanti

"Extending the RiPLE-Design Process with Quality Attribute Variability Realization"

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

M.Sc. Dissertation presented to the Graduate Program in Computer Science of the Universidade Federal de Pernambuco in partial fulfillment of the requirements for the degree of M.Sc. in Computer Science

Advisor: Prof. Dr. Silvio Romero de Lemos Meira

Co-advisor: Prof. Dr. Eduardo Santana de Almeida

Recife, August/2010

Cavalcanti, Ricardo de Oliveira
Extending the riple-design process with quality attribute variability realization / Ricardo de Oliveira Cavalcanti - Recife: O Autor, 2010.
x, 136 folhas : il., fig., tab.

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

Inclui bibliografia e apêndice.

1. Engenharia de software. 2. Reúso de software. I. Título.

005.1 CDD (22. ed.) MEI2010 – 0167

Dissertação de Mestrado apresentada por **Ricardo de Oliveira Cavalcanti** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Enhanced RiPLE-Design for Quality Attribute Variability Realization**", orientada pelo **Prof. Silvio Romero de Lemos Meira** e aprovada pela Banca Examinadora formada pelos professores:

[illegible]

Prof. Carlos André Guimarães Ferraz
Centro de Informática / UFPE

[illegible]

Prof. Patrick Henrique da Silva Brito
Instituto de Computação / UFAL

[illegible]

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

Visto e permitida a impressão.
Recife, 30 de agosto de 2010.

[illegible]

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

To my parents, Ana and Toinho, my sisters, Natália and Nara

and my girlfriend, Emília

ACKNOWLEDGMENTS

Initially, I would like to thank the professors from Federal University of Pernambuco (UFPE) who provided me with the necessary educational support to get here.

I would like to thank Inove Informática for supporting my desire to obtain my master degree.

I would like to thank the Reuse in Software Engineering (RiSE) group for the support and opportunity to discuss my work. I am grateful to Flávio Medeiros and Vanilson Burégio, for their revisions, thoughts and comments. My eternal gratitude to the RiSE Labs members for their patience during the experiment, seminars, presentations and discussions about my work.

Last but certainly not least, I would like to thank my family, friends, and my girlfriend; especially, my parents that have always supported me.

RESUMO

Reúso de software é uma forma viável de obter ganhos de produtividade e melhoria no *time-to-market* tão desejados pelas empresas. O reúso não sistemático (*Ad hoc*) pode ser prejudicial, uma vez que a reutilização de artefatos de baixa qualidade pode diminuir a qualidade dos produtos finais. O reúso sistemático através da adoção de Linhas de Produto de Software (LPS) é uma boa alternativa para alcançar metas de qualidade e de redução de custos. Essa abordagem se tornou uma solução efetiva para gerar vantagem competitiva para as empresas.

Arquiteturas de linhas de produto devem se beneficiar das comunalidades entre os produtos e possibilitar a variabilidade entre eles. Ao mesmo tempo, como uma arquitetura de software, precisa atender requisitos de atributos de qualidade. O desafio de atender atributos de qualidade em sistemas únicos (*single systems*) torna-se ainda mais complicada no contexto de linhas de produto porque a variabilidade pode ocorrer também nos atributos de qualidade.

A variabilidade em atributos de qualidade é uma questão complexa. Entretanto, ela tem sido negligenciada ou ignorada pela maioria dos pesquisadores, uma vez que as atenções têm se mantido no alcance da variabilidade funcional. O foco deste trabalho é definir um processo para o design de arquiteturas de linhas de produto de software que possa lidar de forma eficaz com variabilidade em atributos de qualidade. O processo aprimora o RiPLE-Design com atividades e guias para o design com variabilidade de atributos de qualidade. Por fim, um estudo experimental é apresentado com o intuito de caracterizar e avaliar as melhorias propostas ao processo.

Palavras-Chave: Reúso de software, linhas de produto de software, arquitetura de software, variabilidade em atributos de qualidade.

ABSTRACT

Software reuse is a viable way to achieve the increase in productivity and short time to market desired by the companies. Ad hoc reuse may be harmful for companies, since the reuse of low quality assets can decrease the quality of their product. Systematic reuse through the adoption of Software Product Lines (SPL) is a good alternative to achieve the quality and time to market goals. Thus, it has become an effective solution for leading competitive advantage.

Product line architecture must benefit from commonalities among products in the family and enable the variability among them. At the same time, as any other software architecture, it must address quality attribute requirements. The challenge of achieving quality attributes in single-systems becomes even more complicated in a product line context because variability can occur also in quality attribute requirements.

The aspect of variability in quality attributes is a complex issue. Nevertheless, it has been neglected or ignored by most of the researchers as attention has been mainly put in functional variability. The focus of this dissertation is to provide architecture and design process for software product lines that can properly deal with quality attribute variability. The proposed approach enhances the RiPLE-Design process for software product line engineering with activities and guidelines for quality attribute variability. Finally, an initial experimental study is presented to characterize and evaluate the proposed process enhancements.

Keywords: Software Product Lines (SPL), Software Architecture, Software Reuse, Quality attribute variability

LIST OF FIGURES

Figure 1.1 RiSE Labs influencing areas 3
Figure 2.1 The key activities or software product line development (Clements; Northrop, 2001). 10
Figure 2.2 Costs for developing n kinds of systems as single systems compared to product line (Pohl et al., 2005) 12
Figure 2.3 Architecture Business Cycle (Bass et al., 2005) 14
Figure 3.1 RiPLE-Design Overview, in flowchart notation 37
Figure 3.2 Identify Architectural Drivers, in (Souza Filho, 2010), in flowchart notation 38
Figure 3.3 Module definition Activities, in (Souza Filho, 2010), in flowchart notation 40
Figure 3.4 Feature Model Diagram for Rise Chair 43
Figure 3.5 RiSE Chair Module View, in UML notation 45
Figure 3.6 Structural view of the Revision component, in UML notation 45
Figure 3.7 Sequence diagram: invite reviewer, in UML notation 46
Figure 3.8 Low demand availability scenario 47
Figure 3.9 High demand availability scenario 47
Figure 3.10 Deployment diagram. System under normal operation 49
Figure 3.11 Deployment diagram. Under faulty operation, proxy server as a request router .. 50
Figure 3.12 Performance scenario with short latency 51
Figure 3.13 Confidentiality scenario 51
Figure 3.14 Performance scenario with longer latency 52
Figure 3.15 Deployment diagram. Load Balancing 53
Figure 4.1 Feature Model for Meshing Tools, in (Rossel et al., 2009b) 59
Figure 4.2 Identify architectural drivers modified, in flowchart notation 61
Figure 4.3 Feature model with quality attribute variability 62
Figure 4.4 Latency quality attribute of a hypothetic product line 62
Figure 4.5 High latency quality scenario 63
Figure 4.6 Low latency quality scenario 63
Figure 4.7 Module definition task modified, in flowchart notation 65
Figure 4.8 Deployment diagram. General case. 67
Figure 4.9 Deployment diagram, secure link. 68
Figure 4.10 Evaluation steps in RiPLE-Design, in flowchart notation 73
Figure 4.11 Evaluate derived product architectures 82
Figure 5.1 Component Coupling 100
Figure 5.2 Component Coupling Average 101
Figure 5.3 Component Instability 101
Figure 5.4 Component instability average 102
Figure 5.5 Functional Requirements 103
Figure 5.6 Functional Variability 104
Figure 5.7 Quality Attributes 105
Figure 5.8 Quality Attributes Variability 106

LIST OF TABLES

Table 2-1 Dependability General Scenario, from (Bass et al., 2003) 20
Table 2-2 Modifiability General Scenario, from (Bass et al., 2003) 21
Table 2-3 Performance General Scenario, from (Bass et al., 2003) 22
Table 2-4 Security General Scenario, from (Bass et al., 2003) 24
Table 2-5 Testability General Scenario, from (Bass et al., 2003) 25
Table 2-6 Usability General Scenario, from (Bass et al., 2003) 26
Table 3-1 Modifiability scenario for the RiSE Chair SPL 43
Table 4-1 Decision guideline for optional availability 69
Table 4-2 Decision guideline for optional availability 70
Table 5-1 One Factor with two treatments design 94
Table 5-2 Subject's Profile 99
Table C-1 Questionnaire for Subject's Background (Part 1) 134
Table C-2 Questionnaire for Subject's Background (Part 2) 135
Table C-3 Questionnaire for Subject's Feedback 136

TABLE OF CONTENTS

1 Introduction 1

1.1 Motivation 2

1.2 Research objective 2

1.3 Overview of the proposed solution 3

1.3.1 Context 3

1.3.2 Outline of the Proposal 4

1.4 Out of Scope 5

1.5 Statement of the Contributions 6

1.6 Organization of the Dissertation 7

2 Software Product Lines: An Overview 8

2.1 Software Product Lines 9

2.1.1 Benefits 10

2.1.2 Software Product Lines Adoption Models 12

2.2 Software Architecture 13

2.2.1 Quality Attributes 16

2.2.2 Product Line Architecture (PLA) 26

2.3 Quality Attributes in Software Product Lines 31

2.4 Chapter summary 34

3 The RiPLE-Design Process 35

3.1 RiPLE-Design 35

3.2 Activities 37

3.2.1 Identify Architectural Drivers 37

3.2.2 Define Architectural Details 39

3.2.3 Represent Architecture 39

3.2.4 Identify Design Decisions 42

3.3 Design of variable quality attributes 42

3.3.1 Variations among different quality attributes 47

3.3.2 Different levels in quality attributes (trade-off variability) 50

3.3.3 Discussion 55

3.4 Chapter Summary 56

4 Extending the RiPLE-Design Process with Quality Attribute Variability Realization 57

4.1 Representing quality attribute variability 58

4.1.1 Represent variable quality attributes in the feature model 60

4.2 Design strategy for varying quality attributes 63

4.2.1 Select architectural drivers 65

4.2.2 Choose Architectural Styles 66
4.2.3 Document decision guidelines 69
4.3 Evaluate the architecture in order to achieve the needed variation 71
4.3.1 Evaluate existing architectures 73
4.3.2 Evaluate the DSSA 75
4.3.3 Evaluate derived product architectures 80
4.4 Chapter summary 82
5 The Experimental Study 83
5.1 Experimental terminology 84
5.2 The experimental study 85
5.2.1 Contextualization 85
5.2.2 Planning 90
5.2.3 Operation 98
5.2.4 Analysis and Interpretation 99
5.3 Conclusions 110
5.4 Lessons Learned 111
5.4.1 Training 111
5.4.2 Motivation of subjects 112
5.4.3 Number of subjects 112
5.5 Chapter Summary 112
6 Conclusion 114
6.1 Related Work 115
6.2 Future work 116
6.3 Academic Contributions 117
6.4 Concluding Remarks 117
References 119
Appendix A Architecture Document Template 129
Appendix B Architecture Evaluatin Report Template 133
Appendix C Instruments of the Experimental Study 134

1 INTRODUCTION

Software reuse is a viable way to achieve the increase in productivity and short time to market desired by companies (Bosch, 2001). Nevertheless, ad hoc reuse may be harmful for companies, since the reuse of low quality assets can decrease the quality of their product. On the other hand, systematic reuse through the adoption of Software Product Lines (SPL) can enhance quality and shorten time to market as shown in (Atkinson et al., 2002; Mili et al., 2001; Pohl et al., 2005). Thus, it has become an effective solution for leading competitive advantage.

Software architecture is a key discipline in SPL development (Clements; Northrop, 2001). Product line architecture must benefit from commonalities among products in the family and enable the variability among them. At the same time, as any other software architecture, it must address quality attribute requirements, externally visible properties not related to the functional capabilities of the system. The challenge of achieving quality attributes in single-systems becomes even more complicated in a product line context because there is variability on quality attribute requirements and different quality constraints are required. The focus of this dissertation is to provide a design process for software product line architectures that can properly deal with quality attribute variability.

This chapter contextualizes the focus and describes the 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

According to (Kolb et al., 2004), “research in the field of software product lines has primarily focused on analysis, design, and implementation to date and only very few results address the quality assurance problems and challenges that arise in a reuse context”. Those challenges relate also to the achievement of quality attributes requirements in a SPL context, where the quality attributes might be required by every product in the product line or only by specific products. Thus, variability can also occur in quality attributes.

It is worth to remark that quality attributes affect each other, often they impact negatively (Barbacci et al., 1995). These trade-off situations are typically resolved by finding a halfway between conflicting quality attributes. Trade-off analysis of quality attributes in SPL is also more difficult than in single-systems due to this variability and the exponential number of possibilities, as mentioned in (Etxeberria et al., 2008).

In particular, the aspect of variability in quality attributes is also a complex issue that has been “neglected or ignored by most of the researchers as attention has been mainly put in the variability to ensure that it is possible to get all the functionality of the products”, as discussed in (Etxeberria et al., 2008).

Based on the definitions of (Niemelä; Immonen, 2007), quality attribute variability can happen in three different situations: (i) *variation among different quality attributes*; (ii) *different levels in quality attributes*; and (iii) *functional variability may indirect cause variation in qualities*, and vice versa.

Therefore, to proper develop a product line, quality attribute and their variability must be gathered and managed throughout the development life cycle.

1.2 RESEARCH OBJECTIVE

Encouraged by the motivations depicted in the previous section, namely the complexity related to the task of managing quality attributes variability in software product lines, the benefits of managing this variability, and particularly, the lack of research regarding quality attribute variability, as discussed in (Etxeberria et al., 2008), the goal of this dissertation can be stated as follows:

"This work investigates the issues related to the handling of quality attribute variability in software product line architectures. After augmenting the deficiencies of an existing product line architecture process, it provides enhancements by modifying existing activities and tasks, as well as proposing new ones. Moreover, the proposed enhancements are based on a set of software product line, component-based development, and design principles".

1.3 OVERVIEW OF THE PROPOSED SOLUTION

1.3.1 Context

This work is developed in the context of the Reuse in Software Engineering (RiSE)[1] Labs, whose goal is to develop a robust framework for software reuse with the purpose of facilitating the adoption of a reuse program (Almeida et al., 2004). The RiSE Labs framework is influenced by several forces as depicted in Figure 1.1.

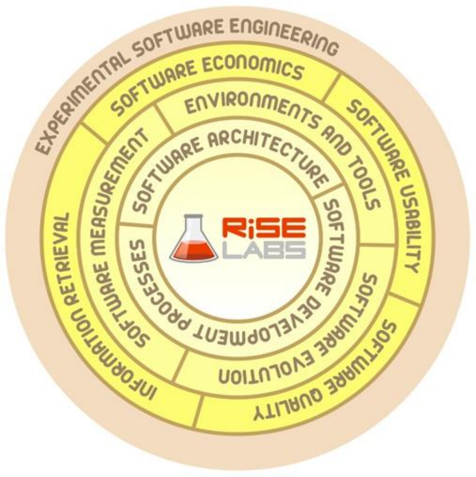

Figure 1.1 RiSE Labs influencing areas

Several sub-projects emerged under the influences of the RiSE Labs framework. The framework embraces several different areas related to software productivity and mainly software reuse. Those areas are the further studies in projects:

- **RiSE Framework** project is focused on processes for software reuse (Almeida et al., 2004; Nascimento, 2008), component certification (Alvaro et al., 2006),

[1] http://www.rise.com.br/research

and reuse adoption processes (Garcia et al., 2008);

- **RiSE Tools** projects are focused on the development of 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** project develops methodology for software product line engineering, which is divided in several disciplines, such as evolution (Oliveira, 2009), tests (Machado, 2010; Silveira Neto, 2010) , requirements (Neiva, 2009), and design (Souza Filho et al., 2009). This dissertation is part of this project, and it is concerned with the enhancement of the design process for software product line architectures;
- **SOPLE** project develops methodology for service-oriented product lines, which is also divided into disciplines, such as architecture and design (Medeiros, 2010);
- **MATRIX** project investigates the area of measurement in reuse and its impact on quality and productivity;
- **BTT** research is focused on methods and tools for detection of duplicate bug reports, as in (Cavalcanti, 2009); and
- **Exploratory Research** investigates new research directions in software engineering and its impact on reuse.

1.3.2 Outline of the Proposal

Developed under the RiSE Labs, the RiPLE-Design process (Souza Filho, 2010) attempts to develop a Domain specific Software Architectures (DSSA) based on quality attribute requirements prioritization. The process lacks guidelines to deal with quality attributes variability.

The proposed solution enhances the existing RiPLE-Design process by modifying existing steps and introducing new ones in order to make the process able do deal properly with quality attribute variability.

The improvements act in the three pillars that, as proposed by (Myllärniemi; Männistö; et al., 2006), are essential to achieve quality attribute variability: (i) specification and modeling varying quality attributes; (ii) design strategies for varying quality attributes; and (iii) evaluation techniques in order to achieve the needed variation.

The proposed improvements maintain the essence of the original RiPLE process, which is its quality driven aspect. The original roles participating in the process are maintained, as well as the inputs and outputs, although the latter have been slightly modified. The detailed description of the RiPLE-Design approach is presented in Chapter 4.

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 aims to enhance an existing design method in relation to its capacity to deal with quality attribute variability. Thus, the following issues are not directly addressed by this work:

- *Other development disciplines*: following the definition of the RiPLE-Design process, other development disciplines will not be described in this work. Nonetheless, other disciplines, e.g., evolution (Oliveira, 2009), requirements (Neiva, 2009) and testing (Machado, 2010; Silveira Neto, 2010), were already envisioned in other works by the RiSE Labs;
- *Architecture reconstruction*: architecture reconstruction is the way the "as-built" architecture of an implemented system is obtained from an existing system (Bass et al., 2003). These techniques will not be covered in this work;
- *Feature Interaction Problems*: The case of functional variability affecting quality attributes is related to the occurrence of feature interaction in software product lines, as pointed out in (Lee; Kang, 2004). The problem of feature interaction can impact the whole SPL development process, as it promotes changes in reusable assets and impacts maintenance costs and other products. The case in which functional

requirements affects quality attributes, and promotes quality attributes variability, will then be left out of the scope of this work. Other cases of feature interaction will be covered by the proposed approach, namely, the case of quality attribute affecting each other; and the case of domain quality attributes required by certain functional requirements;

- *Product Development*: An important issue in a SPL process is to create individual products by reusing the core asset, i.e. products development *with* reuse. However, this aspect can be as complex as *core assets development*, involving the definition of activities, sub-activities, inputs, outputs, and roles. This work focus on the core asset development phase of a product line development effort.

- *Dynamic Product Lines*: a recent trend in SPL research that aims to develop strategies to deal with product configuration at runtime, as proposed by (Kim et al., 2007). This work will focus on traditional software product line approaches and in the handling of quality attribute variability during the architecture development. Although a traditional product line architecture development approach may lead to a solution in which quality attribute variability is addressed dynamically, this is not the focus of this work.

1.5 STATEMENT OF THE CONTRIBUTIONS

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

- The identification of limitations of a traditional process for designing product line architectures from quality attributes;

- Enhancements to the existing RiPLE-Design process for product line architecture, in order to make it able to properly deal with quality attribute variability;

- The definition, planning, analysis of an experimental study in order to evaluate the proposed process.

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. It also presents an overview on software architecture, architecture quality attributes and their relation with product line architectures;
- Chapter 3 presents the RiPLE-Design and augments its deficiencies concerning the treatment of quality attribute variability;
- Chapter 4 describes enhancements to the existing RiPLE-Design process in order to properly handle quality attribute variability;
- Chapter 5 describes the definition, planning, operation, analysis and interpretation of an experimental study for the proposed approach performed with the intention of characterizing and refining it;
- Chapter 6 presents some concluding remarks about this work, its related work, and directions for future work.
- Appendix A presents an Architecture Document Template to aid the adoption of the Extended RiPLE-Design approach.
- Appendix B presents an Architecture Evaluation Report Template to aid the adoption of the Extended RiPLE-Design approach.
- Appendix C describes the instruments used during the performed experimental study.

2
SOFTWARE PRODUCT LINES: AN OVERVIEW

Since 1968, when Douglas McIlroy published his *Mass Produced Software Components* paper (McIlroy, 1968), and coined the term, *software reuse* started to be seen as a mean to triumph over the software crisis, where software producers must deal with growing demand for more complex and reliable software systems.

For (Krueger, 1992), "*Software reuse is the process of creating software systems from existing software rather than building them from scratch*". The use of existing software leads to reduction of development effort and enables a producer to deliver software in less time, since the reused software piece will not be developed again. In addition, when reused components are mature and bear high quality, the resulting product is also expected to display high quality.

Nevertheless, those benefits cannot be achieved easily. (Sametinger, 1997) shows that managerial commitment and appropriate organizational structure are among the main key factors on adopting software reuse as a way to reach less effort on producing software. (Sametinger, 1997) also points out the lack of explicit procedures as a great obstacle. Along these lines, we can infer that reuse can become profitable if the company adopts it in a systematic, planed way. A known systematic approach to achieve systematic reuse is Software Product Lines.

The Software Engineering Institute (SEI) defines Software Product Lines (SPL) as *a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* (Clements; Northrop, 2001).

The remainder of this chapter will discuss the main activities of the SPL approach and the benefits of its adoption, in Section 2.1; as well as the software architecture activity and the importance of quality attributes, in Section 2.2. In Section 2.3, the specificities regarding quality attribute in the context of software product line architectures will be discussed. Section 2.4 presents the chapter summary.

2.1 SOFTWARE PRODUCT LINES

In (Dijkstra, 1972) and (Parnas, 1976) the notion of Software Families was introduced. Parnas points out that it is worth considering the development of a set of software that share common characteristics as a collective design. He mentions the importance of planning before developing a program family. He also gives importance to the order in which the design decisions are made, and suggests that an approach for software families should choose the degree of importance of each aspect and characteristic so that the resulting program addresses its purposes properly.

The need for SPLs recalls, in some sense, the product lines from the manufacturing world. The common platform sharing high levels of commonality, yet enabling differences among products are the key similar point.

(Pohl et al., 2005) define software product lines engineering as “a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization”. The use of platforms are related to the development of a software system that share high levels of commonality, which lead to lower development costs in the long run. According to (Pohl et al., 2005), “developing applications using platforms means to plan proactively for reuse, to build reusable parts, and to reuse what has been built for reuse”. Mass customization consists of producing products that are tailored to individual user needs. In the context of software-intensive systems it means “employing the concept of managed variability, i.e. the commonalities and the differences in the applications (in terms of requirements, architecture, components, and test artifacts) of the product line have to be modeled in a common way.” (Pohl et al., 2005).

There are three essential activities in SPL development, as shown in

Figure 2.1: core asset development, product development, and management.

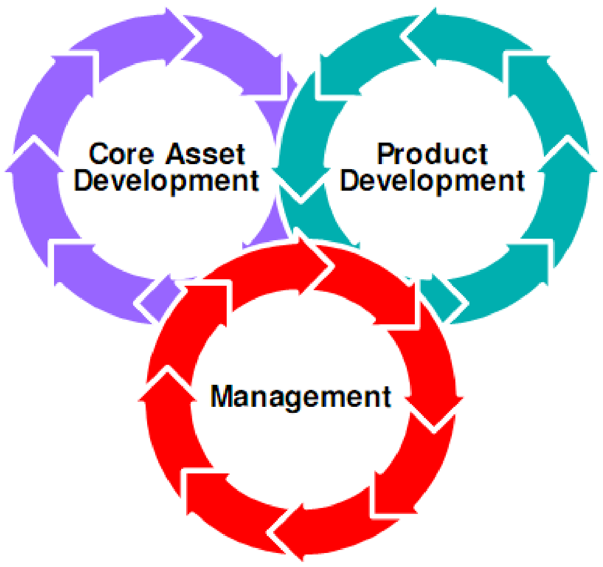

Figure 2.1 The key activities or software product line development (Clements; Northrop, 2001)**.**

The goal of the core asset development activity, also called domain engineering, is to establish the creation of common assets and the evolution of the assets in response to product feedback and new market needs, that is, the product line scope and the production plan. Based on these three artifacts, the product development activity, also known as application engineering, focusing on creating of individual products by reusing common assets, providing feedback to core asset development and products evolution.

The management activity includes technical and organizational management. Technical management supervises the core asset development and product development activities, ensures that the other activities are following the processes defined for the product line. It also decides on the production method and provides the project management elements of the production plan (Clements; Northrop, 2001). Organizational management coordinates the technical activities in and iterations between the critical activities of core asset development and product development.

2.1.1 Benefits

Systematic reuse through the adoption of Software Product Lines (SPL) can enhance quality and shorten time to market as shown in (Atkinson et al., 2002; Mili et al., 2001; Pohl et al., 2005). It can be seen as an effective solution for leading competitive advantage.

Some benefits from the adoption of SPL are discussed in (Pohl et al., 2005):

- **Quality Improvement.** Reusable assets have their quality attested in many opportunities, in different contexts, leading to a higher product quality.
- **Reduction of Development Costs**. Figure 2.2 compares the costs of producing several single systems to the costs of producing them using a SPL approach. Higher upfront investments are normally needed to produce core assets, as the reusable assets are more complex than specific ones. On the long run, the costs of producing new products in mature SPL should be very low. (Clements; Northrop, 2001) shows that the *break-even point*, when the costs are the same for developing the systems separately as for developing them by product line engineering, is achieved around three systems. Aspects like customer base, the expertise, the range and kinds of products and the SPL adoption strategy influence the exact location of the break-even point.
- **Reduction of Time to market**. Although initially higher, the time to market is significantly shortened as numerous artifacts can be reused in new products.
- **Reduction of Maintenance Effort**. Changes in reusable assets are propagated to all products in the line that use these assets. Such propagation may be exploited to reduce maintenance effort, even though testing each product can still be unavoidable.
- **Benefits for the Customers**. Customers get higher quality products for lower prices. In addition, as the range of products broadens, they get products adapted to their needs and wishes. Similar user interfaces can also help the customer to switch from one product to another in the same SPL.

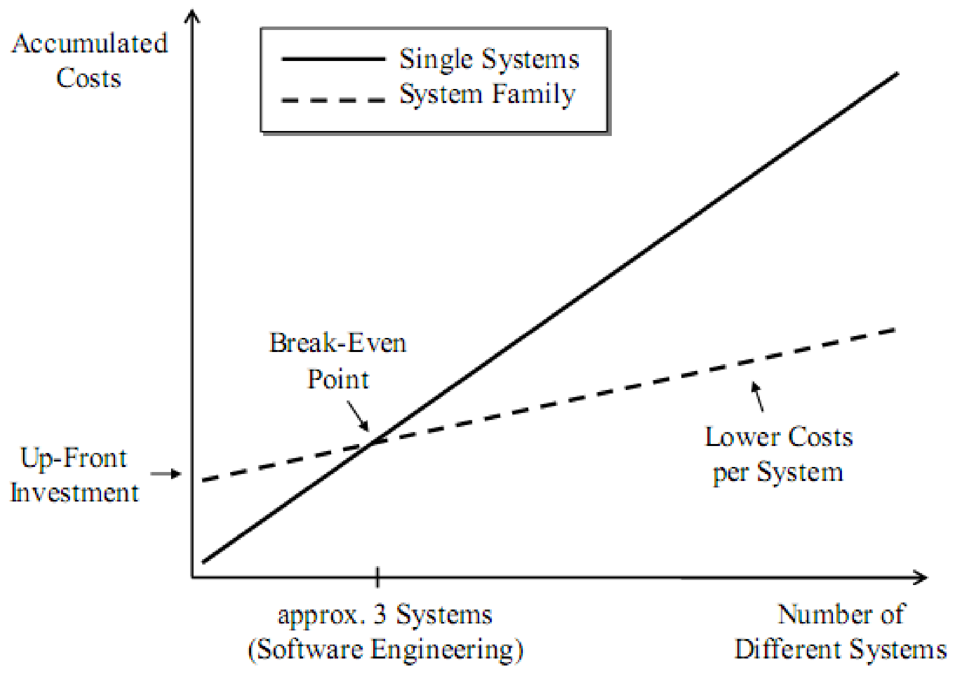

Figure 2.2 Costs for developing n kinds of systems as single systems compared to product line (Pohl et al., 2005)

2.1.2 Software Product Lines Adoption Models

The introduction of a software product line engineering approach is usually motivated by economic considerations. The economic pressure originates from the drive to get the new products to the market faster to stay competitive or produce them more efficiently (Pohl et al., 2005). Software product line adoption can help to solve both issues: to decrease development costs and reduce time-to market of products (van Der Linden et al., 2007).

An organization can adopt product line engineering using some adoption models. These models are not mutually exclusive, and should be chosen depending on its objectives, budget, time and requirements as described next (Krueger, 2002):

The *proactive* model corresponds to a heavyweight adoption approach. In order to support the full scope of products needed on the foreseeable horizon, the organization analyzes, designs and implements a complete software product line. It fits organizations that can predict their product line requirements well into the future, and have the time and resources for a long development cycle;

With the *reactive* approach, the organization incrementally grows their software product line when the demand arises for new products or new requirements on existing products. It is appropriate when the requirements for new products in the production line are somewhat unpredictable. This incremental approach offers a less expensive and quicker transition into software product lines, since only a minimum number of products must be incorporated in advance.

The *extractive* adoption model reuses existing products as the initial baseline for the product line. It is most appropriate when the collection of systems has a significant amount of commonality and also consistent differences among them.

The adoption of software product line engineering requires upfront investment, brings implications for the development process, and may also require modifications on the organizational structure (van Der Linden et al., 2007). 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 are all essential prerequisites for the adoption (Pohl et al., 2005). It is indispensable that each organization analyze its own budget and objectives before selecting a proper adoption model.

2.2 SOFTWARE ARCHITECTURE

Parnas, in (Parnas; Siewiorek, 1975), steps aside from software correctness and formal proof of programs and discusses whether a program that outputs correct is useful if we cannot rely on it when we demand. He presents the notion of software reliability as “a measure of the extent to which the system can be expected to deliver usable services when those services are demanded”. Along with Dijsktra, Parnas introduces the concern about the structure of a software system, the interfaces between modules as well as the communication among them. Those concerns were further defined as software architecture.

The Carnegie Mellon’s Software Engineering Institute defines software architecture as (Bass et al., 2003):

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software

elements, the externally visible properties of those elements, and the relationships among them.

Some interesting points come with the definition shown above. First, that an architecture defines elements, called structures. Second, the definition clarifies that systems can and do comprise more than one structure. Third, it implies that every software system has an architecture. Fourth, the behavior of each element is part of the architecture. Finally, the definition is indifferent as to whether the architecture for a system is a good one or a bad one. (Bass et al., 2003).

Software architecture is a result of technical, business and social influences. This cycle of influences is defined as the *Architecture Business Cycle* (ABC) in (Bass et al., 2003), and further revisited in (Bass et al., 2005). The revisited form is represented in Figure 2.3.

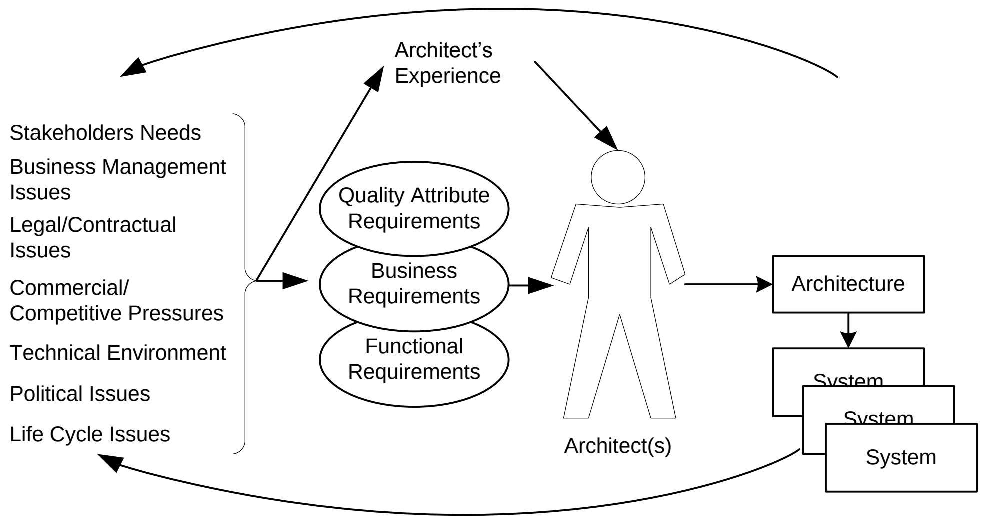

Figure 2.3 Architecture Business Cycle (Bass et al., 2005)

End users, developers, project manager, maintainers and even sellers influence the architecture. The *stakeholders* have concerns that they wish the end system to guarantee. While end users wish for more usability, maintainers want the system to be easier modifiable, customers want and low costs. All those concerns and goals sometimes are contradictory, and it is the architect's role to mediate the conflicts and resolve trade-offs.

Architecture is also influenced by the nature and structure of the *development organization*. For example, depending on the set of skills the team of employed developers has, a specific architectural approach may be chosen. Both long-term and immediate business decisions may also influence, such as the purchase of some development toolkit, market trend, commercial agreement or strategic decision.

Experience is a great factor of influence in the resulting architecture. Successful previous approaches are likely to be tried again on new development efforts. Education and training of an architect such as his exposure to successful architectural patterns or, conversely, to systems that worked poorly are also good sources of possibilities. Standard industry practices, prevalent engineering techniques in the architect's professional community forming a *technical environment* are also reflected in the resulting system architecture.

The ABC depicts seven forces that influence the architectural constructions, and those forces are grouped in three overlapping sets, namely Quality Attribute Requirements, Business Requirements and Functional Requirements. Those three sets, along with the architect's experience, serve as inputs to architectural construction and further system development.

The ABC also represents that the architecture affects the factors that influence them, as described in (Bass et al., 2003). Firstly, a prescribed system structure, dictates the units of software that must be implemented and, furtherer, serve as basis for development's project structure and team formation. Secondly, successful systems can enable some company to establish itself in a specific market niche as the architecture can provide opportunities for further production and deployment of similar products. Third, the knowledge gained during the development of a system adds to the corporate knowledge base. Fourth, some system may actually change the software engineering culture and establish new best practices and standards. Application frameworks like Ruby on Rails and the so-called "NoSQL" database Cassandra (Apache Foundation, 2010) are such examples.

From a technical perspective, there are three reasons for software architecture's importance, as explained in (Bass et al., 2003):

(i) It represents a common abstraction of a system and serves as a common language in which different concerns can be expressed, discussed and resolved among different stakeholders, even for complex systems.

(ii) It represent a system's earliest design decisions, which are the hardest to change later in the development process, it also defines constraints on implementation and organizational structure. The design decisions also inhibits or enable the achievements of quality attributes, as it will be explained furtherer.

(iii) The architectural representation of a system is a relative small, understandable model for how a system is structured and how its elements work together. This model is transferable across systems and can be applied to other systems having similar quality attributes and functional requirements.

2.2.1 Quality Attributes

The externally visible properties described in the software architecture definition are attributes not related to functional capabilities of the system. Those properties are the quality attributes which a software system has, for example, the ability to start-up in less than 10 seconds; or the possibility to modify some module and ship a new product within a week. Modifiability, performance, security and usability are examples of *system quality attributes* and will be described further.

Functionality is the ability of a system do the work for which is was intended. It may not be possible to have functionality and quality attributes as orthogonal concerns sometimes, for example, manipulating complex multimedia content will probably make very low response time impossible. On the other hand, functionality may be achieved through the use of any of a number of possible structures; it is the purpose of software architecture discipline to constrain the allocation of functionality to such a structure where other so-called non-functional requirements can also be achieved.

The main objective of the software architecture discipline is to evolve the organization of modules of a software system in a way that the functionalities can perform gracefully, that is accomplishing the desired quality attribute requirements. As well put in (Bass et al., 2003), “systems are frequently redesigned not because they are functionally deficient – the replacements are often functionally identical – but because they are difficult to maintain, port, or scale, or are too slow, or have been compromised by network hackers.”

The achievement of system quality attributes is to be considered throughout design, implementation and deployment, because no quality attribute is entirely dependent on design, nor entirely dependent on implementation or deployment, as discussed in (Bass et al., 2003).

Usability, for example, include implementation aspects of developing clear and easy to use interfaces, but also include architectural aspects such as providing capabilities to cancel and undo operations or to re-use data previously entered. In a similar way, the performance quality attribute is affected by communication among components, and the functionality allocated to each of the components, and these are architectural concerns; the choice of algorithms for selected functionality is a matter of implementation.

It is also important to remark that within complex systems, quality attributes can never be achieved in isolation, and that those systems often fail to meet quality attribute requirements “when designers narrowly focus on meeting some requirements without considering the impact on other requirements or by taking them into account too late in the development process”, as discussed in (Barbacci et al., 1995). Security and reliability can illustrate the trade-off that an architect must face and solve when designing a complex system: secure systems have must have the fewest points of failure, while reliable systems need redundant processes where the failure of any one will not cause the system to fail. The issue of tension among quality attributes is not new, as illustrated by Boehm:

> Finally, we concluded that calculating and understanding the value of a single overall metric for software quality may be more trouble than it is worth. The major problem is that many of the individual characteristics of quality are in conflict; added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability; added accuracy often conflicts with portability via dependence on word size; conciseness and conflict with legibility. Users generally find it difficult to quantify their preferences in such conflict situations. (Boehm, 1978)

There are a variety of published taxonomies and definitions for quality attributes. Many of them have their own research and practitioner communities. Three problems related to system quality attributes are pointed out by (Bass et al., 2003):

- Definitions of an attribute are often not operational. For example, it is meaningless to say that a system will be modifiable without specifying to which set of changes it should be modifiable.
- Attribute definition and its implications are often overlapping. For example, a system failure can be an aspect of availability, of security or even of usability.

- The vocabulary around each attribute varies greatly. An occurrence can be described as an “event”, an “attack”, a “failure”, or simple “user input” in the context of performance, security, availability or usability, respectively.

Quality attributes scenarios are used to represent and analyze quality attributes. They express in a concise way some important aspects of the quality attributes requirement and its concretization and perception in the system under development. Allied with a brief discussion of each attribute and its meaning to the involved stakeholders, quality scenarios solve the abovementioned problems by characterizing quality attributes in six parts.

- *Source of stimulus*. The stimulus generator, such as a user or another computer system.
- *Stimulus*. The condition to be considered when it arrives at a system, e.g., an unanticipated message in an availability scenario.
- *Environment*. The conditions within the stimulus occur.
- *Artifact*. The stimulated artifact, such as the system, or some part of it.
- *Response*. The undertaken activity in response to the arrival of the stimulus.
- R*esponse measure*. The measure of the response, so that the quality attribute requirement can be tested.

Generic quality attribute scenarios will be used next to describe some quality attributes, namely: dependability, modifiability, performance, security, testability and usability. When scenarios are used during system design activities, concrete scenarios should be generated, replacing the generic possibilities that will be shown next with real system expectations, measures and stimuli.

Besides system quality attributes, there are also other classes of non-functional qualities that influence software architecture, namely, *business qualities* and *architectural qualities* (Bass et al., 2003). Business qualities normally center on cost, schedule, market, and marketing considerations. Examples of such qualities are *time to market*, *cost and benefit*, *projected lifetime of the system*, *targeted market*, *rollout schedule and integration with legacy system*.

Architectural qualities refer to the architecture itself in a broader way than system quality attributes and are very important, although difficult to measure. *Conceptual integrity* is the underlying theme or vision that unifies the design of the system at all levels, from the appearance to the user, to the architectural layout. *Buildability* is another architectural quality that the allows the system to be completed in a timely manner by the available team and to be open to certain changes as development progresses.

(International Organization for Standardization/International Electrotechnical Commission, 2001) classifies software quality in a structured set of characteristics and sub-characteristic. Those definitions are similar to quality attributes. This work does not intend to encompass all characteristics and sub-characteristics from the ISO/IEC 9126 standard. Nonetheless, as it will focus on the system quality attribute definition from (Bass et al., 2003), the software quality definition from the aforementioned standard seem to be covered.

The focus of this work is on system quality attributes, although scenarios and evaluation techniques can also be used to assess business and other architectural qualities.

2.2.1.1 Dependability

Laprie, in (Laprie, 1992), defines *Dependability* as “that property of a computer system such that reliance can justifiably be placed on the service it delivers”. It is concerned with system failure and its associated consequences. *Availability* and *reliability* are two of the most important sub-attributes of dependability. Other sub-attributes are *safety*, *confidentiality* and *integrity*, that will be discussed under the security quality attribute; and *maintainability*, that will be discussed next, in the modifiability subsection.

According to (Barbacci et al., 1995), “the availability of a system is a measure of its readiness for usage.” It is measured as the limit of the probability that a system will be operational when it is needed. It is typically defined as:

$$\alpha = \frac{\text{mean time to failure}}{\text{mean time to failure} + \text{mean time to repair}}$$

For (Barbacci et al., 1995), the reliability of a system is a measure of the ability of a system to keep operating over time. It is measured as the system’s mean time to failure that is the expected life of the system. Availability measures the readiness of a system only when it is needed, while reliability measures its readiness along the time, regardless the need for the system.

The main impairments of dependability are *faults* and *failures*. These two must be differentiated: failures are observable by the system's user and faults are not. Therefore, a fault may become a failure if not corrected or masked. A fault can occur by one of the following reasons: *omission*, when a component fails to respond to an input; *crash*, when the component repeatedly suffers omission faults; *timing*, when a component responds, but it is too early or too late; *response*, when the response is an incorrect value.

The characterization of dependability involves how the system detects some failure, how frequent a failure may occur, the response to that occurrence and how long a system is allowed to be out of operation. Prevention of failures, the safety of its occurrence and what notifications are required when a failure occurs are also described in availability and reliability scenarios. Table 2-1 summarizes the portions of a Dependability scenario.

Portion of Scenario	*Possible Values*
Source	Internal to the system; external to the system
Stimulus	Fault: omission, crash, timing, response
Artifact	System's processors, communication channels, persistent storage, processes
Environment	Normal mode; reduced capacity (i.e., fewer features, a fall back solution)
Response	System should detect event and do one or more of the following: record it; notify appropriate parties, including the user and other systems; disable sources of events that cause fault or failure according to defined rules; be unavailable for a prespecified interval, where interval depends on criticality of system; continue to operate in normal or degraded mode
Response Measure	Time interval when the system must be available Availability time Time interval in which system can be in degraded mode Repair time

Table 2-1 Dependability General Scenario, from (Bass et al., 2003)

2.2.1.2 Modifiability

Modifiability relates to "the cost of change and refers to the ease with which a software system can accommodate changes" (Northrop, 2004). It brings up four concerns: (i) Who makes the change? (ii) When is the change made? (iii) What can change? and (iv) How is the cost of change measured?

From a scenario perspective, “who makes the change” is the source of the stimulus. Normally, scenarios will refer to changes made in source code, so developers will be the source. Architects and system administrators can also be the source as they can also make changes that will change aspects of the system. Following these thoughts, even an end user can be the agent of change.

Regarding the time when a change is made, it can be *during design*, by detailing architectural design; can refer to the *implementation*, by modifying the source code; can be *during compile*, using compile-time switches; *during build*, by choice of libraries; *during configuration setup*, by including parameter setting, for example; or *during execution*, by parameter setting.

A change can occur in any part of the system, such as the functions is operates, the platform the system exists on, the environment within the system operates, the qualities the system exhibits and its capacity. (International Organization for Standardization/International Electrotechnical Commission, 2001) defines two categories for change: *maintainability* and *portability*. Maintainability is the capability of the software product to be modified and portability is the capability of the software product to be transferred from one environment to another. Concerning these subdivisions, they are both considered here as forms of modifiability.

Modifiability can be measured by Cost in terms of number of elements affected, effort, money and the extent to which this affects other functions or quality attributes. Table 2-2 summarizes the Modifiability scenario.

Portion of Scenario	*Possible Values*
Source	End user, developer, system administrator
Stimulus	Wishes to add/delete/modify/vary functionality, quality attribute, capacity
Artifact	System user interface, platform, environment; system that interoperates with target system
Environment	At runtime, compile time, build time, design time
Response	Locates places in architecture to be modified; makes modification without affecting other functionality; test or deploy modification
Response Measure	Cost in terms of number of elements affected, effort, money; extent to which this affects other functions or quality attributes

Table 2-2 Modifiability General Scenario, from (Bass et al., 2003)

2.2.1.3 Performance

According to (Barbacci et al., 1995), "performance is that attribute of a computer system that characterizes the timeliness of the service delivered by the system." It refers to either the time required to respond to specific events or the number of events processed in a given interval of time. This means not only to be fast, but to meet timing constraints. In real time systems, it means to be predictable and meet overall average-case or worst-case predictions.

The characterization of performance scenario starts by the definition of the event sources and arrival patterns and describes how the system must allocate resources in order to respond to the request timely. An arrival pattern for events may be either *periodic* or *stochastic*. A periodic event may arrive, for example, every 50 milliseconds. This kind of arrival pattern is most often seen in real-time systems. Stochastic arrival means that events arrive according to some probabilistic distribution. Events can also arrive *sporadically*, that is, according to a pattern not classified as either periodic or stochastic.

There are four main concerns involving the performance quality attribute: *latency*, i.e., the time between the arrival of the stimulus and the systems response, and its variation, named jitter; *throughput*, the number of transactions the system can process in a given interval of time; *capacity*, or, how much demand can be placed on the system while continuing to meet latency and throughput requirements; and *modes*, i.e., characterizing how the system should behave if its capacity exceeds, the number of events not processed, and data lost because the system was too busy, whether it should function in *reduced capacity* or in *overload* mode, sacrificing timing requirements. These concerns are characterized by the response of the systems for a given stimulus. Table 2-3 summarizes a performance general scenario.

Portion of Scenario	*Possible Values*
Source	One of many independent sources, possibly from within system
Stimulus	Periodic events arrive; sporadic events arrive; stochastic events arrive
Artifact	System or specific subsystem
Environment	Normal mode; overload mode; reduced capacity
Response	Processes stimuli; changes level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate, data loss

Table 2-3 Performance General Scenario, from (Bass et al., 2003)

2.2.1.4 Security

The USA's National Research Council, in (System Security Study Committee et al., 1991) defines security as:

> […]
> 2. Computer security is protection of data in a system against disclosure, modification, or destruction. Protection of computer systems themselves. Safeguards can be both technical and administrative.
> 3. The property that a particular security policy is enforced, with some degree of assurance
> […]

From an architectural point of view, a system that enables security must provide several capabilities: nonrepudiation, confidentiality, integrity, assurance, availability, and auditing. A definition for each of these terms is provided next, according to (Bass et al., 2003).

1. *Nonrepudiation* is the property that a transaction cannot be denied by any of the parties to it. It functions as one had a signature or stamp in a receipt, and cannot deny he did the buy.

2. *Confidentiality* is the property that data or services are protected from unauthorized access. An example of its violation is a hacker that accesses some confidential data by sniffing the network.

3. *Integrity* is the property that data or services are being delivered as intended. Database transactions normally guarantee that data is recorded without modification, allowing integrity of the data.

4. *Assurance* is the property that the parties to a transaction are who they purport to be. In real world transactions, identification cards guarantee this property.

5. *Availability* is the property that the system will be available for legitimate use. Under a security point of view, an availability quality attribute requirement can be violated through denial-of-service attacks, for example.

6. *Auditing* is the property that the system tracks activities within it at levels sufficient to reconstruct them.

Each of these security capabilities can be characterized by scenarios, as summarized in Table 2-4.

Portion of Scenario	*Possible Values*
Source	Individual or system that is: correctly identified, identified incorrectly, of unknown identity who is: internal/external, authorized/not authorized with access to: limited resources, vast resources
Stimulus	Tries to: display data, change/delete data, access system services, reduce availability to system services
Artifact	System services; data within system
Environment	Either: online or offline, connected or disconnected, firewalled or open
Response	Authenticates user; hides identity of the user; blocks access to data and/or services; allows access to data and/or services; grants or withdraws permission to access data and/or services; records access/modifications or attempts to access/modify data/services by identity; stores data in an unreadable format; recognizes an unexplainable high demand for services, and informs a user or another system, and restricts availability of services
Response Measure	Time/effort/resources required to circumvent security measures with probability of success; probability of detecting attack; probability of identifying individual responsible for attack or access/modification of data and/or services; percentage of services still available under denial-of-services attack; restore data/services; extent to which data/services damaged and/or legitimate access denied

Table 2-4 Security General Scenario, from (Bass et al., 2003)

2.2.1.5 Testability

As stated by (Voas; Miller, 1995), "*Testability* suggests the testing intensity, and provides the degree of difficulty which will be incurred during testing of a particular location to detect a fault". As a large part of the cost of developing system is taken up by testing, the payoff can be large if the software architect can reduce this cost.

The testability is measured by the probability that a system will fail on its next test execution, assuming that it has at least one fault. As this measure is very complicated to calculate, other measures are normally used. (Bass et al., 2003) suggest that "the response measures for testability deal with how effective the tests are in discovering faults and how long it takes to perform the tests to some desired level of coverage."

From an architectural point of view, testability can be improved when it is possible to control each component's internal state and inputs and then observe it outputs. Playback capabilities and information hiding are good ways to promote testability. The testability general scenario is summarized in Table 2-5.

Portion of Scenario	*Possible Values*
Source	Unit developer Increment integrator System verifier Client acceptance tester System user
Stimulus	A milestone in the development process: Analysis, architecture, design, class, subsystem integration completed; system delivered
Artifact	Piece of design, piece of code, complete application
Environment	At design time, at development time, at compile time, at deployment time
Response	Provides access to state values; provides computed values; prepares test environment
Response Measure	Percent executable statements executed Probability of failure if fault exists Time to perform tests Length of longest dependency chain in a test Length of time to prepare test environment

Table 2-5 Testability General Scenario, from (Bass et al., 2003)

2.2.1.6 Usability

According to (IEEE Computer Society, 1990), *Usability* is "the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component." This definition can be split into the following areas, according to (Bass et al., 2003): (i) learning system features; (ii) using a system efficiently; (iii) minimizing the impact of errors; (iv) adapting the system to user needs; (v) increasing confidence and satisfaction. Works have been done to show the relation between usability and software architecture, (Bass; John, 2000; Bass; John, 2003), and describe how to address usability issues, early in system development, introducing system's capabilities that will help the user in each of the aforementioned categories.

In Usability scenarios, will normally have the end user as the source of stimulus and the system as artifact. The expected response falls in one of the usability categories, and is measured by the incidence of errors, elapsed time to perform some task, user satisfaction, and so on. Table 2-6 summarizes the usability general scenario.

Portion of Scenario	*Possible Values*
Source	End user
Stimulus	Wants to learn system features; use system efficiently; minimize impact of errors; adapt system; feel comfortable
Artifact	System
Environment	At runtime or configure time
Response	System provides one or more of the following responses: to support "learn system features": help system is sensitive to context; interface is familiar to user; interface is usable in an unfamiliar context to support "use system efficiently": aggregation of data and/or commands; re-use of already entered data and/or commands; support for efficient navigation within a screen; distinct views with consistent operations; comprehensive searching; multiple simultaneous activities to "minimize impact of errors": undo, cancel, recover from system failure, recognize and correct user error, retrieve forgotten password, verify system resources to "adapt system": customizability; internationalization to "feel comfortable": display system state; work at the user's pace
Response Measure	Task time, number of errors, number of problems solved, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, amount of time/data lost

Table 2-6 Usability General Scenario, from (Bass et al., 2003)

2.2.2 Product Line Architecture (PLA)

As already mentioned, the architecture permits or precludes system quality attributes. It also determines the structure and management of the development project in addition to the resulting system, as teams are formed and allocated around architectural components. In the product line development, the software architecture plays an even more important role. As mentioned in (Svahnberg; Bosch, 2000):

> The role of the software product line architecture is to describe the commonalities and variabilities of the products contained in the software product line and, as such, to provide a common overall structure.

As part of the architect's job, there are two things to consider: the identification of variation points and the mechanisms to support them. Variations can be substantial, since, as pointed out by (Clements; Northrop, 2001): "products in a product line exist simultaneously and may vary from each other in terms of their behavior, quality attributes, platform, network, physical configuration, middleware, scale factors and a myriad of other ways."

The identification of variation points is a continuous activity. Variations can be discovered during requirement gathering, architecture design, and also during the implementation of a second or subsequent product in the line. Those variations can include features, platforms, user interfaces, target markets and implementation options. Supporting variability can take many forms in a Product Line Architecture (PLA). They can be accomplished, for example, by the introduction of build-time parameters, assuming that all variants have been envisioned. Inheritance and delegation, from object-oriented languages, can enable variation by specializing particular classes. Entire components can be replaced by others that embody particular variants. These and other techniques focusing on the enablement of functional variability are discussed in (Gacek; Anastasopoules, 2001) and (Svahnberg; Bosch, 2000).

The documentation of a PLA carries two responsibilities. The first is to describe the architecture. Architectural views, as described in (Clements et al., 2002), come into play to describe runtime and processes interaction, structural elements, allocation and deployment of components, data flow, and so on. For a software product line, the views must show the variations that are possible. The second responsibility when documenting a PLA is to explain the architecture's instantiation process, i.e., how the product plan will deal with the architecture. The documentation must clearly show the variation points, how to exercise them, and a rationale for the variation. It must show how to instantiate and evolve the architecture (Bass et al., 2003).

The literature proposes several approaches to build SPL. A systematic review was performed by the RiSE Labs in order to understand and summarize evidence about the Architecture and Design (A&D) activities of those approaches (Souza Filho; Cavalcanti; et al., 2008). The main focus of the research questions was *How existing domain design approaches are organized?*. In particular, two sub questions asked about how the approaches dealt with variability and processes to deal with domain variability. The approaches selection was performed by five M.Sc. candidates and two Ph.D. in conjunction with weekly discussions and seminars with the Reuse in Software Engineering (RiSE) Labs. After data source collection and analysis, nine approaches were selected, and the analyses were based on 11 papers, two theses and four books. The approaches studied are described next as well as the systematic review conclusion.

The **Product Line Software Engineering (PuLSE)** (Bayer et al., 1999; DeBaud et al., 1998) was developed with the purpose of enabling the conception and deployment of SPLs in a large variety of enterprise contexts. The approach is flexible, allowing the instantiation and customization of techniques and models of requirements for a particular application, envisioning the evolution of the scope of the project and the consequent change in the requirements. Its Analysis and Design discipline (PuLSE-DSSA) has as foundation a scenario based technique, where architectural adaptations are made to fulfill a prioritized set of quality attribute scenarios. The variability is documented textually and taken in to consideration for each scenario.

The **Family-Oriented Abstraction, Specification and Translation (FAST)** (Weiss, 1999) is a software development process focused on building families. It is used in industry demonstrating a certain level of maturity at *Lucent Technologies*, where it was developed. The specific goal of FAST is to make the software engineering process more efficient by reducing multiple tasks, decreasing production costs, and shortening the marketing time. FAST has very well defined documentation regarding the development of the product line. The five step design process includes activities for developing family design, the development of an application modeling language, the establishment of a standard engineering process by which the product line will be developed and the development of the application environment. This approach does not explicitly addresses quality attribute variability.

The **Feature-Oriented Reuse Method (FORM)** (Kang et al., 1998) was developed in Pohang University of Science and Technology, Korea as an extension to the Feature-Oriented Domain Analysis (FODA). FORM adds software design and implementation phases to FODA. It prescribes how the feature model is used to develop domain architectures and components for reuse. It is a light process and seems to be very easy to be adapted in a software factory. This method is used in industry being applied on elevator control systems and telecommunication infrastructure systems (Matinlassi, 2004). It does not provide guidelines for derive reference architecture with the basis on existing product's architecture.

The **Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products (COPA)** (America et al., 2000; Obbink et al., 2000) is being developed and used at the Philips Research Labs. The specific goal of the COPA method is to achieve the best possible fit between business, architecture, process and organization (BAPO) having the greatest level or reuse as possible. This is achieved with the BAPO product family approach. The method was used in different enterprise contexts and domains, such as telecommunication, medical imaging and consumer electronics. The A&D process is guided by commercial and technical considerations. Commercial considerations are made in the form of an evaluation of which functionality is available in market (COTS) based on product and supplier quality. Technical considerations take into account the stability of components for the product family, the coupling and cohesion, single technology domain, and the implementation only of features that are always together in each product. This approach covers a wide range of quality attributes, including non-technical attributes, however it does not address quality attribute variability explicitly.

Komponentenbasierte Anwendungsentwicklung **(KobrA)**, german for component-based application development, was developed in Fraunhofer Institute for Experimental Software Engineering (IESE) and it is a ready-to-use customization of PuLSE and focuses on the architecture components (Atkinson et al., 2000; Atkinson et al., 2002). It uses UML models with stereotypes for documenting architecture and has specific phase for architecture evolution. It defines an entity called *Komponent*, which stands for a component inside the architecture that groups all kinds of models used to define the product line architecture. The processes uses decision models to deal with variability and divides the A&D process in two phases, which will lead to two logical representation of the system being developed:

Komponent specification describes the external properties of a *Komponent* with the structural model, the behavioral model, the functional model, and the decision model; and *Komponent Realization* that describes how to realize the *Komponent*'s specification.

The **Product Line UML-based Software Engineering (PLUS)** is a RUP based approach to software product line engineering defined in (Gomaa, 2004). It focuses on representing variability and other product line concerns with UML. PLUS can be seen as an extension to the Unified Process and uses UML tools and diagrams. It follows the RUP process for specifying a software architecture and introduces the variability concerns in the UML models as stereotypes.

The **Quality-driven Architecture Design and quality Analysis (QADA)** (Matinlassi et al., 2002) is a quality-driven architecture design method. It means that the architecture is built based on quality requirements. The method uses UML notation to represent variability in its models with defined stereotypes. Its design phase consists of two main steps, the *conceptual architecture design*, where the conceptual components are defined, according to the functional and quality requirements, and *concrete architecture design*, where the components are specified in a lower level of abstraction.

The **RiSE process for Domain Engineering (RiDE)** (Almeida, 2007) is based on the definition of a feature model (FODA) and the main purpose is to detail activities related with each phase of the domain engineering. During the domain design activity, the modules are decomposed, based on assets produced in domain requirements engineering, such as business goals, constraints, domain use case model, feature model, and scenarios. After the definition, the modules are refined choosing the architectural drivers that will be addressed by the architecture, choosing architectural patterns that can be applied, and allocating systems functionality to modules. The variability is represented in class diagrams, mapping the feature model with the suggested design patterns. Components are defined based on the messages changed among them and the DSSA is represented with the components defined in the previous steps.

The **SEI framework** (Clements; Northrop, 2001) was developed in the United States' Department of Defense and at the Software Engineering Institute (SEI) in order to establish patterns for the Software Product Line practice. SEI's Framework is a set of guidelines for domain architects. It has well defined steps and is based on RUP. Also it is very easily adapted since it has no suggestion of process, only steps that can be incorporated in factories processes.

From the aforementioned A&D approaches for software product line, only QADA takes into consideration explicitly the variability in quality attributes (Matinlassi, 2005).

The conclusion in (Souza Filho; Cavalcanti; et al., 2008) has shown that although many processes have very well defined guidelines, not all of them have available documentation for domain architects to follow the process and achieve the reference architecture. It also demonstrated the need to develop a process addressing these issues in order to decrease the effort in the software product line adoption for factories that started development without a software product line approach. The result of this effort is the RiPLE Design Method and can be seen in (Souza Filho, 2010).

The **RiPLE Design Method (RiPLE-Design)** was developed based on the ADD method (Bass et al., 2002), and provides as main output a Domain Specific Software Architecture (DSSA) as well as a detailed description of domain quality attributes in the form of quality scenarios. The process is divided in four main steps:

- **Architectural Drivers Identification**, where quality scenarios are to be developed and chosen to be the architectural drivers. Key features may also become architectural drivers.
- **Architectural Details Definition**, where the views needed to represent the architecture are identified based on stakeholders needs.
- **Architectural representation**, where the architecture is represented using views and specific models for each view.
- **Design Decisions Identification**, where the technology and variability techniques to be used are identified and documented with rationale.

2.3 QUALITY ATTRIBUTES IN SOFTWARE PRODUCT LINES

As already mentioned, product line architecture must enable the variability among products in the family. At the same time, as any other software architecture, it must address quality attribute requirements. According to (Kolb et al., 2004), nevertheless, “research in the field of software product lines has primarily focused on analysis, design, and implementation to date and only very few results address the quality assurance problems and challenges that arise in a reuse context".

According to (Etxeberria; Sagardui, 2005), quality attributes in product line architecture can be classified in two different types: product line quality attributes and domain-relevant quality attributes. Product line quality attributes are inherent to product lines to allow the architecture to be the basis for a set of existing products and future new products. These attributes are related to variability or flexibility, and must be achieved in order to be possible to get all the desired functionality envisioned during product line scoping. Modifiability under the form of extensibility, portability and scalability, for example, are related to the variation and evolution over time. Configurability under the form of reusability, composability and interoperability, represent variation over space.

Domain-relevant quality attributes are those targeted to a specific domain, such as performance in the real-time domain and reliability in embedded systems. As pointed out by (Bosch, 2000), it is better to address those in the beginning of the product line architecture definition otherwise consequences and implications can be very difficult to fix, requiring major effort and architectural changes. Different products in the domain may also require different levels of attributes, so there can also be variability in quality attributes.

As in functional variability, a software product line supports quality attribute variability in space and time (Bosch, 2000). Variability in space denotes divergence between the products or product variants, whereas variability in time refers to product family evolution.

The aspect of variability in quality attributes has been "neglected or ignored by most of the researchers as attention has been mainly put in the variability to ensure that it is possible to get all the functionality of the products", as discussed in (Etxeberria et al., 2008). The challenge of achieving quality attributes in single-systems becomes even more complicated in a product line context because there is variability on quality attribute requirement and different quality constraints are required. Trade-off analysis of quality attributes is also more difficult than in single-systems due to this variability and the exponential number of possibilities, as mentioned in (Etxeberria et al., 2008).

The impact of not dealing with quality attributes and the consequences of not considering and managing their variability are not trivial:

> If a product line is developed without considering the quality attribute requirements' variability, this product line will not cover all the products of the scope and will probably not cover new products in the future. As a consequence, the investment for developing the software product line will not be cost-effective. (Etxeberria et al., 2008)

In order to perceive the motivation for varying quality attributes, it is worth to remark that quality attributes affect each other, often they impact negatively (Barbacci et al., 1995). In single systems development, these trade-off situations are resolved by finding a halfway between conflicting quality attributes. Quality attribute variants are an alternative way of solving the impasse. Instead of developing one system as a compromise of conflicting quality attributes, develop a set of systems that optimize one quality on behalf of another (Myllärniemi; Männistö; et al., 2006). In many cases, these variants can be realized effectively as a software product line.

Quality Attributes can also conflict with business qualities, such as cost, time-to-market and project lifetime (Bass et al., 2003). The introduction of quality variability can also help in establishing differentiated price, e.g., a product with higher security costs more.

External varying constraints can also be helped when treated under the light of quality attribute variability. Hardware-related constraints, as the massive variation among mobile platforms are such an example. Different mobile platforms offer different memory and graphical processing capabilities, mobile game developer companies can benefit from a quality attribute variability approach producing differentiated products that benefits specifically from each platform (Myllärniemi; Raatikainen; et al., 2006).

Based on the definitions of (Niemelä; Immonen, 2007), quality attribute variability can happen in three different situations:

(i) *variation among different quality attributes,* for example, a product may require high security and in another product security is not a concern at all;

(ii) *different levels in quality attributes*. The levels define how critical a quality attribute requirement is in a product. This situation can also be seen as trade-off variability, because, as some quality attribute requirement cannot cope with one another, the prioritization guides the architect to benefit one in spite of the other; and

(iii) *functional variability may indirect cause variation in qualities*, and vice versa, for example, a variation point in the execution platform may cause variability in performance requirements. The other way round, variability in security requirements results in different user authentication policies.

2.4 CHAPTER SUMMARY

This chapter showed a brief overview on software architecture and software product line concepts. It discussed the motivation behind and benefits from the adoption of software product line engineering paradigm. The essential activities during the software product line engineering were presented, as well as some adoption models. Concerning adoption models, the risks, strengths and drawbacks of the adoption models were discussed. Industrial experiences adopting software product lines approaches were also presented in this chapter.

Next software architecture fundamental concepts were presented and its close relation to software product lines. The importance of quality attributes in software product line architecture was described, particularly the possibility of quality attribute variability and its implications.

The next chapter presents an overview on RiPLE-Design process, as well as some deficiencies encountered in the process when treating quality attribute variability.

3 THE RIPLE-DESIGN PROCESS

In this chapter, the RiPLE-Design will be described. It is a process formulated with the purpose of generating a Domain Specific Software Architecture (DSSA) that represents common and variable elements of a domain, as a part of RiSE Product Line Engineering Process (RiPLE). The process will be examined in order to understand how it addresses quality attribute variability.

The remainder of this chapter is organized as follows: Section 3.1 describes the RiPLE-Design process; Section 3.2 presents its activities and guidelines; Section 3.3 shows how the process addresses quality attribute variability; Section 3.4 concludes this chapter with its summary.

3.1 RIPLE-DESIGN

The RiSE Product Line Engineering Process (RiPLE) provides activities, roles and artifacts for every phase during the software lifecycle of a software product line. It is divided in three main areas: *core asset development*, *product development* and *evolution management*. *Core asset development* comprises processes and activities required to develop assets that will be reused across the software product line. *Product development* includes process and activities that will serve the purpose of deriving new products based on the core assets previously developed. *Evolution management* includes supporting activities, such as version control, change management and release management, and a process that need to be specific in the product line context. RiPLE defines processes and activities for each one of the software disciplines, examples are the RiPLE-Requirements process (Neiva, 2009) for requirement engineering, RiPLE-Design process (Souza Filho, 2010), for design and architecture, RiPLE-Scoping process (Moraes, 2010), for product line scoping and RiPLE-Test (Machado, 2010) for software testing in the context of software product lines.

The RiPLE-Design process focuses on defining a Domain Specific Software Architecture (DSSA), which represents the architectural elements from the software product line. Such architectural definition must enable variability among products in a certain domain and must take advantage of commonalities among those products in order to promote software reuse.

Following the RiPLE-Design process, an architect can systematically define the DSSA in an iterative and incremental way, using clear models and techniques.

The process receives as mandatory inputs a *feature model*, representing the mandatory, optional and alternative domain features; a list of domain *stakeholders* and the description of the domain's *non-functional requirements*. Optional inputs include *domain requirements*, *domain use cases* and *quality scenarios*. Quality scenarios, if not provided, will be developed during the process. The main output produced by the process is the DSSA representation, which includes traceability between architectural models and domain features. The DSSA representation is documented in a clear and concise form, in order to satisfy the main stakeholders involved in the construction of a software product line. Another possible output from the process is the Quality scenarios description, if not provided as inputs are produced within the process as well.

The process was validated in an experimental study involving nine subjects (Souza Filho et al., 2009). The subjects were seven M.Sc. students and two Ph.D. students from the Federal University of Pernambuco. Among them, three students were graduated for more than five years, while the others for less time. The experimental study was conducted during part of a M.Sc. and Ph.D. course in software reuse, in November 2008, at Federal University of Pernambuco (UFPE), Brazil.

The study analyzed the viability of subjects using the process to design an easy to change and simple, i.e. not complete, DSSA. This study also analyzed the effort spent in this process in comparison with the whole SPL life cycle for the project under development.

Based on the collected results, the analysis performed in this study showed that the process can be viable for the definition of a DSSA for web applications domain, even with the reduced number of nine subjects. Although the analysis also identified some directions for improvements, none of the improvements pointed to address quality attributes variability.

3.2 ACTIVITIES

The Process is divided in four main groups of activities: *Identify Architectural Drivers*, *Define Architectural Details*, *Present the Architecture* and *Identify Design Decisions*. An overview of the process, including the four steps can be seen in Figure 3.1.

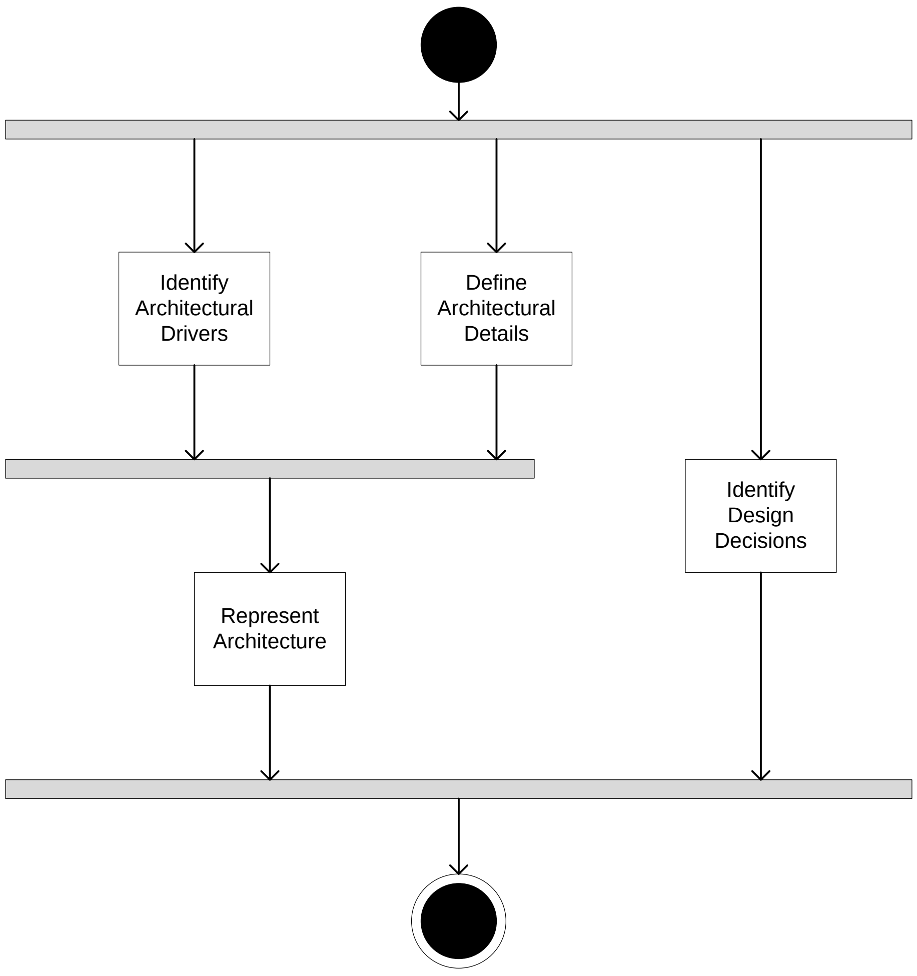
Identify Architectural Drivers
Define Architectural Details
Identify Design Decisions
Represent Architecture

Figure 3.1 RiPLE-Design Overview, in flowchart notation[2]

3.2.1 Identify Architectural Drivers

In this activity, three main tasks take place. Figure 3.2 shows the steps needed to identify the architectural drivers. First, if not provided, quality attribute scenarios are developed, based on non-functional requirements, with the purpose to better represent quality attributes

[2] This and other flowcharts obey to the notation from (International Organization For Standardization/International Electrotechnical Commission, 1985)

requirements that the architecture needs to achieve. An elementary quality attribute scenario is a pair of *stimulus* and expected *response* that describe the expected behavior of the system under certain situation (Bass et al., 2003). Besides the stimulus and the expected response, in RiPLE-Design, some other aspects related to the quality attribute are also described, namely, the *source of the stimulus*; the *environment*, i.e., the conditions within the stimulus occurs; the stimulated *artifact*, which can be the whole system or pieces of it; and the *response measure*, that will help the architect to test some quality attributes requirement (Bass et al., 2003; Souza Filho, 2010). The quality attribute scenarios must then be ranked based on their importance to the domain. A catalog of general quality scenarios can be found in (Bass et al., 2003).

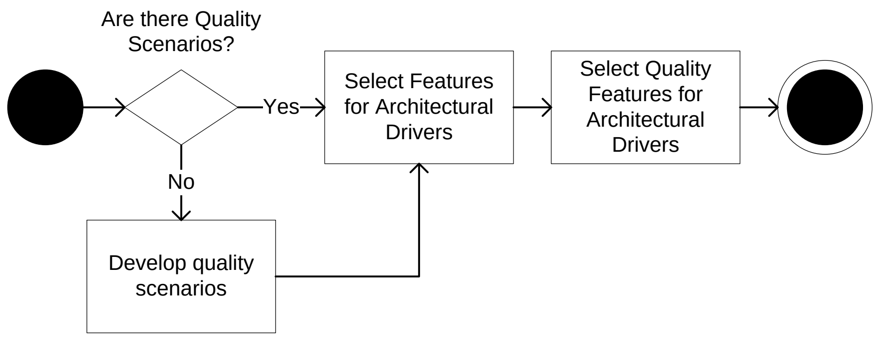
Are there Quality Scenarios?
Yes
No
Select Features for Architectural Drivers
Select Quality Features for Architectural Drivers
Develop quality scenarios

Figure 3.2 Identify Architectural Drivers, in (Souza Filho, 2010)**, in flowchart notation**

In the second task in this activity, functional features are selected to figure as architectural drivers. The architectural drivers will guide the main architectural definitions, and the selection of key features following their importance to the domain, which is mainly based on the business value of certain feature to the domain. Architectural dependency may also influence the priority of a particular feature.

In the last task, the quality attributes represented in the scenarios are prioritized and selected to be architectural drivers as well. This task also involves the identification of conflicts among quality attributes and its documentation. The prioritization of quality attribute scenarios is a key aspect of the RiPLE-Design method, because conflicts between quality attributes are not rare, and the architect must address those conflicts by selecting as the top priority quality attributes to be addressed first. The next activities will shape the architecture, so that the quality attributes are fulfilled according to their importance.

3.2.2 Define Architectural Details

This activity serves the purpose of defining the level of detail that will be used to describe the architecture in each one of the behavioral, structural and process views. This definition takes as input a list of stakeholders and decides based on it, the proper level of detail in each view. Customers, as an example, are more likely to be interested in high-level structural and behavioral views description. In this case, the structural view should depict modules and main components, while the behavioral view should show the key interactions among them. Product implementers may need more details, as a result, more classes and interactions among classes may appear in the architectural views. This definition is made by the domain architect and documented in the DSSA description document.

3.2.3 Represent Architecture

The definition and documentation of architectural models are done in this activity. The three views, i.e., structural, behavioral and process, are defined in different steps.

The structural view shows the domain architecture static structure. It also shows how variability is achieved inside this structure. In RiPLE-Design, the structural view definition is composed of component and class diagrams. The formers show how the whole system is divided into modules and those modules into components and the latter shows the classes that will guide the implementation of each component.

During the definition of the architecture structure, Architectural drivers are selected from the prior ranked list and guide the choice of the architectural styles that will help to accomplish the desired quality attributes requirements. With the architectural styles chosen, modules are defined representing high-level abstractions based on domain features, requirements and use cases. High-level modules are refined as new quality attributes come into play and must be fulfilled. Figure 3.3 depicts the iterative process of module definition.

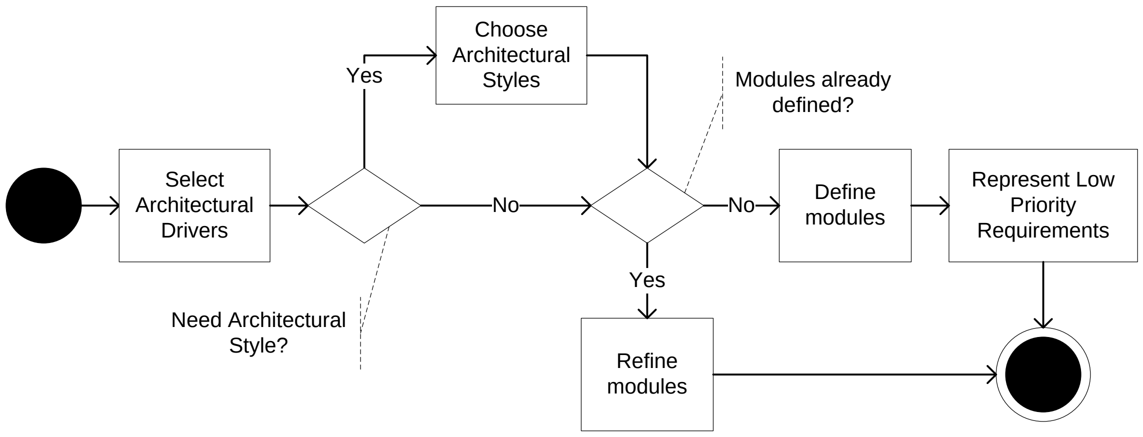
Choose Architectural Styles
Yes
Modules already defined?
Select Architectural Drivers
No
No
Define modules
Represent Low Priority Requirements
Yes
Need Architectural Style?
Refine modules

Figure 3.3 Module definition Activities, in (Souza Filho, 2010)**, in flowchart notation**

Following the structural view definition and documentation, components that will be part of each module are defined. The definition of components in a software product line context is extremely important, as the components must hold variability found in the domain requirements. The RiPLE-Design process shows us forms of variability represented in components: (i) external variability, where a component holds the implementation of certain feature, and can be presented on a product or not, depending on the selection of that feature; (ii) internal variability, where the internal structure of a component changes depending on certain feature, e.g., algorithm choice; (iii) structural variability, where external structure of a component can differ from one product to another; (iv) configuration variability, where different arrangements and configuration of components can be done to achieve certain feature selection.

In RiPLE-Design, components can be defined from domain features and use cases. When the domain features are used, the resulting components represent a feature or a group of feature that can be implemented in a single component. The variability inherent to a group of features is represented internally or externally in the component using UML component diagrams and stereotypes. High-level variant features lead normally to components with external variation, while features that represent implementation selection and details lead to internally variant components.

When the components are defined from use cases, the RiPLE-Design process follows the guidelines proposed by the RiDE process in (Almeida, 2007). The technique follows three steps to group, define and specify components based on their functional dependency. Component grouping starts measuring functional dependency between use cases based on the subsystems they belong to, the actors involved in the use cases, the amount of data shared among the use cases and the coupling between them. The four criteria define a metric that is used to create use case clusters. The clusters help defining the set of use cases that will be realized by each component. The component is then specified through its interfaces and classes.

As the fine grain structural representation is normally needed, classes that will implement each component must be defined. Some techniques can be chosen to accomplish the desired variability. Aspect orientation, conditional compilation, design patterns and simple parameterization are examples (Gacek; Anastasopoules, 2001). The selection of the proper technique is made by the Domain Architect and documented in the DSSA. RiPLE-Design provides guidelines to use design patterns to implement variability as follows. *Alternative* features can be implemented with design patterns that allow substitution and varying construction of classes, so, the *Prototype*, *Abstract factory*, *Builder* and *Strategy* design patterns are good options to implement this kind of features. *Or* features need patterns that adapt objects and guarantee that one implementation is always available should be used. *Adapter*, *Bridge*, *Decorator* and *Chain of Responsibility* are examples of such patterns (Gamma et al., 1995). *Optional* features can be implemented through the same patterns of *Or* features, except that in this case, there is no need to guarantee that at least one feature is present.

The Behavioral view is represented next with the development of sequence diagrams based on the functional features and the defined classes in the structural view. This activity is mandatory since the domain behavior can aggregate much information to developers during implementation. Functional features and use cases are analyzed in order to collect information that will fill the sequence diagrams. The classes in the sequence diagrams come from the fine-grain structural view diagrams. Variable messages can also be presented in these diagrams.

Taking into consideration some quality attributes, such as performance and availability, the architecture can decide to perform a thorough analysis of runtime characteristics of the domain to address issues related to concurrency, distribution, fault tolerance and system integrity. In this case, the RiPLE-Design process provides activities do define and represent the Process View of the architecture using activity diagrams to represent processes, thread and other runtime concerns.

During the whole activity of architecture definition, the domain architect plays a central role and is aided by experienced developers. The domain manager also helps prioritizing quality attributes and defining the key features of the domain.

3.2.4 Identify Design Decisions

This activity is performed throughout the entire design process. Decisions made during architectural development should be documented in conjunction with a rationale about the selected decision, alternative decisions left aside and possible enhancements to the chosen option, so that other architects and developers can identify the motives behind the decision. Examples of important choices to document are technology choices, variability techniques, and the selection of one architectural style over another.

3.3 DESIGN OF VARIABLE QUALITY ATTRIBUTES

Regarding the achievement of quality attributes, RiPLE-Design's approach consists in prioritizing quality attributes under the form of quality scenarios, and evolving the architecture by choosing architectural styles and patterns that enable the architecture to satisfy the planed quality attributes. The RiPLE-Design process, however, does not define how to treat quality attribute variability. Therefore, this work makes an assumption that it could be possible to define a DSSA that could deal with quality attribute variability using known architectural styles and patterns.

In this context, this work will assess how quality attribute variability could be addressed using RiPLE-Design. The assessment will be built on top of real scenarios. The scenarios were extracted from a software product line for a paper submission system, called RiSE Chair, which was developed by the RiSE Labs for academic purposes. The product family can handle various flows of submission such as papers submissions for a journal or a conference. It enables users to evaluate papers in a single or many rounds and provides different ways to grade and select papers for publication. The software product line provides features to handle assignment of reviewers automatically or manually as well. The complete feature model of the RiSE Chair SPL can be seen in Figure 3.4. It comprises 63 features being 13 optional , 2 alternative, 18 inclusive-or and 30 mandatory features.

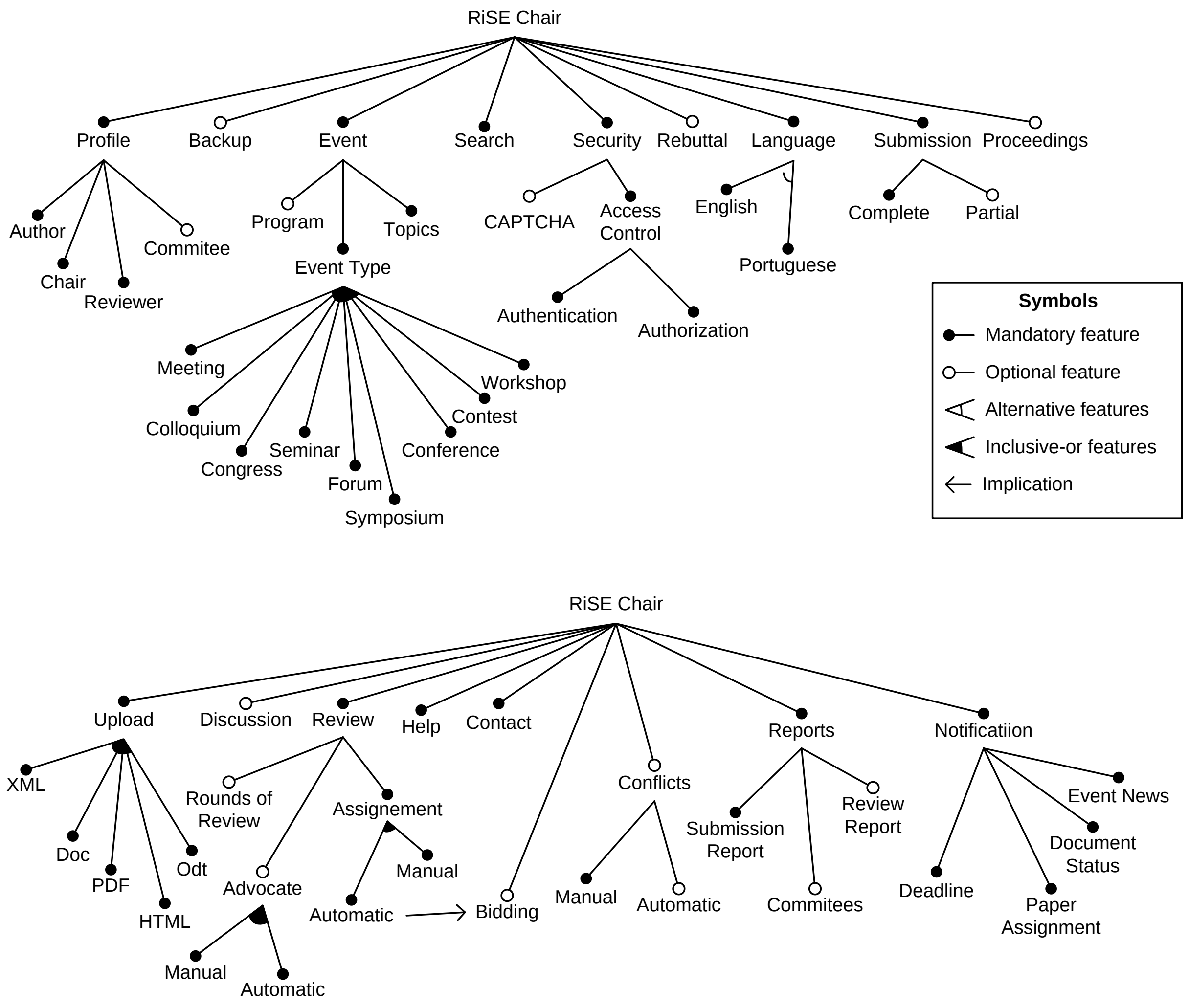

Figure 3.4 Feature Model Diagram for Rise Chair

The development of the RiSE Chair software product line followed the complete RiPLE process. Specifically, the domain architecture development followed RiPLE-Design and started by deriving quality attribute scenarios from non-functional requirements. As an example, Table 3-1 shows a modifiability scenario for the RiSE Chair SPL.

Source:	Developer
Stimulus:	Wishes to change
Artifact:	The User interface
Environment:	During development
Response:	User interface is changed
Response Measure:	In less than 4 hours with no impact to the rest of the system.

Table 3-1 Modifiability scenario for the RiSE Chair SPL

Following the RiPLE-Design process, the Architectural Drivers Identification task has lead to two drivers. They were:

(i) *mandatory features*;

(ii) *variability found in some functional requirements*

These two characteristics lead the architectural development, being the key features access control, paper submission, paper revision, event management, notification and internationalization. The variability present in some functional features was the main reason why a product line approach was chosen. The architecture was designed to support high modularity and flexibility.

Modularity and flexibility are the key characteristics of software product line architecture. Loosely coupled and highly cohesive modules permit the reorganization of the architecture in order to achieve high degrees of reuse. Flexible architectures allow a large range of product variants to be implemented easily. Such an architecture must be simple to change and very tailorable, so, when new products with alike, but not the same features, need to be developed, they can be delivered in less time with less effort.

From the *Define Architectural Details* activity, the DSSA should be described in low level details.

The next activity in the RiPLE-Design process is the *Represent Architecture* activity. During the structural view definition task, an architectural style was chosen to address the architectural drivers. The layered architectural style was chosen as it promotes high modularity and independence among layers.

On top of the layered architecture, a web platform was chosen to deliver the functionalities. The web technologies served well the purpose of submission systems. It enables peers to submit papers from anywhere in the globe. Conference chairs and papers evaluators could also work from anywhere without downloading any application, since the web application works inside their browsers.

The *module definition task* produced the structural representation of the modules shown in Figure 3.5.

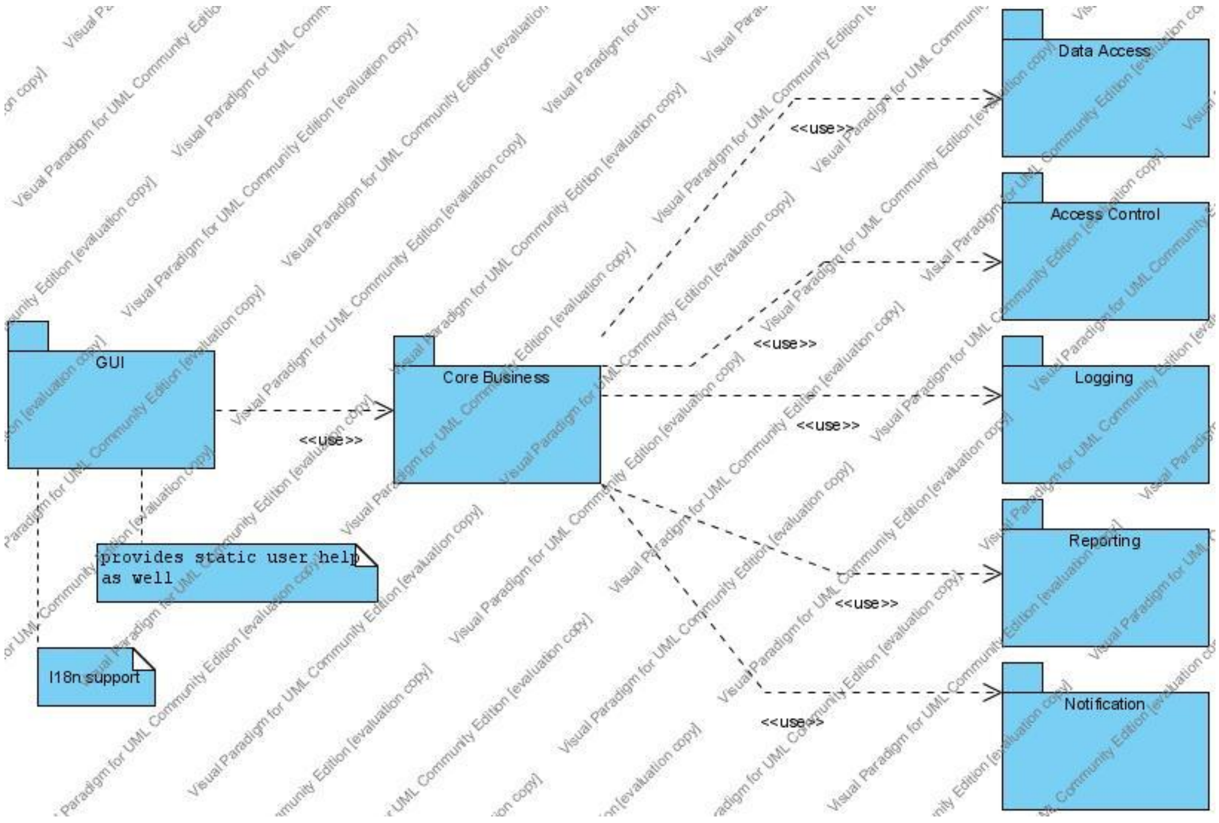

Figure 3.5 RiSE Chair Module View, in UML notation[3]

After the module definition, components from each module were defined as well as the relationship among them. As a result of the *component definition task*, the *RevisionImpl* component, responsible for handling assignment of reviewers for papers and registering the grades from each reviewer, is shown in Figure 3.6.

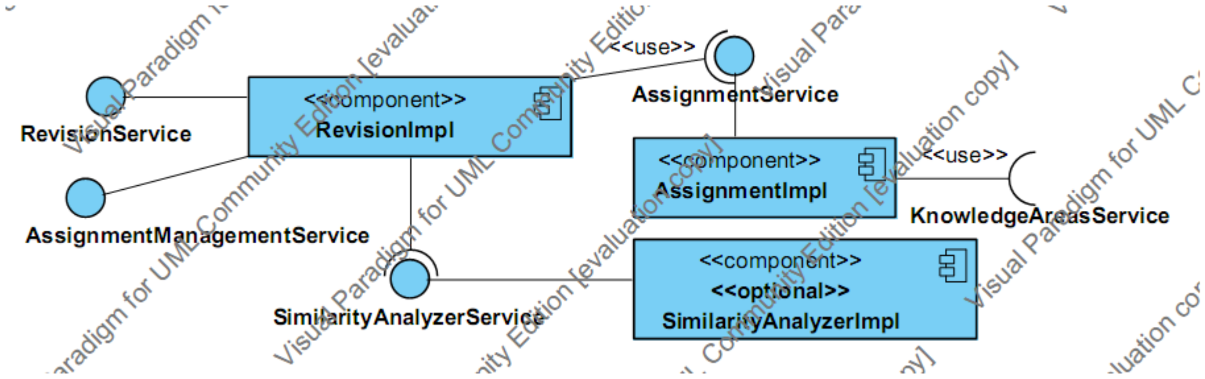
RevisionService
AssignmentManagementService
<<component>> RevisionImpl
<<use>>
AssignmentService
<<component>> AssignmentImpl
<<use>>
KnowledgeAreasService
SimilarityAnalyzerService
<<component>> <<optional>> SimilarityAnalyzerImpl

Figure 3.6 Structural view of the Revision component, in UML notation

[3] Following the UML Notation, from (Fowler; Scott, 2000)

The *behavioral view definition task* follows the structural view definition and an example of its result is illustrated in Figure 3.7.

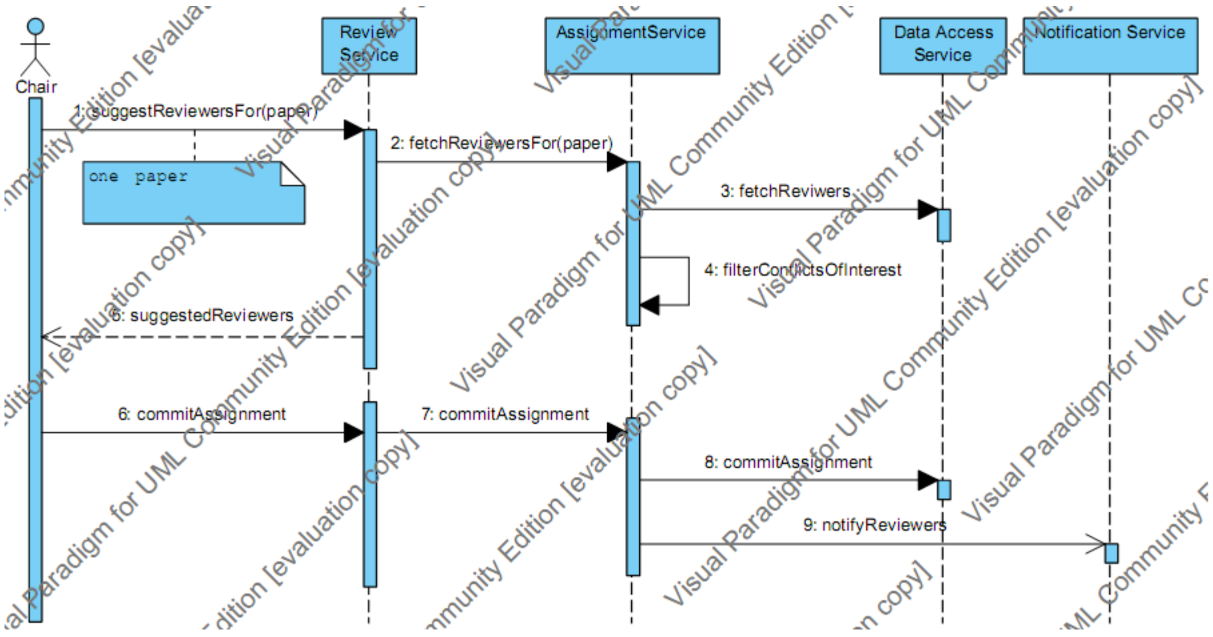

Figure 3.7 Sequence diagram: invite reviewer, in UML notation

Given the layered web platform architecture as a basis, the next subsections describe, for each case of quality attribute variability, according to the definitions of (Niemelä; Immonen, 2007), how the architectural solution was adapted using the RiPLE-Design process.

The case of functional variability affecting quality attributes is related to the occurrence of feature interaction in software product lines. As pointed out in (Lee; Kang, 2004), sometimes "[…] features cannot perform their functionalities alone, they need to interact among them in order to accomplish the products requirements. In this context, a feature interaction occurs in a system whose complete behavior does not satisfy the separate specifications of all its features." Functional features that impact on non-functional features are, thus, a case of feature interaction. The problem of feature interaction can impact the whole SPL development process, as it promotes changes in reusable assets and impacts maintenance costs and other products. The adaptation of the RiPLE Process to perform dependency analysis among features and analyze how they impact on each other is described in other Project from the RiSE Labs. The case of functional variability affecting quality attributes, so that quality attribute variability occurs, will then be left out of the scope of this work.

3.3.1 Variations among different quality attributes

Most of RiSE Chair products have low availability demands, as exemplified in the quality scenario show in Figure 3.8.

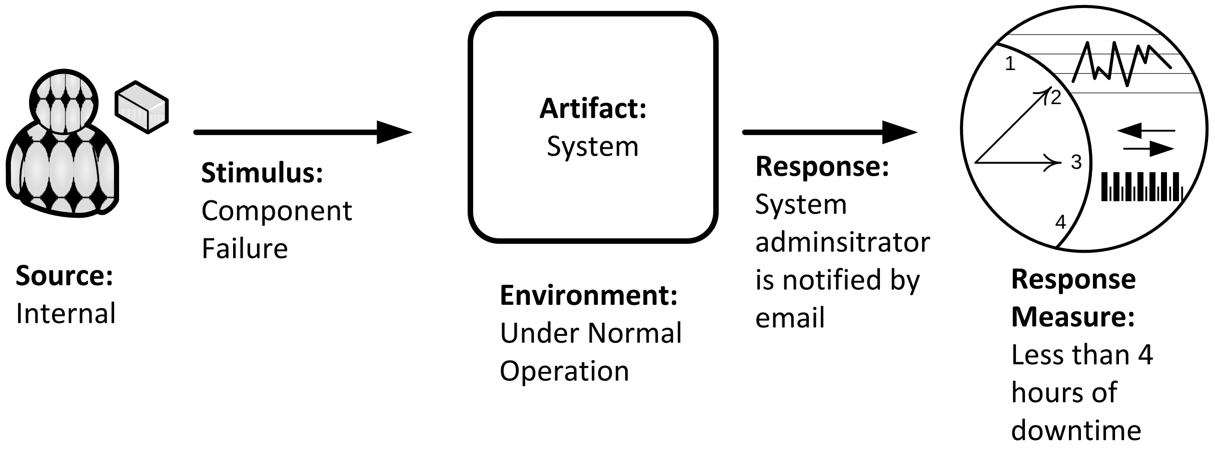

Figure 3.8 Low demand availability scenario

Other products, however, have a requirement for high *availability* and *reliability*, which can be seen in Figure 3.9.

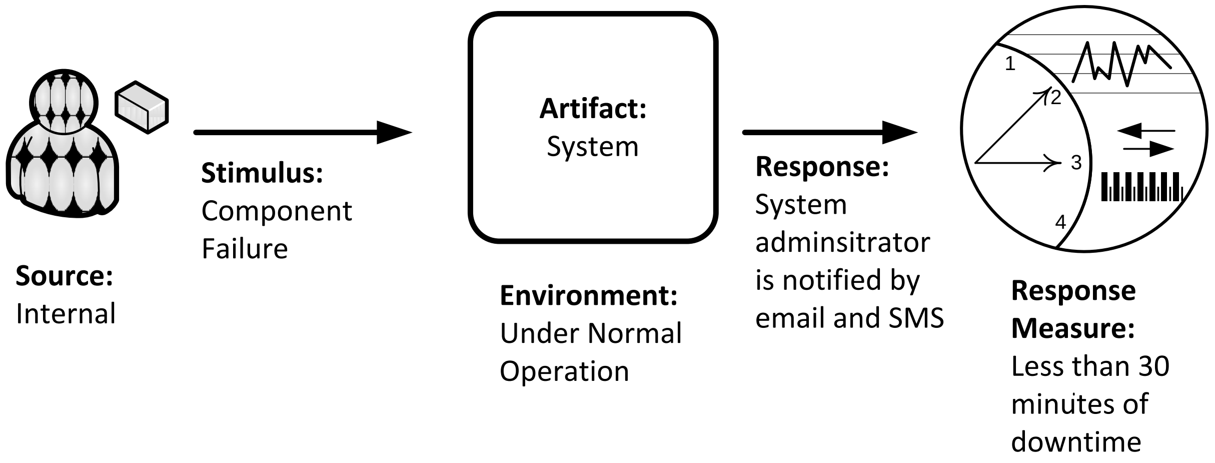

Figure 3.9 High demand availability scenario

Availability and reliability can be categorized as *dependability* attributes. According to (Laprie, 1992), *dependability* is that property of a computer system such that reliance can justifiably be placed on the service it delivers. As stated by (Barbacci et al., 1995), *availability* measures the readiness for usage of a system while the *reliability* of a system is a measure of the ability of a system to keep operating over time.

Some products of the software product line are aimed at large conferences with worldwide submission. As someone could be submitting a paper from anywhere in the globe, it can happen anytime. So, the system must be online 24 hours a day, every day such a conference is being held. Other classes of products do not have high concerns with availability.

Following the RiPLE-Design process, the architect must prioritize the quality attribute scenarios and adapt the architecture structure to address quality issues. In this case, the architect pondered that the high available system would affect more the architecture definition than the less available option. So, the quality scenarios priority established the low demand for availability as less important than the other availability scenario. The architecture was built as a consequence of that priority.

According to (Bass et al., 2003), all the approaches to maintaining availability involve some type of redundancy, some type of health monitoring to detect a failure and some type of recovery when a failure is detected. Many of the available tactics to address availability issues are available within standards execution environments. Database transactions, for example, can be considered a fault prevention tactic.

Automatic fault detection tactics are very common in real time and mission critical systems, examples are *Ping/echo*, when one component issues a ping and expects to receive back an echo, within predefined time, from the component under inspection; and *Heartbeat*, where one component emits a heartbeat message periodically, and another component listens for it. In both cases, if the heartbeat or the echo fails, the component is assumed to have failed and a fault correction component is notified.

In this case, there is no need for automatic detection of failures, as none of the products of the RiSE Chair product line involve high risks of money or human losses. The fault detection mechanism will be manual.

For the desired degree of availability, the architecture will only focus on fault recovery. Fault recovery often involves redundancy. Automatic fault recovery mechanisms are very complex. They are used, for example, in air traffic control systems. Examples of fault recovery tactics are, as in (Bass et al., 2003), *Active redundancy*, *Passive redundancy* and *Spare*. In *Active redundancy*, several components are maintained in the same state and respond to events in parallel. The response from only one component is used and the others discarded. When the fault occurs, the system's downtime is milliseconds since the time to recover is only the switching time. *Passive redundancy* consists of one primary component responding to events and informing the other standby components of state updates they must make. When a fault occurs, the system must first ensure the backup state is sufficiently fresh before resuming services. In the *Spare* tactic, a standby computing platform is configured to replace many different failed components. Somehow it must be managed to the spare platform to be at the same state as the primary platform. The downtime of this tactic is usually minutes. Concerning fault recovery, it is also important to define a fault model, describing which types of fault will be considered and which ones will be ignored. In the case of RiSE Chair, only internal component failures were considered.

For the RiSE Chair architecture, the *Spare* tactic was applied. In this special case, the standby platform consisted in another web application server ready to go associated with redundant databases. This strategy was used because of its simplicity. It can be easily applied, and does not influence maintainability, modifiability or any other key quality attribute, since the system is replaced as a whole, without affecting internal components. A proxy server must be placed to receive and reroute user HTTP requests.

Figure 3.10 shows the deployed system under normal operation.

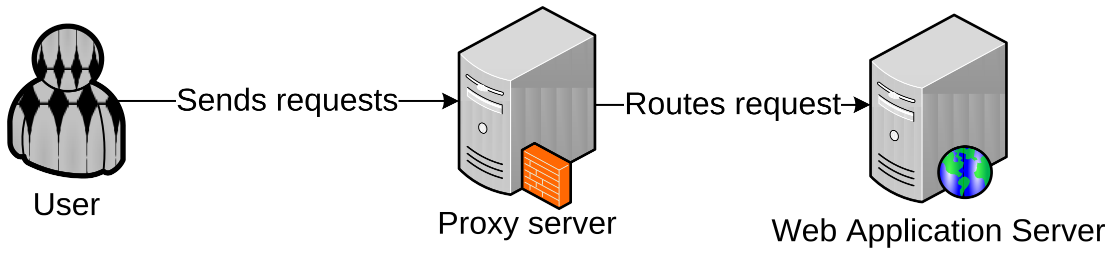

Figure 3.10 Deployment diagram. System under normal operation

Figure 3.11 shows the deployment configuration when a fault occurs, and the proxy is set up to reroute the requests to a spare web application server.

The products derived from the RiSE Chair software product line are mainly data-driven, meaning that the state of the system is in its data. It is easy to manage the spare platform to maintain its state equivalent to the primary platform, since the database redundancy can be configured and automatically provided by the database server.

The spare tactic is associated with error reports to the system administrator, who would be responsible to setup the HTTP requests rerouting. This strategy enabled an estimated downtime of a few minutes to an hour, which would be satisfactory even for the most critical products.

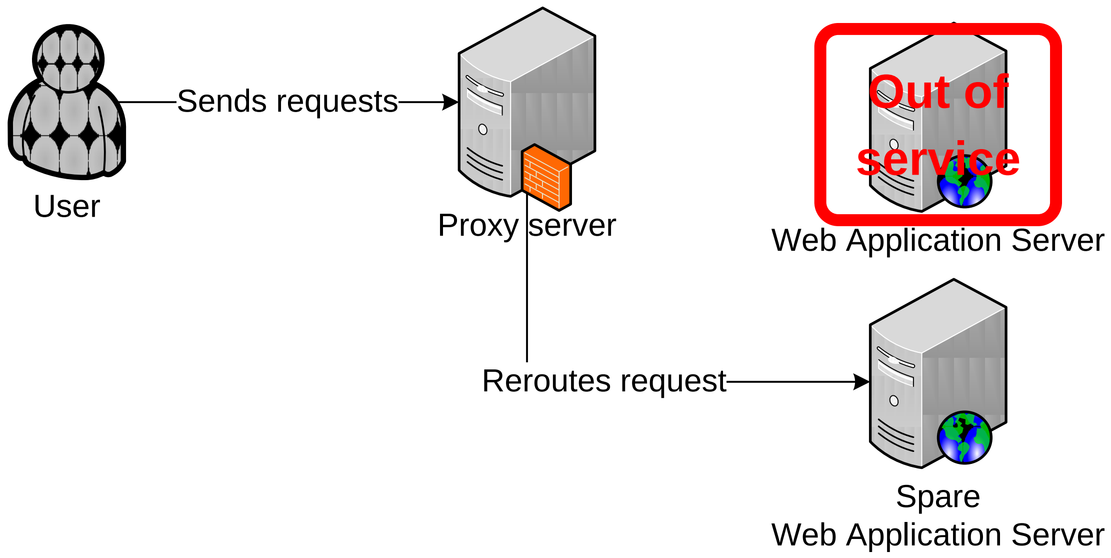

Figure 3.11 Deployment diagram. Under faulty operation, proxy server as a request router

It is important to remark that although the variability in system downtime is not due to the used process, but to the manual treatment, the request rerouting. The single architectural solutions, aims to solve the most critical scenario.

3.3.2 Different levels in quality attributes (trade-off variability)

To exemplify the case of different priority levels of quality attributes, there is a case in the RiSE Chair product line where some products demand high security and another group of products that demand short latency with high scalability. Business research has shown that some users are willing to pay more for more secure systems, even if they are not so fast. On the other hand, some users do not have much confidentiality concerns, they prefer a faster system.

Figure 3.12 shows the performance scenario for the class of products with high performance demands. Other class of products, demands higher security, this case is exemplified in Figure 3.13. The main security concern is *confidentiality*, which is the requirement that data and processes to be protected from unauthorized disclosure (Barbacci et al., 1995). Also according to (Barbacci et al., 1995), *latency* refers to a time interval during which the response to an event must be executed.

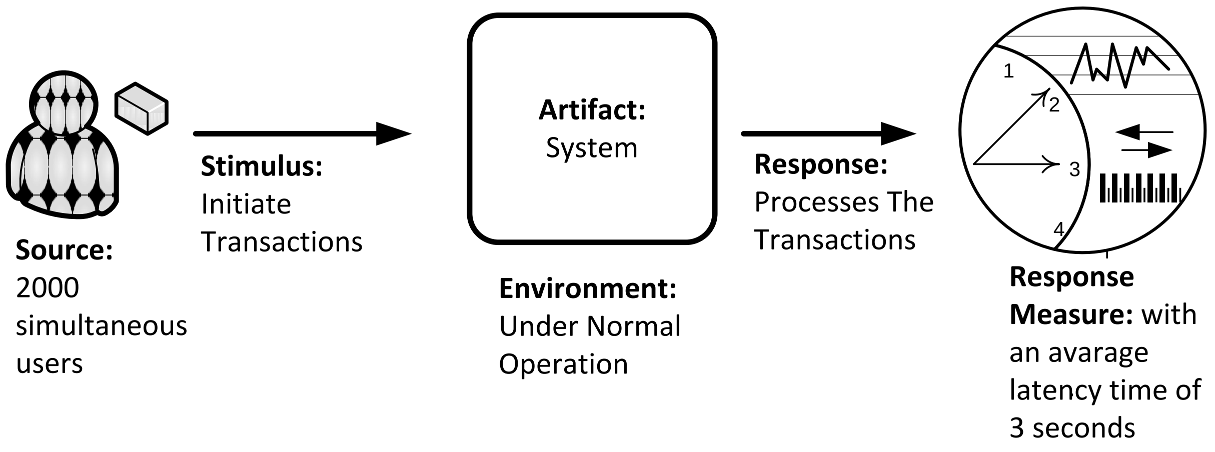
Source: 2000 simultaneous users
Stimulus: Initiate Transactions
Artifact: System
Environment: Under Normal Operation
Response: Processes The Transactions
Response Measure: with an avarage latency time of 3 seconds
1
2
3
4

Figure 3.12 Performance/scalability scenario with short latency

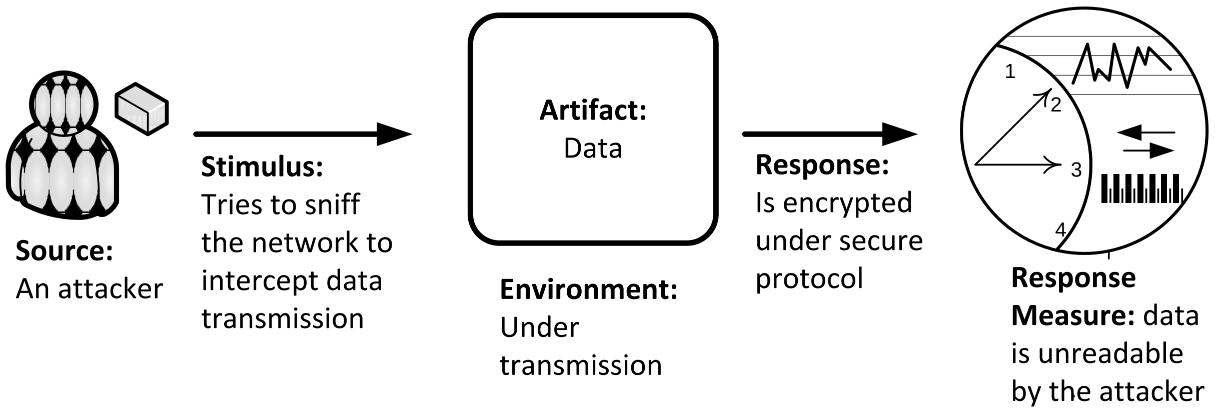
Source: An attacker
Stimulus: Tries to sniff the network to intercept data transmission
Artifact: Data
Environment: Under transmission
Response: Is encrypted under secure protocol
Response Measure: data is unreadable by the attacker
1
2
3
4

Figure 3.13 Confidentiality scenario

Those two quality attributes are often in opposition as higher security leads to encryption protocol that are time consuming. This situation reflects the business need for two different classes of products where some customers are willing to accept less security if they can get shorter response time, and other customers prefer more security even though the performance is hindered. Hence, as discussed, security needs are often in opposition to performance requirements.

Following the RiPLE-Design process, the quality attributes must be prioritized, and one quality attribute addressed after another. The occurrence of variability in quality attributes has no precedents in RiPLE-Design process. This means that the process gives no guideline about how to represent that in some products *security* is more important than *latency* and other products are just the opposite.

During the prioritization of the quality attribute, the domain architect together with the domain manager must come to a single priority queue, and evaluate what are the impacts of having *security* before *latency* or the other way around. As an architect, one way to follow is to assume that shorter response time *and* stricter security are possible at the same time. The domain manager must also agree that having both quality attributes at the same time is viable, or perhaps it is the only path to follow. Note that the domain manager must leave aside the business concerns that may have lead to the conclusion of having two classes of products and benefit from the trade-off variability between security and performance. Both, domain architect and domain manager must also come to an agreement of the lowest acceptable performance demand. The scenario in Figure 3.14 shows this.

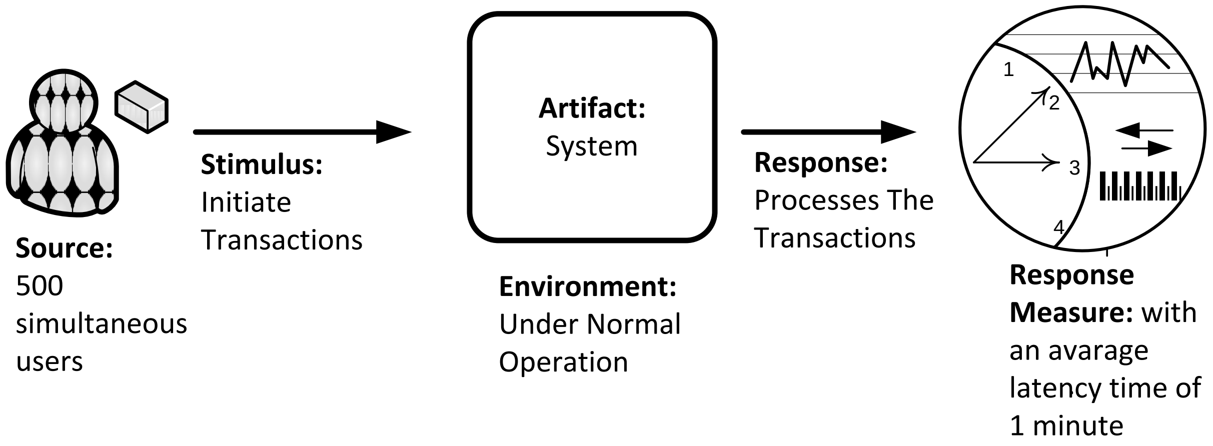
Source: 500 simultaneous users
Stimulus: Initiate Transactions
Artifact: System
Environment: Under Normal Operation
Response: Processes The Transactions
Response Measure: with an avarage latency time of 1 minute
1
2
3
4

Figure 3.14 Performance scenario with longer latency

Latency requirements can be measured and the proposed architecture can be validated against them, but the architect would address them first. According to (Bass et al., 2003), there are two main contributors to the latency of a request: (i) *resource consumption*, such as CPU, data stores and network communication bandwidth and (ii) *blocked time*, which can be caused by contention of resources, availability of some resource or dependency in other computation.

The tactics to address latency issues fall into three categories: *resource demand*, *resource management* and *resource arbitration*.

Resource demand tactics that can be introduced in this solution include *Increase computation efficiency*, by optimizing some algorithm from a critical area; and *Reduce computational overhead*, that is mainly removing intermediaries in an event stream being processed, which may worsen the modularity of the architecture.

Resource management tactics can also help to reduce latency, some of them are described in (Bass et al., 2003) and can be applied in the web application platform: *Introduce concurrency*, if the requests can be processed in parallel, different threads of execution can process different stream of events; and *Increase available resources*, simply providing faster processors, additional memory and faster networks can help to reduce latency, with the obvious cost/performance trade-off.

Resource arbitration tactics comprise scheduling techniques and criteria. They are common in real time systems, in which time deadlines must be met.

The proposed solution includes *Introduce concurrency* and *Increase available resources*. The *introduce concurrency* tactic is characterized by the proxy server, described earlier, to function also as a load balancer, directing request to different spare servers, as seen in Figure 3.15. This tactic can be seen as an enhancement to the *spare* tactic applied to address availability. In this case, there is no need of an application server runs out of service, the load balancer can redirect the requests to another server. Load balancing here can help to maintain the level of service when a large number of users access the system.

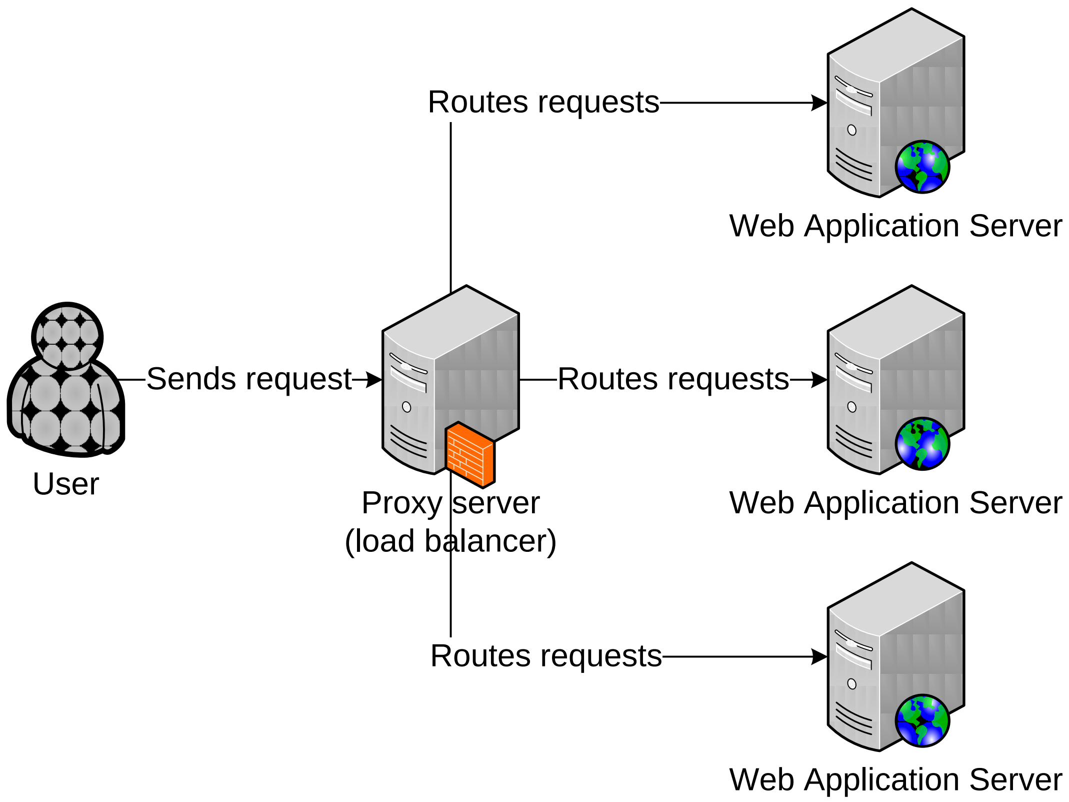

Figure 3.15 Deployment diagram. Load Balancing

The second tactic, *increase available resources*, works as a fine tuning, adjusting the deploy setup to meet the latency requirements. The *Increase computation efficiency* tactic did not seem to apply, since the products involved no kind of complex computation.

The next step is to introduce the security issues and adapt the architecture to meet these requirements while maintaining latency quality attributes. Tactics for achieving security can fall into three categories: *resisting attacks, detecting attacks* and *recovering from attacks*. (Bass et al., 2003).

Most of security tactics are concerned with resisting attacks. They include *Authenticate user*, where simple login-password pairs or complex biometric identification can be used; *Authorize user,* to ensure that an authenticated user has the rights to access and modify either data or services; *Maintain data confidentiality*, which is normally related to encrypting data and communication channels; *Maintain Integrity*, implemented using checksums or hash results; *Limit exposure* to certain services, or to distribute the service hosting so an attack does not affect all data and services at once; and *Limit access*, which is to restrict the access to known sources, through firewalls or a Demilitarized Zone (DMZ) (Bass et al., 2003).

Attack detection can be done with some kind of *Intrusion Detection* System (IDS). Those systems can analyze the patterns of requests and user behavior to infer whether an attack is in progress.

Tactics to recover from attacks are concerned with *restoration* of services and data, and *identification* of attacker. Restoration tactics are the same already described for availability issues. The main identification tactic is to *maintain an audit trail* from each transaction applied to the data together with identifying information (Bass et al., 2003).

The RiSE Chair software product line already mentioned in its feature model the need for user authentication and authorization. These security tactics were implemented into a specific module and further refined into components, without major changes in the architecture. The security concern that needed further analysis, as shown in Figure 3.13, involves data confidentiality. In this case, the *Maintain data confidentiality* tactic was applied. The encryption link was implemented by a Secure Sockets layer (SSL) very common in web applications.

Using secure protocols and data encryption, it is expected that the overall system latency increases. If the security tactic damages performance, adjustments in deployment can be made, i.e., better processors in the deployment machine. As described before, adjustments in the performance can be made adopting the *Increase Available Resources*.

3.3.3 Discussion

RiPLE-Design focuses on producing an architectural definition that enables variability among products in a certain domain and takes advantage of commonalities among those products in order to promote software reuse. This architectural definition must, at the same time, satisfy the proposed quality attributes requirements represented in quality attributes scenarios.

The examples described above showed how it would be to use RiPLE-Design process to deal with variability in quality attributes. In the first case, the availability and reliability demands were high only in a single class of products, in other words, availability was optional to that class of products. The use of a ranked list of quality attributes directs the architect to consider the availability scenario, even though those quality attributes were different from product to product. The need for availability guided the architecture development and a single architectural solution was proposed to satisfy both cases, even when that extra complexity was not needed. Thus, there will be an over engineered class of products, with robust infrastructure that will seldom be used.

The second example showed two classes of products that could take advantage of the trade-off between security and performance. The solution proposed by following RiPLE-Design, however, lead the architecture definition to an over engineered solution where both security and performance would coexist. Although the solution was coherent and viable, it discarded the variability among the classes of products, and delivered a single architectural solution. This situation hides the trade-off between both quality attributes, as it tries to achieve both requirements at the same time, in other words, every product in the product line would have high security and short latency. For a class of products, the confidentiality quality attribute represented a layer of computation that could be dismissed in order to increase performance, following the *Reduce computational overhead* tactic.

Both examples, resulting in a single architectural solution did not cope well with quality attribute variability, they simply left aside the reason why the quality attribute variability needed to be exploited. One sound reason to have quality attribute variability, in the case of trade-off between confidentiality and latency, was the business advantage of offering differentiated products to specific markets. The proposed solution simply considered that faster products would please both market shares. Yet, faster does not always mean better. The single solution brings the problem of customers willing to pay for slow but secure systems will have to pay also for the infrastructure needed to make that system fast.

The guidelines from RiPLE-Design result in a single architecture representation that leaves behind any variability in quality attributes. The use of a prioritized list quality attribute scenarios and tactics do not seem to address well the problem of variability in quality attributes requirements. The idea that one solution fits all does not hold when the architect must fulfill other stakeholders' desires, such as business and marketing strategy. Sometimes, business wants exactly to benefit from the variability among quality attributes to offer different products for different markets. This gap in the main process leads to the central contribution of this work, which is to *extend RiPLE-Design guidelines to cope with variability in quality attributes requirements*. As discussed in the previous sections, the actual RiPLE-Design guidelines do not cope well with quality attribute variability.

3.4 CHAPTER SUMMARY

This chapter described the RiPLE-Design process for DSSA development. Besides showing the roles, activities, tasks, inputs and outputs of the process, it also described through a guided example how the process addresses quality attributes variability. It was discussed that the guidelines from RiPLE-Design result in a single architecture representation that leaves behind any variability in quality attributes.

The next chapter presents some enhancements proposed to overcome those limitations.

4 EXTENDING THE RIPLE-DESIGN PROCESS WITH QUALITY ATTRIBUTE VARIABILITY REALIZATION

Studies have reported the idea of variation in quality attributes (Myllärniemi et al., 2006; Niemelä; Immonen, 2007). (Myllärniemi; Männistö; et al., 2006) suggest that three points must be addressed to achieve satisfying results when the architecture presents variations in quality attributes:

(i) Specify and model varying quality attributes;

(ii) Find a design strategy for varying quality attributes;

(iii) Evaluate the architecture in order to achieve the needed variation.

Many approaches address variability modeling and specification taking into consideration non-functional features (Etxeberria et al., 2008; González-Baixauli et al., 2007; Jarzabek et al., 2006; Sinnema et al., 2004). Studies have also shown the problem of evaluating the architecture derived from a Product Line Architecture focusing on its quality attributes requirements (Olumofin; Misic, 2005). Nevertheless, only a few works focus on strategies to realize variability in the quality level, i.e., finding a design strategy for varying quality attributes, some of them are (Bosch, 2000; Hallsteinsen et al., 2003; Rossel et al., 2009), and are described next.

(Rossel et al., 2009a) describes an approach based on Model-Driven Engineering (MDE) where the PLA is seen as a set of transformations associated with the domain features. In his approach, the quality attributes requirements are also modeled as features. A derived product, built from a selection of features, can have its architecture built through the application of the earlier mentioned transformations. The variations in quality attributes requirements produce different transformation in the model and can make product architectures completely different from one another.

(Bosch, 2000) suggests the possibility of transforming quality attributes requirements into functionalities. For example, the requirement of security can be converted into login and encrypted passwords and protocols. This attempt to make non-functional requirements into functionalities does not work always. Not all quality attributes requirements can be transformed into functionality, e.g., there is no functionality that deals with performance the same way access control functionalities deal with security. It is not guaranteed that a quality attributes requirement is achieved by a specific set of functionalities. In other words, a system can have access control with highly encrypted passwords and protocol and still not be secure.

The way (Hallsteinsen et al., 2003) copes with quality attribute variability is by leaving the architecture open "on the points where the variation in requirements makes it impossible to standardize architectural decisions". The architecture documentation comprises tactics used to achieve quality attributes requirements. Tactics, in this case, are any solution proposal, guideline, design pattern, architectural style and so on. In case of variability in quality attributes requirements, more than one solution may be proposed, the variation points are made explicit and decision models are documented with the knowledge necessary to ponder about the better solution for each product to be derived.

The suggestions described in the next subsections address the three points described by (Myllärniemi; Männistö; et al., 2006). Section 4.1 and its subsections describe activities to specify and model varying quality attributes; section 4.2 and its subsections are based on the work from (Hallsteinsen et al., 2003) and describe a design strategy for varying quality attributes; Section 4.3 and its subsections describe guidelines to evaluate the architecture in order to achieve the needed variation. Finally, Section 4.4 summarizes this chapter.

4.1 REPRESENTING QUALITY ATTRIBUTE VARIABILITY

The representation of the relation between variants and design possibilities is important because, as pointed out in (Etxeberria et al., 2007), many variants represent design decisions that can have great impact on quality attributes. From the example described in (Rossel et al., 2009b) of software product line for Meshing Tools, whose feature model is reproduced in Figure 4.1: the feature Mesh Processing Distribution impacts directly in the design of the product line, as the authors show "not only a different deployment view is required to show the distributed setting, but also new components in charge of dividing the mesh among different processors and synchronizing the results are required as part of the tiers architecture too."

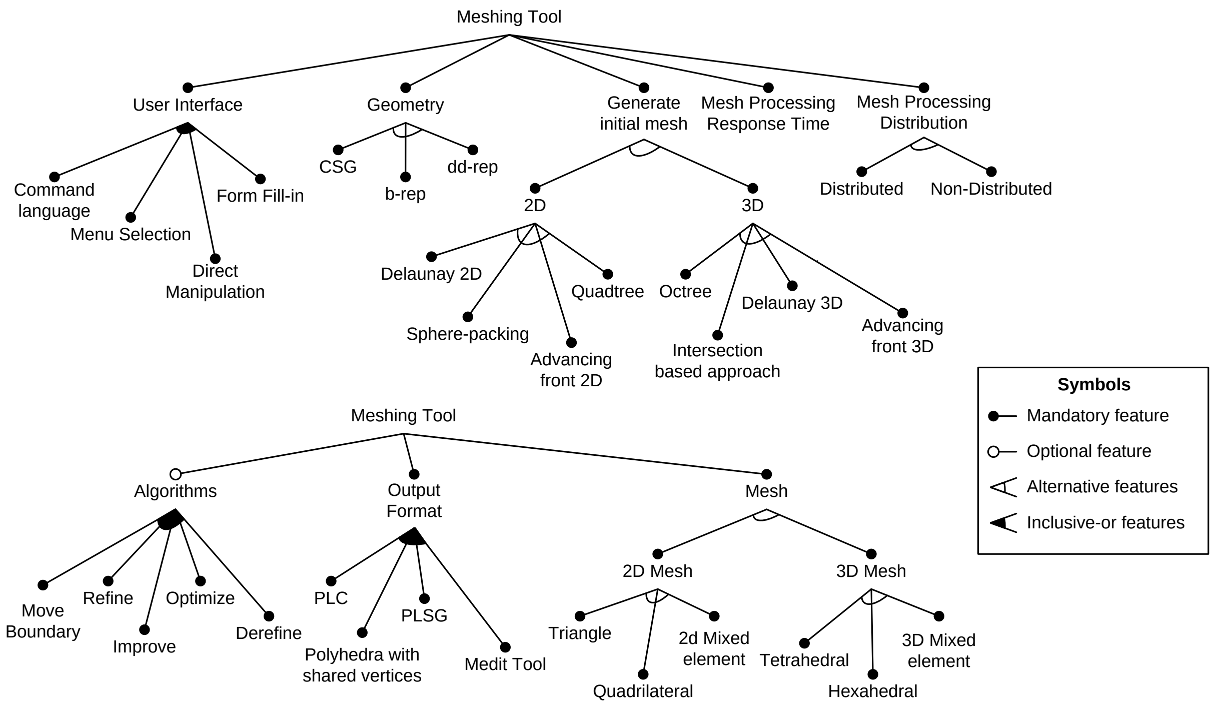

Figure 4.1 Feature Model for Meshing Tools, in (Rossel et al., 2009b)

It is even more critical when the design must deal with variable quality attributes, because, as acknowledged, quality attributes impact in each other.

Two examples of modeling methods that address varying quality attributes are F-SIG (Feature-softgoal interdependency graph) (Jarzabek et al., 2006) and COVAMOF (ConIPF Variability Modeling Framework) (Sinnema et al., 2004).

F-SIG (Feature-softgoal interdependency graph) (Jarzabek et al., 2006) was built as a framework to record design rationale in the form of interdependencies among variant features and quality attributes. It extends FORM (Kang et al., 1998) with concepts of goal-oriented analysis (Chung et al., 1999). It uses a new graph composing a feature model elements and softgoal interdependency elements. The use of softgoals helps to explicitly represent the relation between features and quality attributes. One limitation is that the F-SIG poorly supports quantitative analysis of non-functional requirements.

COVAMOF (ConIPF Variability Modeling Framework) (Sinnema et al., 2004) uses the CVV (COVAMOF Variability View), which encompasses the variability of artifacts on all layer of abstraction of the product family. This makes it possible to model the variability of the product family from features to code. As pointed out in (Etxeberria et al., 2007), a drawback of this framework is that it does not properly characterize quality attribute, what makes it hard to cope with its natural ambiguity.

(Etxeberria et al., 2007) establishes a set of requirements that are important to exist in an approach for modeling varying quality attributes. Among them are:

(i) The necessity of a mechanism for describing and explaining a quality attribute adequately;

(ii) The need to represent optionality of quality attributes;

(iii) The need to represent different levels of priority of a quality attribute.

Based on those requirements, a new activity is suggested to take advantage of existing models and document quality attributes and its possible variations among products in the product line.

4.1.1 Represent variable quality attributes in the feature model

As proposed in (Rossel et al., 2009b), it is interesting to consider not only functionality, but also quality attributes as features. The authors consider "quality attributes as part of the feature model since in several settings they may also be considered variabilities". By the definition from (Kang et al., 1990) a feature is "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems". Thus, it seems natural to consider quality attributes as features, since they are noticeable in the products.

In Figure 3.4 shown before, where the RiSE Chair Feature Model diagram is specified, we can observe *security* already described as a feature. In that case, the security feature represents some functionality that must be present in the products. Those functionalities materialize the security quality attribute. In the mentioned case, the access control feature and its children, authentication and authorization. That special case, however, does not characterize a process guideline from RiPLE. Instead, the security feature was modeled exactly to group the related functionalities. Other security concerns, e.g., confidentiality, were not mapped in the feature model diagram, even though it was an architectural concern, represented in the quality attribute scenarios.

The modified version of the activity *Identify architectural drivers* can be seen in Figure 4.2. It shows a new task, where the Domain Modeler together with the Domain Architect analyze the developed quality attribute scenarios, in order to find possible quality attribute variability.

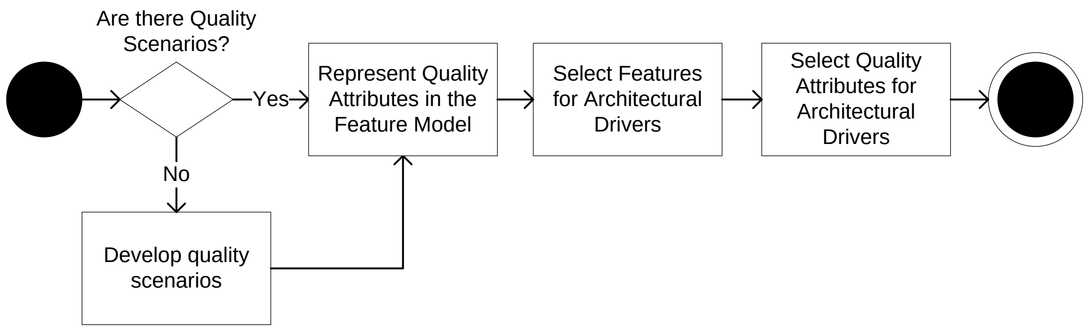
Are there Quality Scenarios?
Yes
No
Represent Quality Attributes in the Feature Model
Select Features for Architectural Drivers
Select Quality Attributes for Architectural Drivers
Develop quality scenarios

Figure 4.2 Identify architectural drivers modified, in flowchart notation

Although already suggested by RiPLE-Design, quality attribute scenarios do not seem the right option to model variability. The quality scenario representation encompasses six parts in their description. The variation can occur in any part. For example, in a variable availability scenario, not only the expected downtime, i.e., the response measure, is likely to vary, but also the response itself, namely how the system should behave after a fault occurs. This means that a single scenario would have many variation points in its descriptions, making it difficult to understand. On the other hand, the information comprised by the quality scenario is an important guide to the software architect to develop the foundations of the system, in this case, the domain architecture and a potential set of systems. This quandary leads to a hybrid solution exposed next.

The *quality attribute variability is represented in the feature model diagram*. The feature model is a central asset in the product line development, as it enables a broad view of variations and commonalities as well as possible products to derive. It is a natural step to represent the variability in this diagram. Figure 4.3 shows an example of a feature model with quality attribute variability represented.

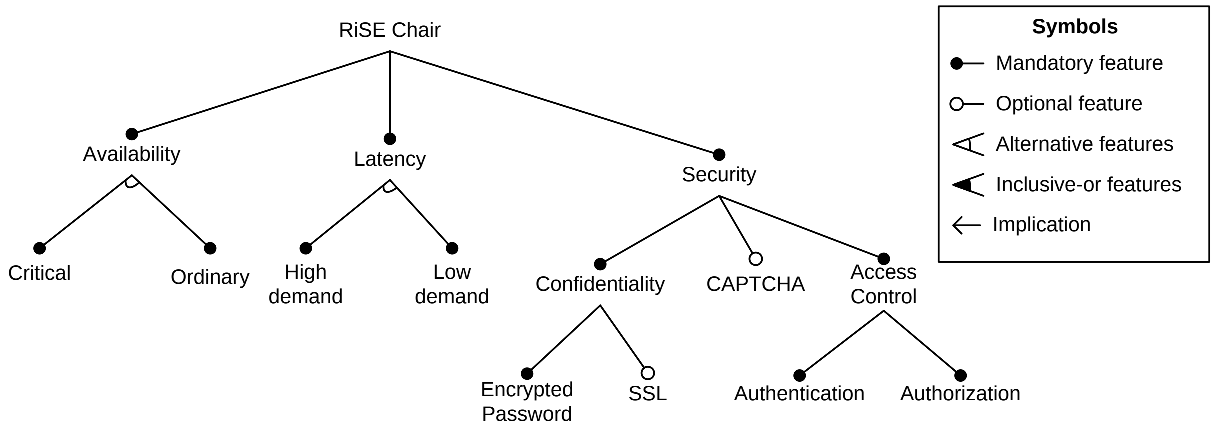

Figure 4.3 Feature model with quality attribute variability

The *quality scenarios work as side documentation to record the metrics* that accompany the attributes. The Latency quality attribute, for example, may be accompanied of lower and upper time boundaries; the Availability attribute must have the expected downtime; and so on. In order to achieve desired quality levels, the quality attributes must be complemented with the system's response measure. In the special case of quality attribute variability, the variation in those attributes is given by those numbers. This additional data is important, as indicated by (Etxeberria et al., 2007): "a model where quality attributes variability is modeled […] is indispensable to take the most adequate decision during design and derivation and get the required quality levels." The metrics can still be recorded in quality scenarios and those can be associated with features in the feature diagram. In the example of Figure 4.4, a part of a feature model diagram from some hypothetic product line is shown. Figure 4.5 and Figure 4.6 represent the quality attribute scenarios associated with each feature, and record the latency time metric related to the quality attribute.

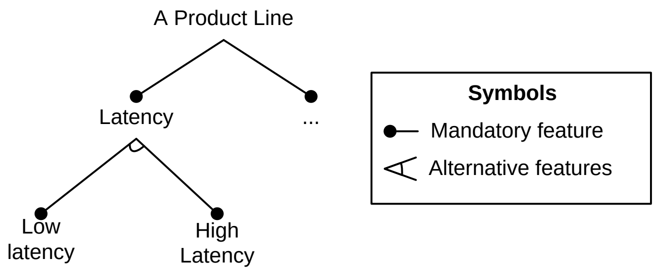

Figure 4.4 Latency quality attribute of a hypothetic product line

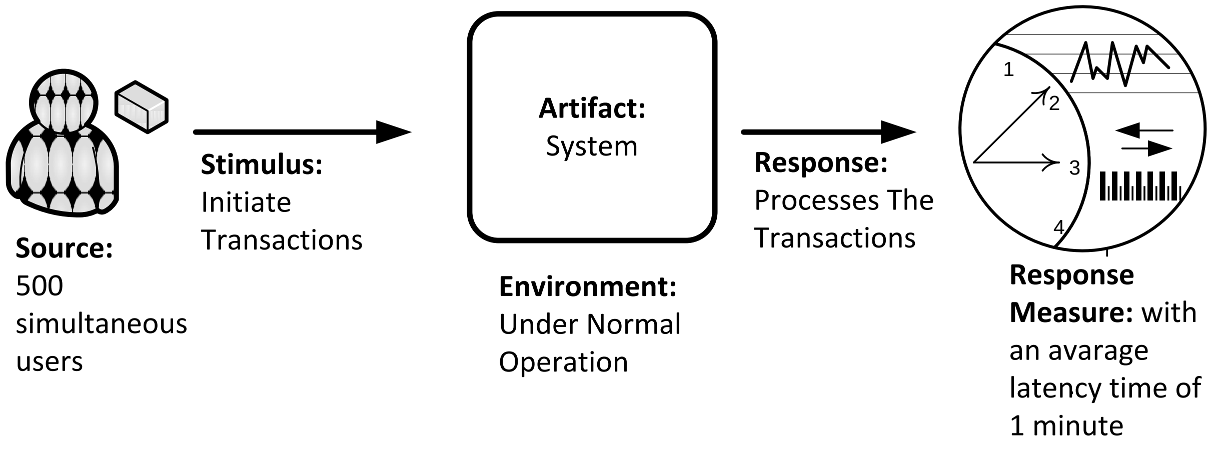
Source: 500 simultaneous users
Stimulus: Initiate Transactions
Artifact: System
Environment: Under Normal Operation
Response: Processes The Transactions
Response Measure: with an avarage latency time of 1 minute

Figure 4.5 High latency and low scalability quality scenario

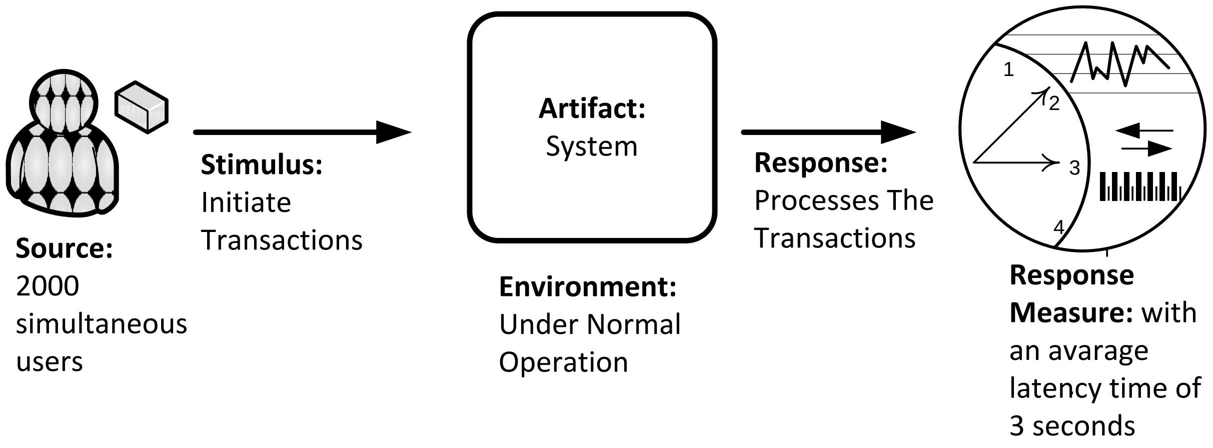
Source: 2000 simultaneous users
Stimulus: Initiate Transactions
Artifact: System
Environment: Under Normal Operation
Response: Processes The Transactions
Response Measure: with an avarage latency time of 3 seconds

Figure 4.6 Low latency and high scalability quality scenario

4.2 DESIGN STRATEGY FOR VARYING QUALITY ATTRIBUTES

“Architecture of product families has both a descriptive and a prescriptive purpose” (Hallsteinsen et al., 2003). The former has its essence in documenting the structure and behavior of the general solution provided by the domain architectures. The design of domain specific software architecture is guided by several architectural drivers, so that the main structure and behavior can be well represented and understood by others stakeholders. The prescriptive aspect focuses on constraining and alleviating the task of the application designers by providing templates for application design. Those templates consists of architectural scenarios along with implementation and configuration guidelines.

The main RiPLE-Design process output is the DSSA document, where the structure and behavior of the DSSA is documented, as well as the rationale behind the design decisions. This documentation is a descriptive aspect of the process. The mechanisms used to achieve functional variability are also documented. Particularly, the RiPLE-Design process provides guidelines to achieve functional variability through design patterns, such as *Strategy* and *Builder* (Gamma et al., 1995), and how products derived from the DSSA should be assembled in order to deliver the desired functionality, so it is viable to benefit from software reuse. This is the prescriptive side of the process.

Concerning quality attributes variability, this section describes a design strategy for varying quality attributes, the second point that should be addressed according to (Myllärniemi; Männistö; et al., 2006). This strategy focuses on the prescriptive aspect of the RiPLE process. It provides ways to standardize and document solutions to achieve recurring and varying quality attributes across the members of a software product line. The guidelines are based on (Hallsteinsen et al., 2003).

Figure 4.7 shows a modified version of the define module task, inside the Architecture Definition activity. The modified version includes changes in two steps, namely, *Select Architectural Drivers* and *Chose Architectural Styles*, and the addition of a new step called *Document Decision Guidelines*. The modification will be described next.

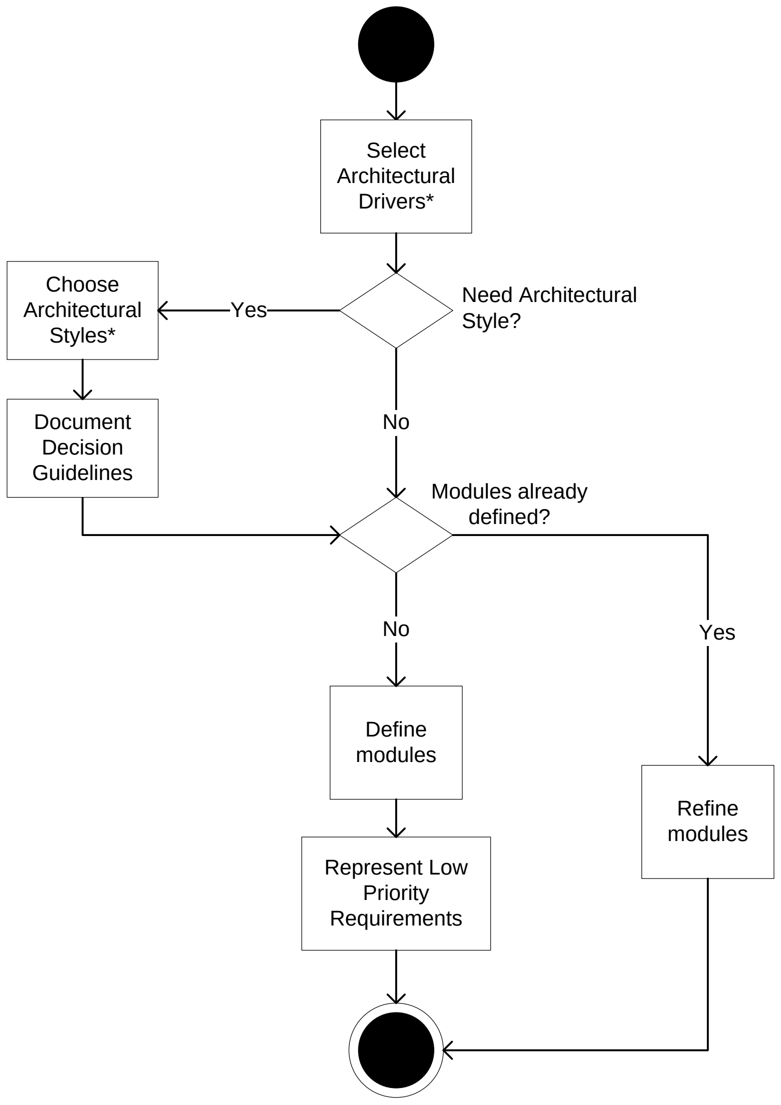
Select Architectural Drivers*
Need Architectural Style?
Yes
Choose Architectural Styles*
Document Decision Guidelines
No
Modules already defined?
No
Yes
Define modules
Represent Low Priority Requirements
Refine modules

Figure 4.7 Module definition task modified, in flowchart notation

4.2.1 Select architectural drivers

Select architectural drivers is the first task in the Module definition activity from the RiPLE-Design process. It uses the Feature Model and a ranked list of Quality Scenarios as inputs to choose the most important functional feature or quality attribute that should be considered to shape the architecture; or to choose the next most important feature or quality attribute, if a DSSA already exists.

One of the most important objectives of the software architecture is to achieve quality attributes requirements, and that is the main concern of the RiPLE-Design process. It is important to remark that to enable variability in quality attributes increases a great deal of the architecture's complexity. Nevertheless, the RiPLE-Design process does not mention quality attribute variability. Therefore, it is important to add to this step the concern about this particular case of variability. Knowing that quality attributes variability may cause high impact on the architectural definition, the proposed change in the *Select architectural drivers* task is that the domain architect enroll variable quality attributes as architectural drivers early in the process.

The selection of variable quality attributes as architectural drivers come to organize the architectural design so that complex issues are addressed first, in order to minimize the effort of introducing solutions for this kind of variability later in the process, when the architecture would have its base structure and applying patterns and tactics would be more complicated.

4.2.2 Choose Architectural Styles

In this task, the domain architect must choose architectural styles and tactics that will structure the architecture definition to fulfill a chosen architectural driver. According to (Shaw; Garlan, 1996), an architectural style "defines a *vocabulary* of components and connector types, and a set of *constraints* on how they can be combined [and used]". Architectural tactics is a term coined in (Bass et al., 2003) that represents "a design decision that influences the control of a quality attribute response." The styles are chosen so that they suit architectural needs and cope well with possible conflicts between new and previously chosen styles.

The choice for a particular architectural style or tactic is then documented in the DSSA. They are documented to solve specific design issues. The architectural documentation is meant to be a pattern language of recurring problems and solutions. Patterns can be well known architectural styles, like the Pipes and Filters style (Shaw; Garlan, 1996); established architectural patterns that concentrate on specific issues from some part of the design, such as the MCV pattern (Buschmann et al., 1996); or even particular solution built by the in-house team. Along with the proposed solution, described in the architecture document, it is important to describe the rationale behind the selection of a pattern and the possible influence it has on important quality attributes. By the end of several iterations, the DSSA documentation is composed by descriptive diagrams of structure and behavior, and rationale about the architectural styles, patterns and tactics applied.

Functional variability will continue to be addressed the same way it was before: by the selection of proper architectural styles to ensure the achievement of system variability. Quality attributes that do not involve variability can also be addressed the same way as before: architectural styles and patterns and other design strategies are chosen to meet quality attributes and balance possible conflict and trade-off among quality attributes addressed beforehand.

When the product line involves variable quality attributes, our approach is to encode architectural variation in the form of optional and alternative design strategies, as suggested in (Hallsteinsen et al., 2003). In this case, patterns represent possible paths to follow for promoting one quality attribute, perhaps over another. Optional strategies may or may not be included in application architecture. Alternative strategies are solving the same problems in different ways. The application architect may choose the one that best fits the application requirements.

Optional strategies serve to document solutions for optional quality attributes. As shown previously in Figure 3.8 and Figure 3.9, the RiSE Chair software product line presents an example of optional quality attribute. High availability is only demanded in some products. The solution discussed in Section 3.3.1 showed a single architecture that achieves the high availability demand, and assumes that it is plausible to offer a single solution, even for products that do not demand high availability. Following the proposed guidelines to deal with optional quality attributes, our DSSA would document two possible solutions. A simpler solution, used in the general case, fulfills the low availability demand represented in the quality scenario of Figure 3.8; and a second solution, more robust, where the deployment introduces a proxy server to reroute the requests, just as explained in Section 3.3.1. The simpler solution, to be applied in the general case, is shown next, in Figure 4.8.

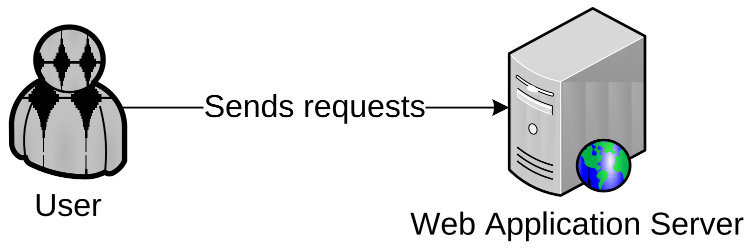

Figure 4.8 Deployment diagram. General case.

Alternative strategies are used to document quality attributes variability that involve trade-off among attributes. In the example shown in Section 3.3.2, security and latency were in clear opposition. Figure 3.12 shows the performance scenario for the class of products with high performance demands and Figure 3.13 correspond to other class of products that demand higher security.

In the previous example, RiPLE-Design process was followed and the conclusion was that the Domain Architect and the Domain Manager should come to an agreement where new response measured should be established so that both quality attributes could coexist in a single architectural solution.

Following the new guideline, the architect can apply different tactics and propose different architectural styles, respecting the fact that both high security and low latency must not coexist. The DSSA should then document two alternative solutions: one for high security, where the latency concerns will be in second plane; and another for low latency, where security concerns will not play such an important role.

Since the class of products demanding high confidentiality do not demand low latency, it is plausible to offer a solution that deals with confidentiality by applying the suggested tactics and patterns discussed in Section 3.3.2, that is applying the *Maintain data confidentiality* tactic, under the form of an encryption link implemented by a Secure Sockets Layer (SSL), very common in web applications. Without any concern about low latency, a simpler architecture can fulfill the expected quality attribute response measure, as shown in Figure 4.9.

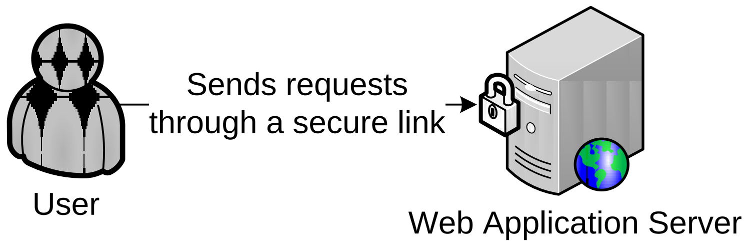

Figure 4.9 Deployment diagram, secure link.

Low latency demands, a security concerns can be implemented as it was suggested before, with a load balancing server, described in Figure 3.15. Aside from that suggestion, the proposed solution would benefit from the known *Reduce computational overhead* tactic, as it would not use the encryption layer, making it easier to achieve the latency response measure expectations.

Quality attribute variability is treated externally to the components, i.e. it is represented in the component connectors. Functional variability can be internal to the component. Functional variability has its way of implementation and documentation. The use of design patterns, for example, is suggested in the RiPLE-Design process. To deal with variable quality attributes, on the other hand, the same patterns may not apply, especially when the variant solutions promote a large change in the architecture. The suggestion is to introduce variation points also in the DSSA documentation. Optional strategies can be documented as suggested solution to deal with optional quality attributes, and alternative strategies can serve alternative quality attributes.

4.2.3 Document decision guidelines

In order to improve and alleviate the work of application designers, there must be some decision guidelines to help them evaluate the consequences of each decision. As there will be many possible solutions for the product architecture, each path must be well documented, the foreseen impacts well described and measured when possible.

This task is helped by the use of quality attribute scenarios. Their stimulus-response structure can be associated with design strategies. This association works very well as documented solutions for the architecture. For a given stimulus, the impact of each design strategy can also be observable in its response. Following the suggestions of (Hallsteinsen et al., 2003), all those aspects can be put together in a summarized form, to help further designer tasks. An example of such summary can be found in Table 4-1, and exemplifies the case of optional availability.

Recovery strategies		**Strategy**	**Affected quality attributes**
Stimulus	**Response**		
Internal component failure	Until 8 hours of downtime	Email notification, system reboot	Availability, recoverability
	Few Minutes of downtime	SMS notification, proxy router server, Backup system	

Table 4-1 Decision guideline for optional availability

If we apply this approach in the RiSE Chair project, in the case of variability in availability, the solution would indicate guidelines at the deployment view as well as special error notification components, for example, introducing notification per SMS. In this case, more critical products would be instantiated with a more robust recovery plan, and faster error notification. A more complete spare software platform could be made ready in case of higher level of availability. Less critical products would not need to afford expensive recover plans and platforms. The solution for the low availability demand is shown in Figure 4.8, the more complex solution, describing a complete spare platform is shown in Figure 3.10 and Figure 3.11.

In the case of trade-off between security and performance, two alternative solutions could be proposed. The summarized documentation of this case is shown in Table 4-2. The first solution introduces the secure protocols and time consuming encryption of data. It also leaves the concerns about latency aside, as they have lower priority. In an alternative solution, the *Reduce Computational Overhead* (Bass et al., 2003) tactic would be applied, by removing layers and intermediaries, along with a load balancer proxy server, as discussed before. As security would have a lower priority, the security protocol would be the first architecture layer to be removed. Further shrinking of the layered structure could damage the solution modifiability and harm the other architectural drivers.

Security/performance strategies		**Strategy**	**Affected quality attributes**
Stimulus	**Response**		
Eavesdropper sniffs the network; loose latency requirements	Data encrypted and unreadable, average latency of 1 minute	SSL	Security, performance
Low latency, high user load	Average latency of 3 seconds	Load balancer, no SSL	

Table 4-2 Decision guideline for optional availability

Both solutions should be described in the architecture document with rationale about when to follow each strategy. The DSSA documentation would describe that SSL encryption are time consuming and could be left aside in the latter solution, as well as a load balancer would be necessary to attend a possible high user demand. The former solution would describe that SSL encryption would serve well the quality attributes response measure expectations as it is a technology of known efficiency against this kinds of attack.

Component specification must still go together with design strategy descriptions. They are still a crucial piece of documentation of the architecture. In RiPLE-Design, component definitions come later in the process, after the module definition step. Components can be derived from domain features and from domain use cases. When the components are defined from use cases, the guidelines proposed by the RiDE process from (Almeida, 2007) are followed. When the domain features are used, the resulting components represent a feature or a group of features that can be implemented by a single component. The variability inherent to a group of features is represented internally or externally in the component using UML component diagrams and stereotypes. High level variant features lead normally to components with external variation, while features that represent implementation selection and details lead to internally variant components. As the fine grain structural representation is normally needed, classes that will implement each component must be defined along with the technique chosen to implement the functional variability.

A well-documented DSSA would count with structural and behavioral diagrams, as well as decision guidelines for each variation point introduced in the architecture by variable quality attributes. In order to derive family members, the application designer must:

(i) Resolve variation points based on the needs of the application architecture. The needs here should recall to quality attributes and the specialized quality model is then used next. That is, the quality scenarios that are important for each member the architect wants to derive;

(ii) Patterns are selected following the decision guidelines. The proposed solution can be verified further using the already documented quality attribute scenarios described in the decision model;

(iii) Components specifications that match the design strategies can then be selected.

4.3 EVALUATE THE ARCHITECTURE IN ORDER TO ACHIEVE THE NEEDED VARIATION

Several types of software architecture evaluation methods exist: some are like SAAM (Clements et al., 2006), which is generic, but is used to evaluate one quality attribute at a time; the already mentioned ATAM is an example of trade-off analysis methods, that evaluate many attributes; and there are some specific for only one attribute, such as SALUTA (Folmer et al., 2003) for usability. Such techniques are essentially made for single system development and cannot be directly applied to product line development. They are not proper to evaluate a PLA, even though they could be used to evaluate each product in the line.

In a product line, different members may require different levels of a quality attribute requirement. This variability leads quality evaluation in software product lines to be much more complicated than in single-system (Etxeberria; Sagardui, 2008). (Olumofin; Misic, 2005) advises that "traditional system development practices for single products cannot be directly applied to product line development". The author shows that architecture-centric evaluation methods are all generally made for single-systems and product line specific evaluation methods normally evaluate only the resulting products.

The evaluation of every product in the product line can be valid, but seems too expensive. Nevertheless, it can be possible to make them shorter and cheaper as (Clements; Northrop, 2001) points out:

> Product architecture evaluation is a variation of the product-line architecture evaluation as the product architecture is a variation of the product-line architecture and the extent to which product evaluation is a separate, dedicated evaluation depends on the extent to which the product architecture differs in quality-attribute-affecting ways from the product-line architecture.

Figure 4.10 shows two new activities added to the main RiPLE-Design process. First, to *Evaluate existing architectures*, in order to detect problematic issues and risks points and perhaps choose an existing architectural approaches to serve as basis of the future DSSA. As the last activity in the process, *Evaluate the DSSA*, so the proposed architectural solution can be assessed[4] about its ability to enable variability and fulfill other quality attribute requirements. A third step is also suggested: to *Evaluate derived product architectures*. It is not shown in Figure 4.10, since it refers to the core asset development part of the RiPLE process, and the evaluation of derived products would be part of the product development part of the process.

[4] In the discussions that follow, the terms assessment, evaluation, and analysis will be used interchangeably.

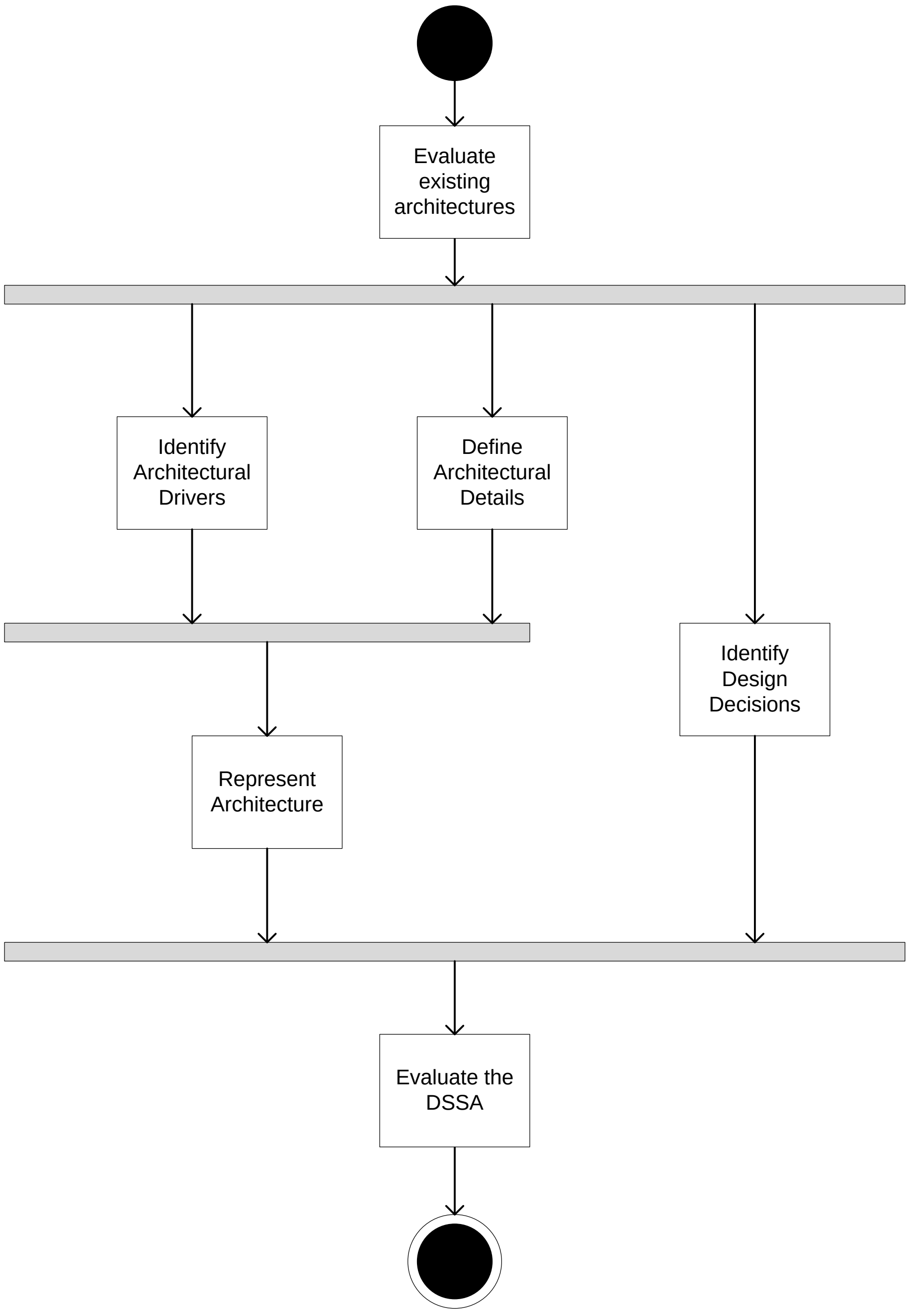
Evaluate existing architectures
Identify Architectural Drivers
Define Architectural Details
Identify Design Decisions
Represent Architecture
Evaluate the DSSA

Figure 4.10 RiPLE-Design including Evaluation steps, in flowchart notation

4.3.1 Evaluate existing architectures

Concerning architecture evaluation, the first change suggested to RiPLE-Design is illustrated in the upper region of Figure 4.10: to *Evaluate existing architectures*, in order to assess its strengths, and apply them to the new DSSA; or its weaknesses and avoid them. This evaluation should follow the Architecture Trade-off Analysis Method (ATAM) (Kazman et al., 2000).

ATAM is a scenario-based approach for single systems architecture evaluation. As discussed in (Kazman et al., 2000), "when evaluating an architecture using the ATAM, the goal is to understand the consequences of architectural decisions with respect to the quality attribute requirements of the system." It fits very well the guidelines already proposed in RiPLE-Design, which is also a scenario-based approach, aiming to fulfill quality attribute requirements of a software product line. Scenarios help to remove the ambiguity in the description of a quality attribute, as stated by (Bass et al., 2002). They are used as a common interface between stakeholders and the design team. The simple language helps to bring different stakeholders together and discuss in the same terms.

In the case of assessing existing architectures to serve as basis for a DSSA, the evaluation must focuses on the scenarios that may compose the domain quality attributes, and assess how the existing architecture has dealt with those scenarios. The main outputs of this activity are a set of domain scenarios, and the architectural decisions under the form of *Risks*, *Non-Risks*, *Sensitivity points* and *Trade-off points*.

Risks are potentially problematical architectural decisions, a possible weakness to be avoided in the DSSA. *Non-risks* are fine decisions that rely on assumptions that are frequently implicit in the architecture, and remain non-risk as long as the assumptions do not change. The domain architect must ensure the good design decision made in prior application architectures still apply in the DSSA.

A *sensitivity point* is a property of one or more components (and/or component relationships) that is critical for achieving a particular quality attribute response. For example, the level of confidentiality in a virtual private network might be sensitive to the number of bits of encryption. Sensitivity points tell a designer or analyst where to focus attention when trying to understand the achievement of a quality goal, as discussed in (Kazman et al., 2000).

A *trade-off point* is a property that affects more than one attribute and is a sensitivity point for more than one attribute. As in the example shown by (Kazman et al., 2000), changing the level of encryption could have a significant impact on both security and performance.

As suggested by (Etxeberria; Sagardui, 2005), evaluation can and should be held in different moments in software product line development. At core asset development, evaluation can be useful to detect problematic issues and risks points or compare software architecture candidates to select the one that best supports the required quality attributes. Evaluation can also be held before developing the reference architecture in order to use existing architectures as basis for the product line.

4.3.2 Evaluate the DSSA

Existing architecture evaluation approaches focus on single product architectures and offer little support for the particular characteristics of product line architectures. Architecture-based development of software product lines, as suggested in the RiPLE-Design process, requires the appropriate architecture evaluation methods specifically addressing the quality of software product line architectures.

Product line architectures bring challenges which are not present in single product architectures, and those differences make the assessment of such architectures rather difficult.

Quality attribute scenarios are more numerous than in single systems, they are context dependent (some of them apply to the whole product line architecture, others to the core assets and some specific product architecture, and yet others to one or more product architecture only). In addition, three distinct forms of quality attributes scenarios may be identified. Mandatory scenarios apply to all products alike. The alternative and optional scenarios are product-specific and should only apply to the analysis of individual product architectures.

Another product-line specific characteristic is that there are two level of architectural abstraction where an evaluation can be performed (software product-line architecture and derived product architectures). To assess all the instances of the product-line may not be worthwhile due to the high cost (Etxeberria; Sagardui, 2005).

Organizational factors can also influence product-line architecture evaluations: a PLA involves more stakeholders than a single system because the scope is much larger than the one of single-product architecture (Etxeberria; Sagardui, 2005).

PuLSE-DSSA also defines a process for evaluation of reference architectures (Anastasopoules et al., 2000). However, it has limited applicability as the evaluation process is bound to the PuLSE methodology, and there is no tradeoff analysis as it iteratively defines evaluation criteria per scenario.

The QADA process, defined in (Matinlassi et al., 2002), includes architecture evaluation guidelines. However, the approach does not provide guidelines for product architectures evaluation.

Other methods focus on a limited set of quality attributes for the assessment, which limits their applicability in practice (Auerswald et al., 2001; Lassing, 2002; Maccari, 2002).

A comprehensive, albeit not too rigorous overview of a number of product line architecture evaluation techniques can be found in (Etxeberria; Sagardui, 2005).

A number of product line architecture assessment methods have been proposed. Nonetheless, the majority of them fall short of addressing the challenges outlined above in a comprehensive and efficient manner. In order to address the outlined challenges, the Holistic Product Line Architecture Assessment (HoPLAA) method was proposed by (Olumofin; Misic, 2005).

HoPLAA starts from the already mentioned ATAM (Kazman et al., 2000) and its trademark feature, the trade-off analysis between quality attributes. HoPLAA extends the ATAM with the qualitative analytical treatment of variation points and the context-dependent generation, classification, and prioritization of quality attributes scenarios. The notions that provide the foundation of the HoPLAA method are stated in (Olumofin; Misic, 2005), and show that the method was developed to cope well with quality attribute variability:

> The problem of evaluating product line architectures must be approached by considering not only the quality attributes common to the family of systems, but also those specific to some members only, and their interrelationships. It should be noted that individual product architectures may require a different prioritization of the quality goals common to the product line architecture, and they may even be associated with quality goals which are not present in other members of the product line.

The HoPLAA method addresses the evaluation of software product line architectures in an integrated approach with two analysis steps for the DSSA and the Product Architecture (PA). The integrated approach simplifies the analysis of quality attributes and their interactions, since the architectural decisions made right in the product line architecture creation impact individual product architectures derivations. The first stage of the method focuses on the DSSA evaluation, and will be described next. The second stage targets individual PA evaluation, and will be the focus of next section.

Besides the traditional outputs obtained through an ATAM evaluation, the HoPLAA method produces outputs with more importance on the *evolvability points* and *evolvability constraints*. An *evolvability point* is an area of the DSSA which is a sensitivity point, and which contains at least one variation point. *Sensitivity points* are design decisions that affect one or more quality attributes, as in (Kazman et al., 2000). As pointed out by (Olumofin; Misic, 2005), the architectural decisions made in the DSSA, and found to be the sensitivity points to one or more quality attributes, continue to be valid for individual product architectures. *Evolvability constraints* are guidelines that accompany each evolvability point as special treatment for variation points that could alter quality. The constraints guide the subsequent PA design decisions and evaluation, so that they do not invalidate quality attribute requirements already addressed in the product line architecture.

The HoPLAA seems a proper method for evaluating PLA and its derived product architectures as it assesses the architectures in different stages and evaluate important aspects of both general system development and specific product line issues.

The activity *Evaluate the DSSA* adapts the guidelines proposed in the Holistic Product Line Architecture Assessment (HoPLAA) method, proposed by (Olumofin; Misic, 2005). The adaptation comprises the use of already generated quality scenarios during the evaluation process.

This activity should be performed by the Domain architect together with the Domain manager. When possible, external architects can be added to the activity in order to improve the evaluation process.

The first stage of HoPLAA, presented in (Olumofin; Misic, 2005), applied in the *Evaluate the DSSA* activity, consists in the seven steps reproduced next.

1. **Present the HoPLAA Stage I**. To present an overview of the method and the activities of both stages;

2. **Present the product line architectural drivers**. The architectural drivers chosen to guide the architecture development. Motivating business needs, scope definition, and common and variable functionalities and quality attributes. Those drivers were already defined in the *Architectural Drivers Identification* activity;

3. **Present the product line architecture**. The DSSA is presented by the Domain Architect;

4. **Identify architectural approaches**. Architectural approaches used in the architecture are identified by the evaluation team. The list of approaches is documented but not analyzed. This step serves the purpose of constraining the set of architectural approaches in order to maintain consistency in the use of architectural approaches throughout the design of the DSSA and the individual PAs.

5. **Classify, and prioritize quality attribute scenarios**. Quality scenarios that were already defined in the RiPLE-Design process will be used again for evaluation purposes. The mandatory quality scenarios, common to all products, are verified in the current stage, while alternative and optional scenarios will only receive special treatment later, when specific product architectures are to be evaluated. This way, only quality attributes concerns common to every product are checked to be addressed by the DSSA. Mandatory quality scenarios will not be verified again thoroughly in the second stage of the evaluation process. It is mandatory that the quality attribute of variability is analyzed at this stage. Since large-scale reuse, the purpose of the product line approach, is best realized when the architecture fully supports variability. Quality scenarios are ranked using three indexes: *Generality*, *Significance* and *Cost*, each of which is assigned a value in the enumeration [10, 20, 30], respectively for Low (L), Medium (M), and High

(H). Generality may be mandatory, alternative or optional, with values 30, 20 and 10, respectively. Significance denotes the importance of the quality attribute scenario to the business driver. Cost represents the effort involved in enhancing the architecture to provide the right responses to the scenario. Once assigned to individual scenarios, the values of indexes are added up with even weight so as to prioritize the list of scenarios,. The most important attribute concerns shared among all products in the product line will characterize the scenarios on top of the list.

6. **Analyze architectural approaches and scenarios**. High priority scenarios from the prior step are analyzed to obtain a set of risks, non-risks, sensitivity points and trade-off points, as well as evolvability points. Guidelines are associated with evolvability points to constrain subsequent changes that try to deviate the architecture from the quality attribute requirements already addressed, or to guide future analysis of product architectures.

7. **Present results**. A report is prepared containing architectural approaches, quality scenarios, product-specific scenarios identified, areas of risk in the DSSA, non-risks, sensitivity points, trade-offs, evolvability points and evolvability guidelines. A report template is provided in Appendix B.

The evaluation process is largely helped by the architectural development activities realized previously. Scenarios developed to guide the architectural definition can be reused during the fifth step of the DSSA evaluation. Architectural approaches are already documented during the *Choose architectural style* activity and evolvability guidelines are documented during the *Document decision guidelines*.

Architecture evaluation is an expensive exercise. The seven steps proposed by (Olumofin; Misic, 2005), during the first stage of the method are two less than the number of steps in a comparable ATAM evaluation. It is expected that important scenarios generated/brainstormed during the fifth step would have sufficed.

In terms of efficiency, Stage I of the HoPLAA analysis should take less time than the equivalent ATAM analysis of the core PLA, since some steps have been merged, and the analysis of some scenarios is deferred until Stage II.

4.3.3 Evaluate derived product architectures

As suggested by (Etxeberria; Sagardui, 2005), architecture evaluation for SPL should be used to evaluate the design of the product line architecture as well as the resulting individual product architectures of the line. At product development, the architectural conformance to the reference architecture can be assured. During product derivation, the impact of architectural decisions in quality attributes can be analyzed.

In order to advance in the direction of a quality aware process that deals well with quality attribute variability, it is important to ensure that derived product architectures are also evaluated. The RiPLE-Design processes focuses on the core asset development phase of the software product line life cycle, nevertheless, the activity of *Evaluate derived product architectures* is suggested.

The evaluation of individual product architectures adapts the second stage of the HoPLAA method, from (Olumofin; Misic, 2005). The motivation to adapt the second stage of the HoPLAA method is that it focuses not only in single-systems architectures, but in product architectures in the context of the DSSA that derived it. This focus lessens the effort of assessing product architectures and justifies the two-staged approach for evaluating DSSA and product specific architectures. It consists in the following steps.

1. **Present the HoPLAA Stage II**. To present an overview of the HoPLAA and the activities of the second stage.

2. **Present architectural drivers**. A short overview of the DSSA and the driving requirements for the particular product architecture being evaluated. Variable features, including functional and quality attribute requirements, should also be described.

3. **Present the product architecture**. The focus must be on the areas of the architecture that have been improved through the realization of variation points.

4. **Identify architectural approaches**. The Domain Architect identifies and documents new or different architectural approaches used in the product architecture. The approaches will be analyzed in a later step. If a newer architectural approach, not foreseen in the DSSA, is used to realize a variation point, it must be accompanied by a rationale.

5. **Prioritize quality attribute scenarios**. Quality attribute scenarios that are specific to this product are reproduced for evaluation purposes. New product-specific scenarios can also be generated. All the scenarios are then prioritized the same way as in Stage I. This prioritization is important because different products may have different priorities.

6. **Analyze architectural approaches**. Two analyses must be performed by the architect: The architect must demonstrate how quality scenarios relative to the whole DSSA are not precluded in the product architecture design and also how the architecture realizes quality goals that are specific to the product being analyzed. Concerning the first analysis, when design decisions do not violate the evolvability guidelines, quality attributes continue to be satisfied. If the contrary is true, risks have been introduced in the product architecture and may prevent the achievement of quality attribute requirements. This analysis is especially important when the product architecture needed to change some DSSA guidelines. The second analysis consists of obtaining the architectural risks, non-risks, sensitivity points, and trade-off points for the product architecture.

7. **Present results**. An evaluation report, similar to the one described in Stage I, is prepared. The main difference is that this report does not include evolvability points or evolvability guidelines, since product specific architectures do not need to support variability. A report template is provided in Appendix Appendix B.

Stage II of HoPLAA evaluates the product-specific architectures. Some of the scenarios created in the previous stage are reused and the evolvability points and constraints are used as guidelines to lead the analysis into those areas of the product architecture that realize variation points and may have change. Figure 4.11 shows the inputs and outputs of the Evaluate derived product architectures step.

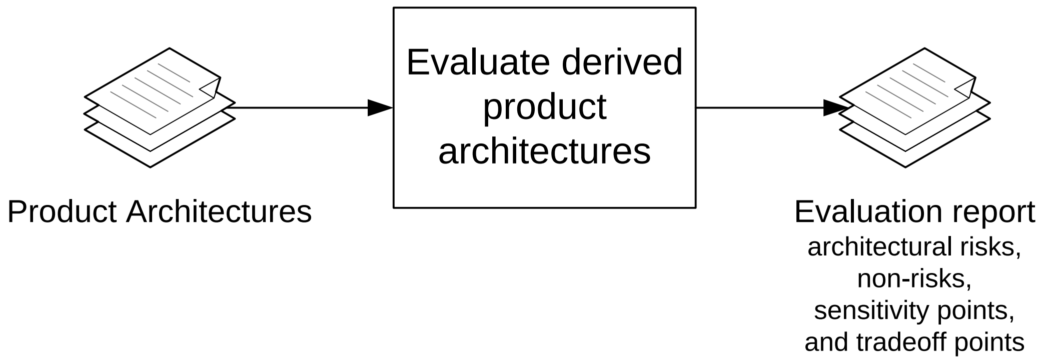

Figure 4.11 Evaluate derived product architectures

With the two staged approach, we can suppose that the HoPLAA analysis of a PLA will take more time than the equivalent ATAM analysis of the core PLA architecture alone, but less time than would be needed to perform the ATAM analysis to each individual PA separately. In either case, HoPLAA would out-perform ATAM in terms of comprehensiveness and effectiveness of the analysis.

4.4 CHAPTER SUMMARY

From the deficiencies of the RiPLE-Design approach concerning the treatment of quality attribute variability, this chapter presented enhancements following the three pillars suggested by (Myllärniemi; Männistö; et al., 2006): the specification and modeling of varying quality attributes, design strategies for varying quality attributes, and architectural evaluation.

The specification modeling of varying quality attributes is done with the aid of the feature model diagram, which is a central asset in the RiPLE process. The design of variable quality attributes adapts the approach from (Hallsteinsen et al., 2003) and documents alternative and optional architectural patterns and tactics in order to handle variation points in quality attribute requirements. The evaluation of product line architectures in order to guarantee variation achievement is done in two phases based on the Holistic Product Line Architecture Assessment (HoPLAA) method, proposed by (Olumofin; Misic, 2005).

In the next chapter, it will be presented an experimental study with the Enhanced RiPLE-Design performed with the purpose of characterizing and refining it.

5 THE EXPERIMENTAL STUDY

Software can be found in broad range of products, from televisions, to missiles and space shuttles. This means that great quantity software has been developed and is being developed (Wohlin et al., 2000). However, software development is a complex task that involves much creativity. Several problems can run into software development, e.g., missing functionalities, poor quality and missed deadlines. Managers are increasingly focusing on process improvement in the software development area, with the intention of reducing the cost of development, testing and maintenance over the life of the application (Chidamber; Kemerer, 1994a).

As discussed in (Basili, 1996), "progress in any discipline depends on our ability to understand the basic units necessary to solve a problem." In the discipline of software engineering, empirical studies, like surveys, case studies and experiments plays an important role in the build of this understanding. Experimentation provides a systematic, disciplined, quantifiable and controlled way of evaluating human-based activities (Wohlin et al., 2000).

Three classical forms of experimental studies are surveys, case studies and formal experiments (Kitchenham et al., 1995). Surveys aims at the development of generalized solutions. They focus on large groups and try to draw conclusions based on a wide range of variables. Formal experiments have as evident characteristics the carefully controlled environment, appropriate levels of replication, randomized selection of experimental subjects and objects. Case studies are observational, and focus on single, typical projects. Case studies are easier to plan than formal experiments, but are harder to interpret and difficult to generalize.

In this sense, this chapter presents a formal experimental study performed with the purpose of characterizing the efficacy, understanding and applicability of the proposed enhancements to the RiPLE-Design process in the context of software product line projects. The process defined in (Wohlin et al., 2000) was used to define, plan and execute a formal experiment. In order to consider SPL problems, the Travel Reservation domain, which contains functional variability as well as quality attribute variability requirements, was the project used in the study (Snell, 2002; Segura et al., 2007).

The remainder of this chapter is organized as follows: Section 5.1 presents essential information to understand the terminology of experimental studies; Section 5.2 presents the definition, planning, operation, analysis and interpretation of the experimental study with the Enhanced RiPLE-Design, Section 5.3 presents the conclusions, and Section 5.4 presents the lessons learned with the experimental study. Finally, Section 5.5 concludes this chapter with its summary.

5.1 EXPERIMENTAL TERMINOLOGY

A controlled experiment is an investigation of a testable hypothesis where one or more *independent variables* are manipulated to measure their effect on one or more *dependent variables* (Easterbrook et al., 2008). In the study of a new development method on the productivity of personnel, the dependent variable is the productivity itself. Independent variables are the development method, tool support, the environment in which the experiment is conducted and the experience of the subject, for example.

Independent variables changed during an experimental study are also called *factors*. A particular value of a factor is called a *treatment*. An experiment that assesses the changing of a development method can analyze two treatments of the factor: the old method and the new one.

The treatments are being applied to the combination of *objects* and *subjects*. In software experiments, experimental subjects are individuals or groups (teams) who use a method or tool and *objects* may be the programs, algorithms, or problems to which the methods or tools are applied (Kitchenham et al., 1995).

An experiment consists of a set of *tests* where each test is a combination of treatment, subject and object (Wohlin et al., 2000). For example, a test can be that person N (subject) uses the new development method (treatment) for developing program A (object).

5.2 THE EXPERIMENTAL STUDY

The experimental study was performed following the process defined in (Wohlin et al., 2000), 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 study. The process divides the experiment process into the following main activities. Firstly, the study is **contextualized** in terms of the problem, objective and goals. The **evaluation hypotheses** are defined next, and the ways to evaluate the study. 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 report** activity.

The next sections present the contextualization, planning, operation, analysis and report of the experiment. The experiment presentation and package is represented with this chapter.

5.2.1 Contextualization

This section determines the foundation of the experiment. If the foundation is not properly laid, rework may be required, or even worse, the experiment cannot be used 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. (Basili et al., 1994).

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):

- Conceptual Level (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;

- Operational Level (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;
- Quantitative Level (Metric): A set of data is associated with every question in order to answer it in a quantitative way. Metrics can be *Objective*, if they depend only on the object that is being measured and not on the viewpoint from which they are taken; or *Subjective,* if they depend on both the object that is being measured and the viewpoint from which they are taken.

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

5.2.1.1 Goal

The goal of this experimental study is *to analyze the Enhanced RiPLE-Design process* for the purpose of *characterization* with respect to the *quality of the generated architecture, process understandability and applicability* from the point of view of *researcher* in the context of *software product line projects with variable quality attributes requirements*.

The goal, questions and metrics will serve the purpose of analyzing quality aspects related to the Modifiability of the generated architecture. They will also aid a superficial assessment on how functional and quality attribute requirements were addressed by the subjects as a result of the experiment procedure. In order to analyze other quality aspects of the architecture, such as the achievement of any other quality attributes, further studies will be required.

Other process aspects such as its overhead, the process scalability for large teams and the adequacy to small teams, will also be left out of this assessment.

Although the Enhancements to the RiPLE-Design process also provide guidelines for DSSA evaluation, this part will be left out of the scope of the experimental study due to time and resource constraints.

5.2.1.2 Questions

Q1. Does the RiPLE-Design aid architects to generate components with loose coupling?

Q2. Does the RiPLE-Design aid architects to generate components with low instability?

Q3. Does the proposed architecture address the functional requirements?

Q4. Does the proposed architecture address functional variability?

Q5. Does the proposed architecture address the quality attributes?

Q6. Does the proposed architecture address quality attribute variability?

Q7. Do the subjects have difficulties to understand the RiPLE-Design enhancements?

Q8. Do the subjects have difficulties to apply the RiPLE-Design enhancements in practice?

5.2.1.3 Metrics

M1. Coupling between components (CBC): as defined in (Perepletchikov et al., 2007), Coupling is a measure of the extent to which interdependencies exist between software modules. Since components communicate with each other through defined interfaces, the Coupling Between Object Classes (CBO) defined in (Chidamber; Kemerer, 1994b), is applicable for components as well. The CBO relates to the notion that an object is coupled to another object if one of them acts on the other. In the context of components, it means that CBC is defined as the count of component a given component calls operations on. This definition is similar to the Direct Component Coupling metric (DCMCM) defined in (Chen et al., 2009). It is a function defined as:

CBC(c) = number of components used by another component (c), where (c) is a component of a given system.

This coupling metric has range [0, n], where n is the number of components different from (c) of a given system. CBC = 0 indicates a totally loosely coupled component, and CBC = n indicates a maximally coupled component.

This metric helps to evaluate the quality of a given architecture as excessive coupling between components is detrimental to modular design and prevents reuse. The more independent a component is, the easier it is to reuse it in another application (Chidamber; Kemerer, 1994b). Loosely coupled components are crucial in the context of product line architecture.

Maintenance is also more difficult in largely coupled architectures, because the larger the number of couples, the higher the sensitivity to changes in other parts of the design.

This measure is also useful to determine how complex the testing of various parts of a design is likely to be. The higher the component coupling, the more rigorous the testing needs to be.

M2. Component Instability (CI): the interdependence of the subsystems within a design is what makes it rigid, fragile and difficult to reuse (Martin, 2002). A rigid design cannot be easily changed. This rigidity is due to the fact that a single change to heavily interdependent software begins a cascade of changes in dependent modules. The impact of the change cannot be easily estimated because the extent of that cascade of changes.

(Martin, 2002) defines stability as a measure of the difficulty in changing a module. It is related to the amount of interrelated modules some module has. The *Instability* of a component is, then:

CI (c)=Ce/(Ca+Ce), where Ca is the number of components that depend upon the component (c), and Ce is the number of components that the component (c) depends upon.

The Component instability metric has range [0, 1], where CI = 0 indicates a maximally stable component and CI = 1 indicates a totally instable component (Martin, 2002).

M3. Addressed Functional requirements (AFD): this metric will be used to identify whether the proposed architectural solution was able to address the functional requirements listed in the experiment task. As any other software architecture, the DSSA must address the domain requirement. This information will be captured after analyzing the DSSA documentation produced by the subjects. The measure is defined as:

AFD = % of functional requirements addressed by the architecture

M4. Addressed Functional Variability (AFV): this metric will be used to identify whether the proposed architectural solution was able to address the functional variability described in the experiment task. The main purpose of a software product line is to achieve variability (Bosch, 2000). This information will be captured after analyzing the DSSA documentation produced by the subjects against the feature model described in the experiment task. The measure is defined as:

AFV = % of functional variation points addressed by the architecture

M5. Addressed Quality Attributes (AQA): this metric will be used to identify whether the proposed architectural solution was able to address the quality attributes listed in the experiment task. As any other software architecture, the DSSA must address quality attribute requirements (Bass et al., 2003). This information will be captured after analyzing the DSSA documentation produced by the subjects. The measure is defined as:

AQA = % of quality attributes addressed by the architecture

M6. Addressed Variable Quality Attributes (AVQA): this metric will be used to identify whether the proposed architectural solution was able to address the quality attribute variability described in the experiment task. Aside from functional variability, quality attribute variability is an important issue that has been neglected for a long time (Etxeberria et al., 2008). The main purpose of the Enhanced RiPLE-Design process is to help domain architects to produce architectures that can enable quality attribute variability. This information will be captured after analyzing the DSSA documentation produced by the subjects. The measure is defined as:

AVQA = % of non-functional variation points addressed by the architecture

M7. Problems (MP): This metric will be used to identify possible misunderstanding problems concerning RiPLE-Design documentation. It is necessary to identify and analyze the difficulties found by the subjects learning the approach. 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 problems in RiPLE-Design 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 = % of subjects that had difficulties to understand RiPLE-Design.

M8. Applicability Problems (AP): This issue will be used to identify possible applicability problems during the execution of RiPLE-Design. It is necessary to identify and analyze the difficulties found by the subjects applying the approach in practice. 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 regarding the applicability of the RiPLE-Design 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 = % of subjects that had difficulties to apply the RiPLE-Design in practice.

5.2.1.4 Qualitative assessment

Apart from the quantitative analysis that will be carried out based on the aforementioned questions and metrics, a qualitative assessment of the architectural solutions produced by the subjects during the experiment will be performed. The objective of this analysis is to assess the quality of the produced artifacts and the proposed solutions. This evaluation is essentially subjective, as many architectural evaluation methods, such as ATAM (Kazman et al., 2000), although very structured as a methodology, are still very dependent on the evaluators' experience.

5.2.2 Planning

This plan identifies all the issues to be addressed so that the evaluation runs smoothly, including the training requirements the necessary measures and the data-collection procedures (Kitchenham et al., 1995). As any other type of engineering activity, the experiment must be planned and the plans must be followed in order to control the study. The results of the experiment can be disturbed, or even destroyed if not planned properly.

5.2.2.1 Experiment task

The objective of this experiment is to evaluate the quality of the generated architecture, understanding and applicability of the RiPLE- Design in the context of software product line projects. The experiment will be conducted in a university laboratory with postgraduate students using a project on the Travel Reservation domain.

The 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).

Given a simplified product line specification, comprising domain description, feature model specification, requirements specification and quality attribute scenarios, the subjects must produce a product line architecture specification describing: modules and component structural specifications (component diagrams), behavioral specification (sequence diagrams) and deployment specification as well as the variation mechanisms adopted.

The subjects of the study will be requested to act as the roles defined in the RiPLE-Design, i.e., domain architects and domain manager. A subject can play more than one role during different activities and tasks of the RiPLE-Design. All the subjects will be trained to use the approach as discussed next.

5.2.2.2 Experiment procedure and instrumentation

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 a second step, independently of the university course, the experimenter will present and discuss the concepts and guidelines of the Enhanced RiPLE-Design for software product line architectural development. It will be discussed during a two-hour lecture. During the training, the subjects can interrupt to ask issues related to the lecture.

In order to assess their experience, 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 software product lines and software reuse.

In order to guide the participants in the experiment, the complete description of the RiPLE-Design, with all supporting material, such as templates and guidelines will be provided by the experimenter. Additionally, the requirements of the software 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. RiPLE-Design: Complete description of the activities and tasks of the RiPLE-Design;
2. Architecture Template: A document template to document the 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 software product line.

The material will also include a second questionnaire (QT2) to evaluate the difficulties of the participants in reading and using the approach in practice. This questionnaire has the purpose of identifying possible misunderstandings and applicability problems during the execution of the RiPLE-Design. The architecture template can be seen in Appendix A, and the questionnaires in Appendix C.

5.2.2.3 Hypotheses, null hypotheses, alternative hypotheses

5.2.2.3.1 Null hypotheses

The null hypotheses determine that the use of the RiPLE-Design in software 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.

H1. µCBC without RiPLE-Design < µCBC RiPLE-Design.

H2. µCI without RiPLE-Design < µCI RiPLE-Design.

H3. µAFD without RiPLE-Design > µAFD with RiPLE-Design.

H4. µAFV without RiPLE-Design > µAFV with RiPLE-Design.

H5. µAQA without RiPLE-Design > µAQA with RiPLE-Design.

H6. µAVQA without RiPLE-Design > µAVQA with RiPLE-Design.

H7. µ More than 50% of the subjects will have difficulties to understand the RiPLE-Design.

H8. µ More than 50% of the subjects will have difficulties to apply RiPLE-Design in practice.

Since no baseline exists concerning the understandability and the applicability of the RiPLE-Design process, neither for the traditional process nor for the enhanced version, the value of 50% was arbitrarily chosen in the hypotheses H7 and H8.

5.2.2.3.2 Alternative hypotheses

H1. µCBC without RiPLE-Design ≥ µCBC RiPLE-Design.

H2. µ CI without RiPLE-Design ≥ µCI RiPLE-Design.

H3. µAFD without RiPLE-Design ≤ µAFD with RiPLE-Design.

H4. µAFV without RiPLE-Design ≤ µAFV with RiPLE-Design.

H5. µAQA without RiPLE-Design ≤ µAQA with RiPLE-Design.

H6. µAVQA without RiPLE-Design ≤ µAVQA with RiPLE-Design.

H7. µ More than 50% of the subjects will not have difficulties to understand the RiPLE-Design.

H8. µ More than 50% of the subjects will not have difficulties to apply RiPLE-Design in practice.

5.2.2.4 Subjects and Objects

The selection of subjects, or sample of population, 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). Ideally, this selection should be randomly chosen.

The subjects of the experimental study will act as domain architect as defined in the RiPLE-Design. In this experiment, the subjects will be selected using a *Convenience Sampling*, in which the nearest and most convenient people are selected. It is a non-probability sampling technique, i.e., the probability of selecting each subject is unknown (Wohlin et al., 2000).

If there is a large variability in the population, a larger sample size is needed (Wohlin et al., 2000). In this study, the variability of the population is not very large, since all of the subjects 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 as some of them have worked in different organizations.

The experience of the subjects with product line projects and architectural design is an *independent variable* of this study and will be used to group subjects.

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 quality of the artifacts produced, the understandability of the process and its applicability are considered *dependent variables* of this study.

The experience of the subjects and the use of RiPLE-Design will be manipulated with the purpose of measuring the effects on the quality of the architecture generated. In addition, the understandability of the RiPLE-Design documentation, and the applicability of the process will be analyzed considering the experience of the subjects.

5.2.2.5 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 5-1. In the context of experimentation, there are three general design principles that are frequently used in experimental studies:

1. Blocking: It is used to systematically eliminate the undesired effect in the comparison among the treatments. Within one block, any undesired effect is the same and we can study the effect on the treatments on that block (Wohlin et al., 2000);
2. 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 (Wohlin et al., 2000).
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 (Wohlin et al., 2000).

Based on those principles, the assignment of subjects will be done by the following rules:

(i) The experience of the subjects, assessed through the use of a questionnaire, will be used to group subjects with similar profiles;

(ii) From the same experience category, the assignments of subjects to the treatments will be done randomly;

(iii) In order to balance the experiment, the same number of subjects will take part in each treatment. Furthermore, each treatment will have similar experience average.

For example, if we count with 4 high experienced and 4 low experienced subjects, each treatment will count 2 high experienced and 2 low experienced subjects.

Factor	
The quality of the PLA produced, measured by Coupling and Instability	
Treatment 1	***Treatment 2***
Design the PLA without a structured method	Design the PLA following the Enhanced RiPLE-Design

Table 5-1 One Factor with two treatments design

5.2.2.6 Validity evaluation

It is fundamental to evaluate the validity of experiment's results. 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, four categories of validity were considered 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 study, the following threats to conclusion validity were considered:

- Reliability of measures: The validity of an experiment is very dependent on the reliability of the measures. In this study, no baseline values for the metrics were found, since the experimentation performed for the original RiPLE-Design (Souza Filho et al., 2009), have had different objectives. This issue can be a problem since baselines in the context of the study cannot be used 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." (Wohlin et al., 2000). In this sense, the study will try to reduce group heterogeneity by choosing subjects from a group of postgraduate students that do research in the same area and attended to a similar set of postgraduate disciplines.

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 threat was considered:

- 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. Thus, it can be assumed that they have some interest in the study.

Construct validity: It concerns the generalization of the experiment's results outside the experiment setting. In this study, the following construct validity threat 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 gives the full picture of the theory. In this sense, it would be better if the proposed enhancements to RiPLE-Design could be analyzed comparing it with other software product line design approach. However, as already mentioned, other systematic and structured approaches that deal with quality attribute variability are still emerging. RiPLE-Design will then be compared with 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 study, the following external validity threat was considered:

- Interaction of setting and treatment: This is the effect of not having the experimental settings or material representative of industrial practices, for example. In this study, the Travel Reservation domain will be used. This domain is commonly used in Service-oriented architecture and software product lines works, such as (Medeiros, 2010; Snell, 2002), and can represent a real and complex problem.

5.2.2.7 The experiment project

The project used in this experimental study was a software product line to develop systems to fit the requirements of different online travel agencies. Three systems in this domain were selected and detailed carefully. The subjects were asked to perform domain design activities with the purpose of designing a set of architectural elements that could be reused to develop the three systems, in the form of a software product line. A similar project was used under experimentation in (Medeiros, 2010).

The systems designed offer to their customers the benefit of planning and reserving travel arrangements on the Internet. The three systems should achieve the following goals (Medeiros, 2010):

- The product line should allow customers to submit travel itineraries and payment information to the product line components using a *Web* interface;
- The travel agency services should automatically obtain and reserve the appropriate services for the airline, hotel or vehicle according to the customer itineraries;
- It should perform compensation operations for canceling itinerary failures;
- 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 accommodation reservations to bigger travel agencies that provide services to reserve airline tickets, accommodation and vehicles.

The agencies focus in different market niches:

- **Massive online agencies** require high level of availability, as they work online and those systems can be accessed from anywhere in the globe, at anytime.
- **Premium agencies** require very high level of confidentiality as they deal with VIP customers, such as companies CEOs. The user of these agencies are normally the VIP secretaries, they normally use its services during workdays. Service latency is **not** a major concern for this kind of product.
- **Regional specialized agencies** do not require very high level of confidentiality, but require very short latency as they offer specialized services.

5.2.2.8 Pilot Project

A pilot project was conducted before performing the study with the same structure defined in this planning. The pilot project was performed by one subject, who was trained and will not participate of the real experiment. In the pilot project, the subject used the same material described in this planning, and was observed by the responsible researcher. The objective of the pilot project was to detect problems and improve the planned material before its use.

The pilot project subject could be categorized as one with significant experience according to the aforementioned classification, having participated in five industrial and four academic projects.

The pilot project could ensure the viability of the experiment in relation to the defined instruments, procedure and metrics. As the pilot was held only with one treatment, there was no comparison baseline to analyze the defined hypotheses. Nonetheless, all metrics could be gathered from the subject's output. Namely, the subject had no difficult to understand variable quality attributes (EoP = 0%); the mean of CBC metric was 1.15; the mean of CI metric was 0.51; The AFD, AFV, AQA and AVQA metrics were all of 100%, meaning that all variation points were embraced by the architecture as well as the functional and quality attribute requirements. The subject demonstrated also no difficulties to neither understand nor apply the Enhanced RiPLE-Design process.

After conducting the pilot project some enhancements to the material were incorporated. Namely, the architecture document template used to gather the proposed solution was enhanced in order to suggest the subject to present structural diagrams for components and modules. The final version of this template is presented in Appendix Appendix A.

5.2.3 Operation

This section presents the details about the execution of the experimental study performed with the purpose of characterizing and refining the Enhanced RiPLE-Design.

5.2.3.1 Environment

The experimental study was conducted with six subjects that performed the experiment in parallel. Each of the subjects designed a Domain Specific Software Architecture (DSSA), as proposed by the experiment task. The execution of the experiment task took 8 hours at the Federal University of Pernambuco (UFPE).

Some subjects from treatment that performed the experiment task without the aid of the Enhanced RiPLE-Design process executed the task remotely and independently. It has not impacted the validity of the study, since those subjects did not require training and should perform the task alone. Those subjects have been instructed via videoconference presentation.

5.2.3.2 Training

The subjects were trained before the experimental study began. The training took 20 hours, divided into 10 lectures with two hours each, during the postgraduate course at the university. In addition, the subjects who used the proposed approach were trained 2 hours more to use the RiPLE-Design.

5.2.3.3 Subjects

Subjects were selected among students from the Federal University of Pernambuco, being three M.Sc. and three Ph.D. students. The recruiting process was open and accepted volunteers that had had already software reuse experience at least during the Software reuse lectures in the postgraduation course. All the subjects had industrial experience in software development for more than three years and had participated in industrial projects involving some kind of reuse activity. In addition, all the subjects had participated in SPL academic projects, and some had applied the approach in industrial context. The subject had also some experience with software design and domain design. Table 5-2 shows a summary of the profile of the subjects involved in this experiment.

Subjects with ID 1, 2 and 3 performed the experiment task with the RiPLE-Design process; subjects with ID 4, 5 and 6 performed the task without the aid of the process. The assignment of subjects and treatments followed the rules described in the Experiment Design section.

ID	Academic Projects	Industrial Projects	SPL Projects
1	(4) Low Complexity (0) Medium Complexity (0) High Complexity	(3) Low Complexity (1) Medium Complexity (1) High Complexity	(2) Academic
2	(1) Low Complexity (1) Medium Complexity (0) High Complexity	(1) Low Complexity (1) Medium Complexity (0) High Complexity	(1) Academic
3	(11) Low Complexity (3) Medium Complexity (2) High Complexity	(5) Low Complexity (5) Medium Complexity (4) High Complexity	(3) Academic
4	(8) Low Complexity (2) Medium Complexity (0) High Complexity	(3) Low Complexity (2) Medium Complexity (1) High Complexity	(3) Academic (1) Industrial
5	(0) Low Complexity (3) Medium Complexity (0) High Complexity	(0) Low Complexity (3) Medium Complexity (0) High Complexity	(1) Academic (1) Industrial
6	(2) Low Complexity (1) Medium Complexity (0) High Complexity	(2) Low Complexity (2) Medium Complexity (2) High Complexity	(2) Academic

Table 5-2 Subject's Profile

5.2.3.4 Costs

The costs associated with the experimental study were relative to its planning and operation. All subjects were student volunteers from the Universidade Federal de Pernambuco, and the execution environment was the university labs. Planning for the experimental study took about two months. Within this period, the planning was reviews and refined three times.

5.2.4 Analysis and Interpretation

The results from the experimental study will be presented in this section. The analysis is split in quantitative and qualitative analysis.

5.2.4.1 Quantitative analysis

Coupling: information about component coupling was collected from the proposed architectural solutions and then analyzed. The coupling between the components identified by the subjects is shown in Figure 5.1. Since different subjects identified and named components in different ways, irrespective of the name given to each component, they are enumerated in the X axis. Y axis represents the CBC metric for each component. Additionally, the subjects with Id = 1, 2, 3 used the Enhanced RiPLE-Design, while subjects with Id = 4, 5 and 6 designed the project without following a structured method.

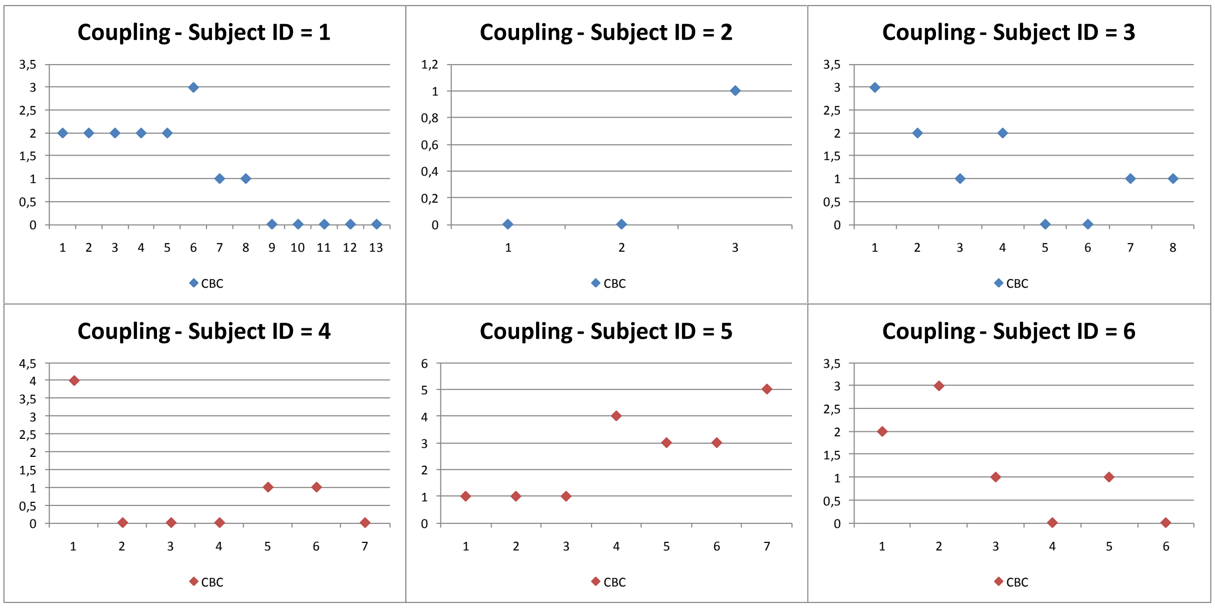

Figure 5.1 Component Coupling

As it can be seen in figure, the components generated using the Enhanced RiPLE-Design are more loosely coupled, when compared with the components produced without using the structured method. With regard to the coupling among the components defined by each subject, no result could be seen as an outlier. The result from subject (id = 4), with a highly coupled component (CBC = 4) among very loosely coupled components seems normal. The highly coupled component might represent a orchestration component, as in the *Mediator* design pattern (Gamma et al., 1995).

Figure 5.2 compares the average of the CBC metric for all components, following the Enhanced RiPLE-Design and without using any method. This comparison denotes the rejection of the null hypothesis (µCBC without RiPLE-Design < µCBC RiPLE-Design).

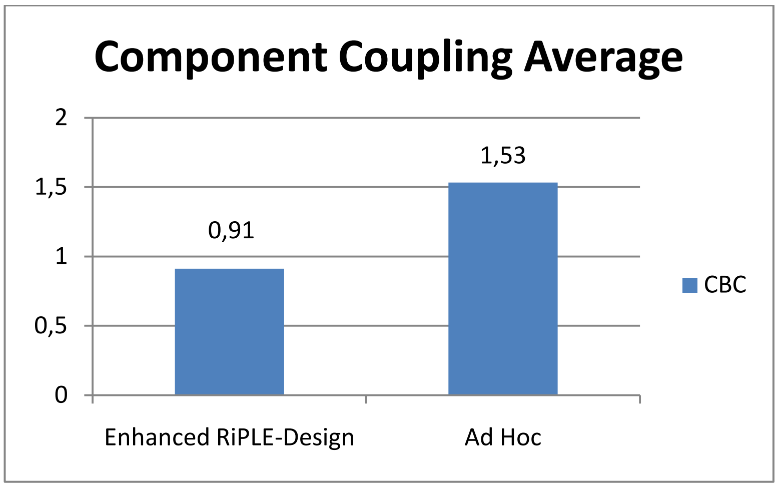

Figure 5.2 Component Coupling Average

We have not found any baseline for the coupling metric in the context of component development as well. Thus, there is no way to judge the values obtained with the treatments. However, these values can be used in new experiments as baselines.

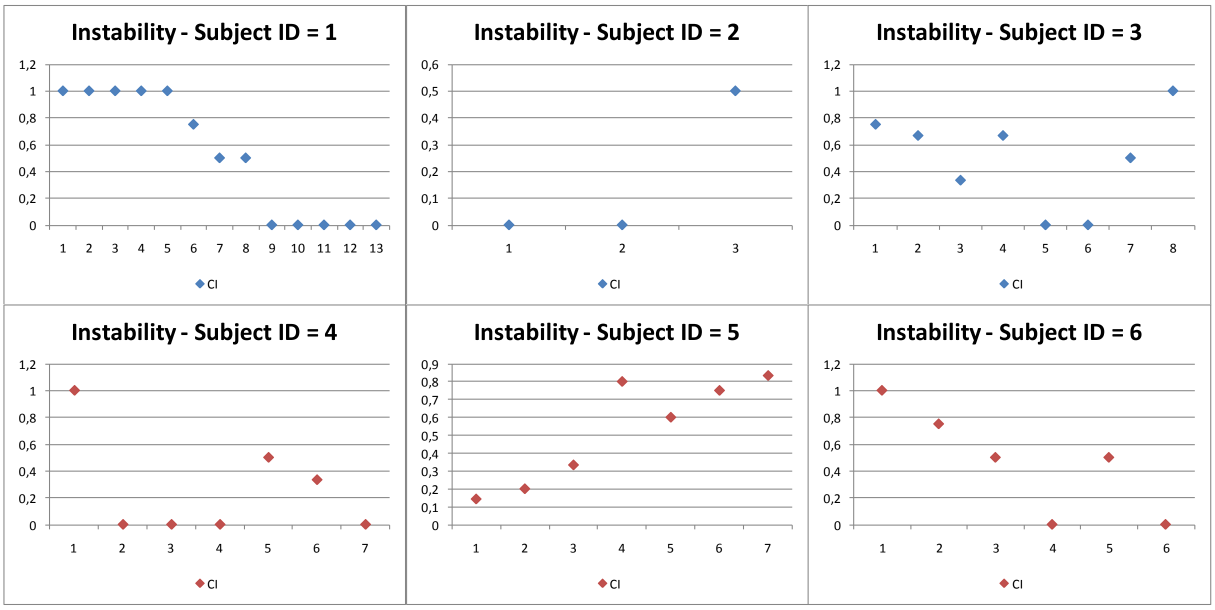

Figure 5.3 Component Instability

Instability: information about component instability was collected from the proposed architectural solutions and then analyzed. The instability form the components identified by the subjects is shown in Figure 5.3. They are enumerated in the X axis with the same identifier as in the coupling analysis. Y axis represents the CI metric for each component. As in the coupling analysis, the subjects with Id = 1, 2, 3 used the Enhanced RiPLE-Design, while subjects with Id = 4, 5 and 6 designed the project without following a structured method. With regard to the instability of each components defined by each subject, no result could be seen as an outlier.

In average, the components generated using the Enhanced RiPLE-Design are slightly less instable, when compared with the components produced without using the structured method, as shown in Figure 5.4. This aspect indicates that the null hypothesis (µCI without RiPLE-Design < µCI RiPLE-Design.) can be rejected.

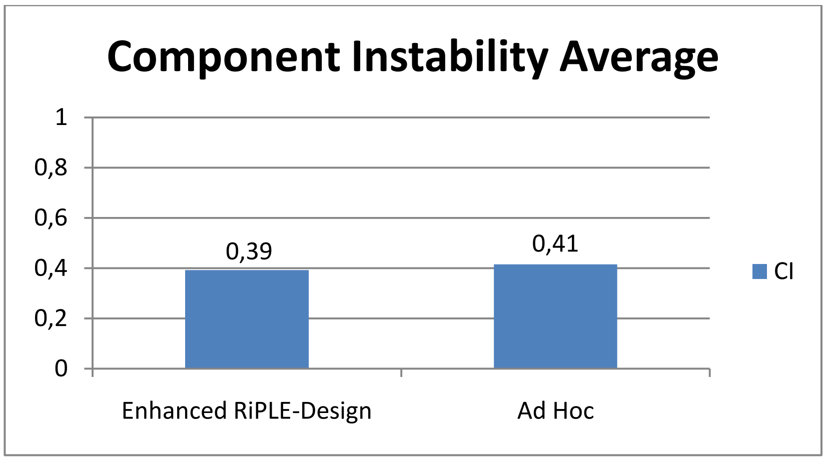

Figure 5.4 Component instability average

We have not found any baseline for the instability metric in the context of component development as well. Thus, there is no way to judge the values obtained with the treatments. However, these values can be used in new experiments as baselines.

As Figure 5.4 shows, the values of the average instability metric are very similar between the two treatments. The main factor that can be related to this similarity is the simplicity of the experiment task domain, which lead the subjects to develop simple architectures, with few components.

Functional requirements: aiming to evaluate whether the proposed architectural solutions were able to achieve the functional requirements described in the experiment task, the AFD metric was collected and analyzed. Six use cases were defined in the travel reservation task and the data collected represents, for each subject, the percentage of use cases that were comprised by the proposed architecture.

Figure 5.5 shows a comparison between the mean of AFD from each treatment. Since every functional requirement was addressed by all architectural solutions, irrespective of the treatment, the null hypothesis (µAFD without RiPLE-Design > µAFD with RiPLE-Design) can be rejected without much significance.

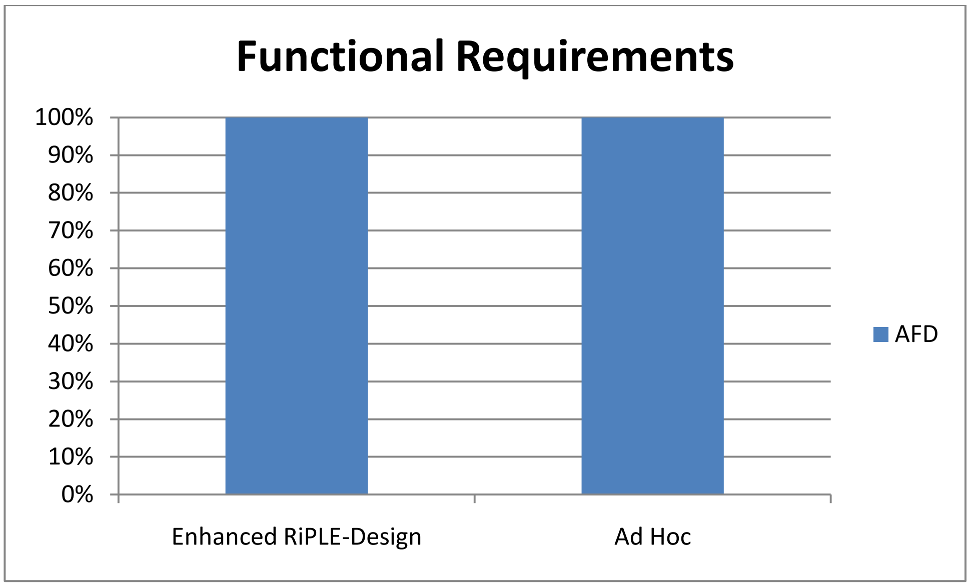

Figure 5.5 Functional Requirements

The equality of the results for both treatments can be justified by the simplicity of the experiment domain and the subjects' experience. Two factors relate to the experiment domain: functional requirements were few and very clearly described in the experiment task. Concerning subjects' experience, since all of them had already played developer and analyst roles in software projects, it can be acceptable that they can perceive and understand functional requirements straightforwardly.

Functional variability: aiming to evaluate whether the proposed architectural solutions were able to achieve the functional variability described in the experiment task, the AFV metric was collected and analyzed. A total of three functional variation points were described in the experiment task, being two groups of three alternative features each and one group of two mutually exclusive (XOR) features. The data collected represents, for each subject, the percentage of functional variation points that were addressed by the proposed architecture.

Figure 5.6 shows a comparison between the mean of AFV from each treatment. Data shows that, irrespective of the treatment, every functional variation point was addressed by the architectural solutions. Thus, the null hypothesis (µAFV without RiPLE-Design > µAFV with

RiPLE-Design) can be rejected without much significance. The small significance of the rejection can also be due to the small number of functional variation points present in the experiment task. It is important to remark, though, that in larger products, with more variation points, the use of a process becomes indispensable due to the raise in complexity.

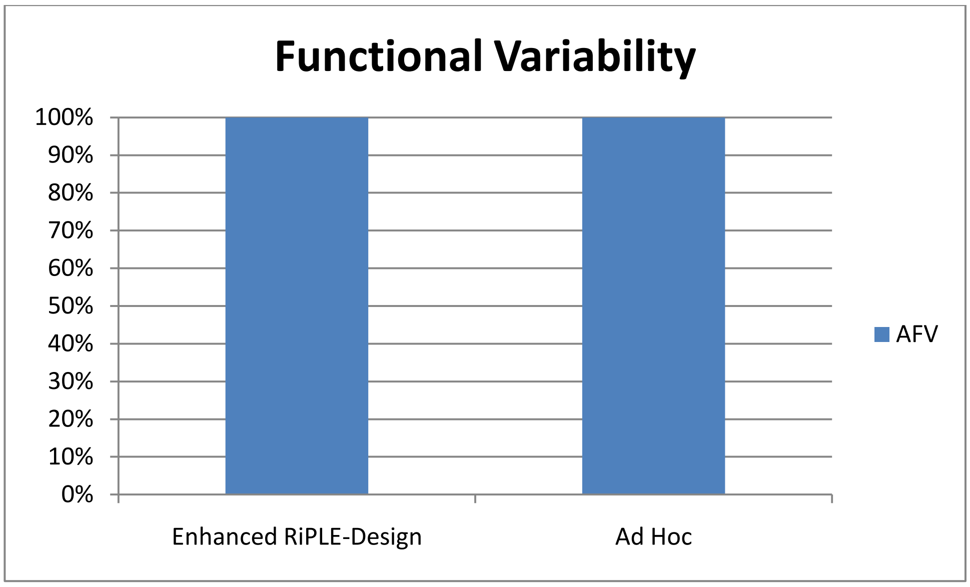

Figure 5.6 Functional Variability

The same line of reasoning that justified the equality of results for the Functional Requirement achievement can be used to explain the similarity in the Functional Variability achievement. Another factor that can be added is the educational support given by the Software Reuse Lectures in the postgraduation course, which all the subjects attended to. During the lectures, attendees are asked to participate in a simulated software product line factory, producing core assets and deriving products. It seems right to assume that all subjects had already worked with and were familiar to functional variability issues such as Feature Model Diagrams and functional variability achievement.

Quality attributes: data was collected in order to evaluate whether the proposed architectural solutions addressed the quality attribute requirement described in the experiment task. Three quality attributes were defined in the travel reservation task and the data collected represents, for each subject, which percentage of quality attribute were addressed by the proposed architecture.

Figure 5.7 shows a comparison between the mean of AQA from each treatment. We could observe that without a proper methodology, the treatment of quality attribute scenarios have been neglected by 16% of the subjects, since the architectural solution for the proposed task does not address all of them.

Figure 5.7 shows that from without the aid of RiPLE-Design, in average, only 67% of the quality attributes requirements were addressed against 100%, when the subjects followed the Enhanced RiPLE-Design process. Based on this analysis, the null hypothesis (µAQA without RiPLE-Design > µAQA with RiPLE-Design) can be rejected.

In absolute numbers, the results represents that a single subject (id = 6) missed the handling of quality attribute. No relation between the subjects profile and the mistreatment of the quality attribute scenario, since the subject is not the most or the less experience in his group. Although the subject has already played system analyst and architect roles in projects, the negligence could have happened due to the particular lack of experience with the quality attribute terminology. The reduced number of subjects suggests that further experimentation is needed to prove the significance of these results. Nonetheless, this preliminary result suggests that the use of the process, even in small teams and small projects, can be a helpful way to standardize the procedures and conduct of team members.

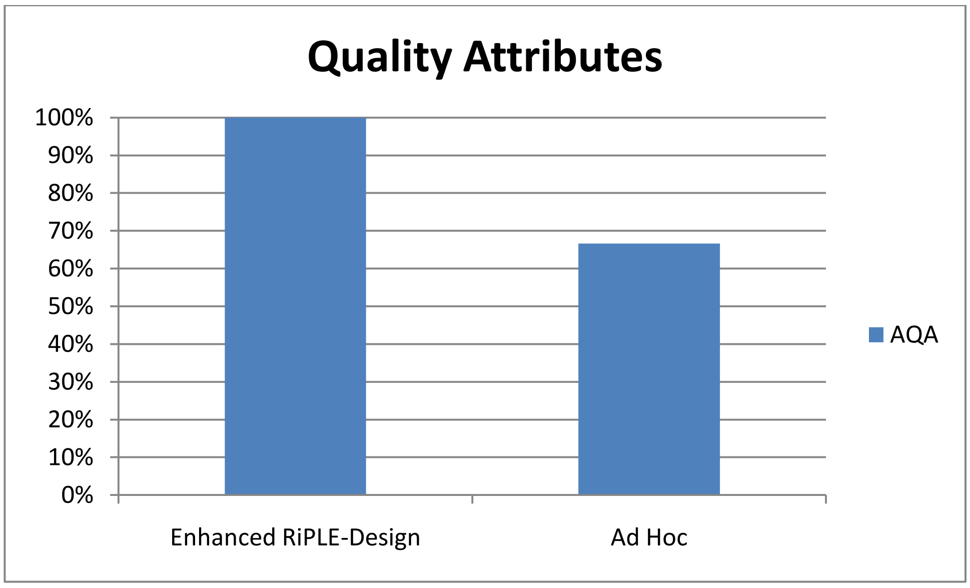

Figure 5.7 Quality Attributes

Quality attributes variability: the data was collected in order to evaluate whether the proposed architectural solutions addressed the quality attribute variability described in the experiment task. Two variation points concerning quality attributes were defined in the travel reservation task. The first one represented the optionality of High availability for certain category of products, the other variation point represented the trade-off between security and performance. The data collected represents, for each subject, the percentage of variable quality attributes that were addressed by the proposed architecture.

Albeit small, the number of variation points concerning quality attributes seems realistic. The development of an architecture with a several quality attributes variation point would be impractical. In such cases it could be better to suppress the variability during the scoping phase.

Figure 5.8 shows a comparison between the mean of AQVA from each treatment. We could observe that without a proper methodology, 16% of the subjects neglected quality attribute variability, since their architectural solution for the proposed task does not address all of them.

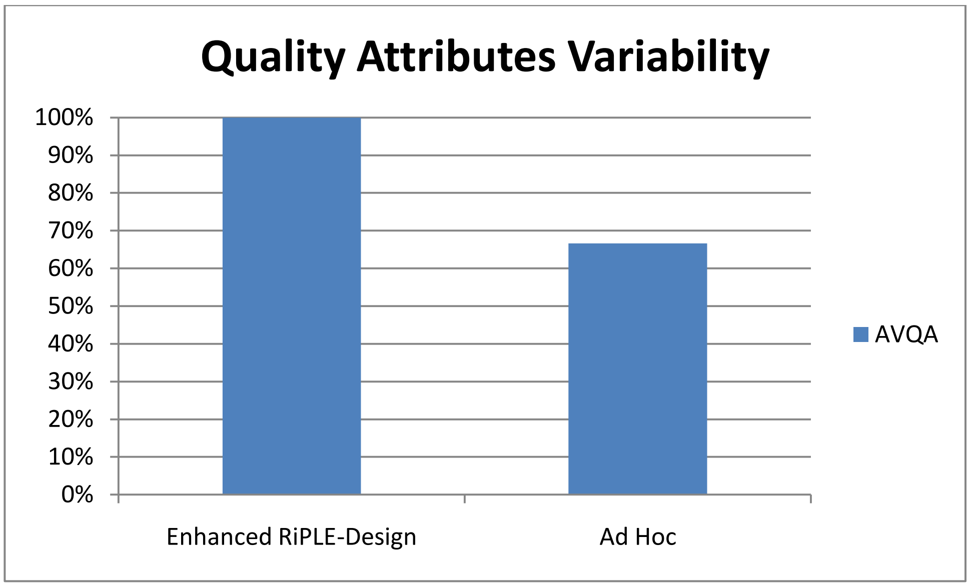

Figure 5.8 Quality Attributes Variability

Figure 5.8 shows that from without the aid of RiPLE-Design, in average, only 67% of the variable quality attributes were addressed against 100%, in average, when the subjects followed the Enhanced RiPLE-Design process. Based on this analysis, the null hypothesis (µAQVA without RiPLE-Design > µAQVA with RiPLE-Design) can be rejected. The reduced number of subjects suggests that further experimentation is needed to prove the significance of these results.

In absolute numbers, the results represents that a single subject (id = 6) missed the handling of quality attribute variability. As well as in the AQA metric, the negligence could have happened due to the particular lack of experience with the quality attribute terminology.

Difficulties to understand the activities: from the answers of the subjects using the questionnaire (QT2) for the difficulties found to understand the RiPLE-Design activities, it was identified that (Id = 3) had difficulties to understand part of the approach, namely, the *Document decision guidelines* activity. It was mentioned that the provided examples left unclear the definition of patterns, since one of the examples showed System reboot as a pattern. This issue was explained during the RiPLE-Design training and the documentation of the approach was reviewed and modified, emphasizing that the documentation should be of design decisions and strategies, not only design patterns or styles.

The subject (Id = 3) had participated of plenty of academic and industrial software projects in technical leading roles. In this sense, it is unlikely that the experience caused the misunderstanding.

In this sense, one subject had problems to understand the Enhanced RiPLE-Design activities, which represents 33% of the total number of subjects that used the process. This aspect confirms that the null hypothesis (understanding problems > 50%) can be rejected. It is necessary to highlight that the null hypothesis was defined without any previous data. Nonetheless, this value can be refined based on this experience, the next time the experiment is performed.

Difficulties to apply the process: from the answers of the subjects using the questionnaire (QT2) for the difficulties found to apply the Enhanced RiPLE-Design in practice, it was identified that one subject (id = 2) reported problems to apply the *Represent Architecture* and mentioned his lack of experience with design as a reason for the difficulties. According to the subject’s profile, the roles played in projects were related mainly to software Tests. It is expected that people with little experience in software design have difficulties in executing such tasks. Software product line engineering adds complexity to the already software architecture discipline. In real case scenarios, more and deeper training involving general software architecture could provide solider foundations to the architects.

New experiments need to be executed in order to get more evidence about the correlation among the experience of the subjects and the difficulties found to apply the approach in practice.

In this sense, one from the three subjects had problems to apply the Enhanced RiPLE-Design activities in practice. It represents 33% of the total number of subjects that used the approach. This aspect indicates that the null hypothesis (applicability problems > 50%) can be rejected. It is necessary to highlight that the value for the null hypothesis was defined without any previous data and this value can be calibrated for new experiments.

5.2.4.2 Qualitative analysis

Based on the answers from questionnaire (QT2) and on the architectural solutions proposed by each subject, qualitative analyses were performed.

Training Analysis: all the subjects who participated in the experimental study attended to lectures composed of slide presentation involving topics related to software reuse and software product lines architecture. The training was performed in 24 hours. Three subjects considered the training very good (Id = 2, 3 and 6), two subjects classified it as good (Id = 1, 4), and one subject as regular (Id = 5). The scale defined was: very good, good, regular, and unsatisfactory.

Two subjects (Id = 2 and 6) mentioned that it would be better to have more lectures on general software architecture, in order to compensate their lack of experience.

The subjects did not comment on the software product line lectures, since it involved several topics and a practical case study during the whole course at the university (approximately during 6 months).

Finally, the subjects (Id = 1, 2 and 3) were trained to use the Enhanced RiPLE-Design, and one subject (Id = 1) emphasized that the Enhanced RiPLE-Design training should be longer in order to improve the results of the experiment. The subjects (Id = 4, 5 and 6) were not trained to use the Enhanced RiPLE-Design, since they designed the project without using a structured method (ad-hoc).

Usefulness of RiPLE-Design: the subjects that used the Enhanced RiPLE-Design reported that the approach was useful to perform the domain design with quality attribute variability.

Nonetheless, one subject (id = 1) suggested the use of an UML adaptation to represent variability. Yet another subject (Id = 3) missed tool support for editing the Feature Model Diagram. Both issues will be commented further as a future improvement to the process.

The subject (id = 3) also highlighted the importance of the quality scenario analysis and its reproduction in the feature model diagram. It was useful to identify the level of influence of the quality attributes on the architecture.

Quality of the Documentation and Instruments: about the experimental study task description, one subject (id = 2) presented difficulties to understand quality scenarios descriptions as he was not used to the quality scenario notation. The lack of experience of the subjects seem to be the main reason of the difficulties since the subject has mainly participated in software projects in testing activities (test architect and tester). The misunderstandings were corrected during the course of the experiment and will be revised in the training material.

Albeit being the most experienced subject, the subject with (Id = 3) complained about the lack of suggestions and guidelines for design strategies regarding each specific quality attribute. Those mentioned design strategies are already available in the software architecture literature, as in (Bass; John, 2003; Garlan; Shaw, 1994; Kim et al., 2009; Klein et al., 1999; Myllärniemi et al., 2008; Schmidt; O'Ryan, 2003). The idea behind the Enhanced RiPLE-Design is that known architectural styles and patterns or any other design strategy can be used to address quality attributes. Alternative or optional strategies can be used when quality attribute variability appears. The next time this experiment is performed, this aspect can be better controlled by including a list of suggested readings or by training the subjects in architectural tactics.

Other subjects commented that the process documentation was fine.

Handling quality attributes variability: besides the use of the process, another aspect clearly influences the handling of quality attribute variability: the architect's experience. In general more experience subjects (id = 3 and 4) that addressed quality attribute variability proposed better solutions than other subjects. Those subjects also documented better the design decisions regarding quality attribute variability.

It is interesting to remark that a subject (id = 6) have not addressed any variable quality attributes. It would be premature do draw any conclusions from only a single subject. It is possible that without the use of a systematic approach, the architect would not mind the occurrence of quality attribute variability. The subject's experience could also have influenced the result. In order to elucidate this occurrence, further experimentation is needed, with a larger population.

Nevertheless, this aspect can actually be a hidden threat to the experiment's validity. As the study was design for the purpose of evaluating a process to handle quality attribute variability, the experiment task construction could have been biased. In order to prove the experiment task's construction as a threat to the validity of the experiment, different domains, taken from industrial cases could be used to construct the experiment task.

Guidelines for product instantiation: analyzing the proposed architectural solutions it could be identified that no subject from the treatment that did not use the Enhanced RiPLE-Design process documented guidelines for product instantiation. Such guidelines are crucial for the next phase in product line engineering: product development.

Quality of produced architectural documentation: experience seems to be closely related to the quality of the proposed architectural solution as well as its documentation. Regardless of the treatment subjects with more experience produced better documented architecture. Examples are subjects 3 and 4. More detailed diagrams and documentation of design decisions rationale could be found. Less experienced architects produced modestly documented architectures with no advice for product instantiation or poor descriptions for quality attribute variability handling.

5.3 CONCLUSIONS

Although the analysis has not been conclusive, the experimental study indicates that the Enhanced RiPLE-Design allows architects to design domain specific software architectures with a reasonable coupling and stability. Moreover, satisfactory results could be seen concerning aspects related to understanding and applicability of the Enhanced RiPLE-Design in practice. Additionally, metrics can be calibrated with the results identified in this experiment. It was also identified that the training on the Enhanced RiPLE-Design should be extended in length and should comprise more details about quality attribute scenarios and design tactics.

Even with a small number of subjects, it can be valid to make some correlations based on the profile of the subjects and the results obtained. Concerning the quantitative analysis, even though no correlations between the experience of the subjects and the quality of the architecture could be found, it was clear that more experienced subjects produced better architecture documentation.

Although the results have shown similar results in respect to functional and quality attributes achievements, the preliminary result suggests that the use of the process, even in small teams and small projects, can be a helpful way to standardize the procedures and conduct of team members. In addition, the use of the process can be very helpful to manage large and complex software product line projects.

Considering the understanding and applicability of the RiPLE-Design, this correlation could be analyzed as described previously. More evidence for these correlations can be obtained from new experiments. In addition, no value was removed when analyzing the metrics results, as no value was considered an outlier (Fenton, 1994).

5.4 LESSONS LEARNED

From the conclusion of the experimental study, some aspects should be considered in order to replicate the experiment, in order to overcome some limitations of its first execution. In this sense, the next subsections present the lessons learned from the performed experimental study.

5.4.1 Training

Besides the improvements related to the RiPLE-Design lectures, two subjects (Id = 2 and 3) highlighted that the training should be longer, more detailed, and include topics like design tactics and general software architecture. The suggestions do not seem to have relation with the subjects' experience, since they are both the more and the less experienced subject in the population. Another possible improvement is the execution of an example, with reduced scope and performed by the subjects to simulate the experiment. These issues could reduce the doubts during the experiment.

5.4.2 Experience of subjects

Being aware of the possible threat due to the random heterogeneity of subjects, this study tried to choose subjects from a group of postgraduate students that do research in the same area and attended to a similar set of lectures. Nevertheless, individual differences could have been even larger than the difference of the treatment. As aforementioned, experience seems to be related to the quality of the proposed architectural solution as well as its documentation. Better strategies must be used I further studies to guarantee better homogeneity of the study group in respect to their experience.

5.4.3 Motivation of subjects

Although the subjects were all volunteers, it was difficult to maintain the subjects motivated, and keep their attention and discipline during the whole execution of the experimental study. Performing the experimental study during lectures of a university course can help solve this problem as suggested by a subject (id = 3).

5.4.4 Number of subjects

It was hard to find volunteers to perform the experimental study. This experiment was performed by a reduced number of subjects (6 participants), and the pilot project with only one subject. After this experimental study, the necessity to increase the number of subjects could be identified. The execution of the experiment during lectures of a university course may solve this issue as well.

5.5 CHAPTER SUMMARY

This chapter presented the contextualization, planning, operation and analysis of the experimental study that characterized the Enhanced RiPLE-Design process evaluating its efficacy, understanding and applicability.

The study analyzed the possibility of subjects using the approach to design product line architecture with good stability and coupling. It was also analyzed the understanding and applicability of the RiPLE-Design in practice. The difficulties were categorized in the different activities of the RiPLE-Design with the intention of evaluating and refining its activities.

Even with the reduced number of subjects, the analysis has shown that the RiPLE-Design 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.

The next chapter will present the conclusions of this work, its main contributions and related work as well as directions for future work.

6 CONCLUSION

As discussed in (van Der Linden et al., 2007):

> The software industry is challenged with a continuous drive to improve its engineering practice, and Software product line engineering is a strategic approach to developing software.
> […]
> It impacts business, organization and technology alike and is a proven way to develop a large range of software products and software-intensive systems fast and at low costs, while at the same time delivering high-quality software.

The ways organization can benefit from systematic software reuse by adopting product line engineering as well as the importance of proper software architecture design are presented in Chapter 2.

Quality attribute achievement is vital for any software architecture. In the context of software product lines, it becomes an even more complex issue as quality attributes can also contain variability. Nevertheless, this aspect have been "neglected or ignored by most of the researchers as attention has been mainly put in the variability to ensure that it is possible to get all the functionality of the products", as discussed in (Etxeberria et al., 2008).

The RiPLE-Design process, presented in (Souza Filho et al., 2008; Souza Filho et al., 2008) also lacks guidelines for proper quality attribute variability handling. Aiming to solve the problem of addressing quality attribute variability, this dissertation presented the enhancements to the RiPLE-Design process.

An experimental study on the Travel Reservation domain was performed with the purpose of characterizing the Enhanced RiPLE-Design and refining it considering the feedback 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. The next section presents the related studies that have considered the quality attribute variability and strategies to properly address it.

6.1 Related Work

Chapter 2 presented some processes for software product line architecture development, which are close to this work. Nonetheless, there is a key difference between this work and others: the specific treatment of quality attributes variability, as discussed subject through this dissertation.

Besides those complete processes for product line architecture development, some that are also closely related to this dissertation focus on parts of the process.

Many approaches address variability modeling and specification taking into consideration non-functional features they are briefly described in Section 4.1.

(Myllärniemi et al., 2007) provide guidelines to model variability in the Security quality attribute. The approach is based on a tool that supports the definition of an ontology and the representation of functional and security variability in different viewpoints.

Not many studies focus on strategies to realize variability in the quality level, i.e., finding a design strategy for varying quality attributes and are described below.

(Rossel et al., 2009a) shows an approach based on Model-Driven Engineering (MDE) where the PLA is seen as a set of transformations associated with the domain features. In his approach, the quality attributes requirements are also modeled as features. A derived product, built from a selection of features, can have its architecture built through the application of the earlier mentioned transformations. The variations in quality attributes requirements produce different transformation in the model and can make product architectures completely different from one another. Although the resulting product architectures may differ drastically from one another, since they all derive from a main root architecture, and is managed as a whole this approach can be seen as software product line architecture approach.

(Bosch, 2000) suggests the possibility of transforming quality attributes requirements into functionalities. For example, the requirement of security can be converted into login and encrypted passwords and protocols. This attempt to make non-functional requirements into functionalities does not work always. Not all quality attributes requirements can be transformed into functionality, e.g., there is no functionality that deals with performance the same way access control functionalities deal with security. It is not guaranteed that a quality attributes requirement is achieved by a specific set of functionalities. In other words, a system can have access control with highly encrypted passwords and protocol and still not be secure.

Finally, (Kim et al., 2007) discusses an approach to address quality attribute variability that must be configured at runtime. It is a special case of quality attribute variability that was left out of the scope of this dissertation.

6.2 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 software product lines with quality attribute variability. 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:

- **Features Interaction**: The case of functional variability affecting quality attributes is related to the occurrence of feature interaction in software product lines. As pointed out in (Lee; Kang, 2004), sometimes "[…] features cannot perform their functionalities alone, they need to interact among them in order to accomplish the products requirements. In this context, a feature interaction occurs in a system whose complete behavior does not satisfy the separate specifications of all its features." Functional features that impact on non-functional features are, thus, a case of feature interaction. The problem of feature interaction can impact the whole SPL development process, as it promotes changes in reusable assets and impacts maintenance costs and other products. Although the quality scenario based approach allows basic treatment of feature interaction, this field of study can be further incorporate to the RiPLE-Design process.
- **Variability representation**: As suggested during the experiment, there could be alternative ways to represent variability, particularly in the deployment view. In this sense, studies from (Gomaa, 2004; Robak et al., 2002) can be adapted.

- **Experimental Study**: This dissertation presented the definition, planning, operation, analysis and interpretation of an experimental study that was executed with the purpose of characterizing and evaluating the Enhanced RiPLE-Design approach. New studies in different contexts, including more subjects and other domains are still necessary in order to calibrate the proposed approach.
- **Architectural Evaluation**: It is also important to perform further studies in order to examine how the proposed adaptation of the HoPLAA evaluation method works together with the Enhanced RiPLE-Design process.

6.3 ACADEMIC CONTRIBUTIONS

The co-authoring of the following publication contributed for acquiring experience and knowledge in the software product line architecture and software reuse area:

- (Souza Filho et al., 2008) Evaluating Domain Design Approaches Using Systematic Review, In 2nd European Conference on Software Architecture (ECSA).

Moreover, one journal article and one conference paper are being prepared.

6.4 CONCLUDING REMARKS

Software reuse is a key factor for companies willing to improve productivity and quality while reducing costs. In this context, this work presented the Enhanced RiPLE-Design, an approach to design software product line architectures. It enhances the existing RiPLE-Design process providing activities and guidelines to handle quality attribute variability.

The Enhanced RiPLE-Design approach was based in the three pillars for handling quality attribute variability, suggested by (Myllärniemi; Männistö; et al., 2006): (i) specify and model varying quality attributes; (ii) find a design strategy for varying quality attributes; and, (iii) evaluate the architecture in order to achieve the needed variation. The approach provides activities and guidelines that under each of the pillars.

Additionally, the approach was evaluated in a software product line project through an experimental study on the Travel Reservation domain, which was analyzed both quantitatively and qualitatively. This experimental study presented findings that the Enhanced

RiPLE-design can be viable to aid software architects during the design of product line architectures with quality attribute variability.

Therefore, this dissertation can be considered a relevant contribution to the area of software reuse and software architecture.

REFERENCES

Almeida, E. S. **RiDE: The RiSE Process for Domain Engineering**. Recife, PE, Brazil: UFPE, 2007. Ph.D. Thesis, Universidade Federal de Pernambuco, March, 2007.

Almeida, E. S.; Alvaro, A.; Lucredio, D.; Garcia, V. C.; Meira, S. R. RiSE project: towards a robust framework for software reuse. In: the 2004 IEEE International Conference on Information Reuse and Integration. **Proceedings of...** . p.48-53, 2004. Las Vegas, USA: IEEE.

Alvaro, A.; Almeida, E. S.; Meira, S. R. A software component quality model: A preliminary evaluation. In: Software Engineering and Advanced Applications, 2006. SEAA'06. 32nd EUROMICRO Conference on. **Proceedings of...** . p.28–37, 2006. IEEE.

America, P.; Obbink, J. H.; van Ommering, R. C.; van Der Linden, F. CoPAM: A component-oriented platform architecting method family for product family engineering. **KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE**, 2000.

Anastasopoules, M.; Bayer, J.; Flege, O.; Gacek, C. **A Process for Product Line Architecture Creation and Evaluation. PuLSE-DSSA - Version 2.0**. Kaiserslautern: Fraunhofer IESE, 2000. Fraunhofer IESE.

Apache Foundation. **The Apache Cassandra Project**. 2010. Available at: <*http://cassandra.apache.org*>. Accessed in: 4/18/2009.

Atkinson, C.; Bayer, J.; Bunse, C.; et al. **Component-based product line engineering with UML**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

Atkinson, C.; Bayer, J.; Muthig, D. Component-Based Product Line Development: The KobrA Approach. In: P. Donohoe; First Software Product Line Conference. **Proceedings of...** . p.289-309, 2000.

Auerswald, M.; Herrmann, M.; Kowalewski, S.; Schulte-Coerne, V. Reliability-Oriented Product Line Engineering of Embedded Systems. In: Software product-family engineering: 4th international workshop, PFE 2001. **Proceedings of...** , 2001.

Barbacci, M.; Klein, M. H.; Longstaff, T. A.; Weinstock, C. B. **Quality Attributes**. Software Engineering Institute, Carnegie Mellon University, 1995. Software Engineering Institute, Carnegie Mellon University.

Basili, V. R. The role of experimentation in software engineering: past, current, and. In: 18th International Conference on Software Engineering. **Proceedings of...** . p.442-449, 1996. Berlin, Germany: IEEE Computer Society.

Basili, V. R.; Caldiera, G.; Rombach, H. D. The goal question metric approach. **Encyclopedia of Software Engineering**, 1994. John Wiley & Sons, Inc.

Basili, V. R.; Selby, R. W.; Hutchens, D. H. Experimentation in Software Engineering. **IEEE Transactions on Software Engineering**, v. 12, n. 7, p. 733-743, 1986.

Bass, L. J.; Klein, M. H.; Bachmann, F. Quality Attribute Design Primitives and the Attribute Driven Design Method. In: PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering. **Proceedings of...** . p.169-186, 2002. London, UK: Springer-Verlag.

Bass, L.; Clements, P. C.; Kazman, R. **Software Architecture in Practice**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

Bass, L.; Clements, P. C.; Kazman, R.; Nord, R. The Architecture Business Cycle Revisited: A Business Goals Taxonomy to Support Architecture Design and Analysis. **News at SEI**, 2005.

Bass, L.; John, B. E. Achieving usability through software architectural styles. In: CHI '00: CHI '00 extended abstracts on Human factors in computing systems. **Proceedings of...** . p.171, 2000. New York, New York, USA: ACM Press.

Bass, L.; John, B. E. Linking usability to software architecture patterns through general scenarios. **Journal of Systems and Software**, v. 66, n. 3, p. 187-197, 2003.

Bayer, J.; Flege, O.; Knauber, P.; et al. PuLSE: a methodology to develop software product lines. In: SSR '99: Proceedings of the 1999 symposium on Software reusability. **Proceedings of...** . p.122-131, 1999. New York, NY, USA: ACM.

Boehm, B. **Characteristics of Software Quality**. North-Holland Pub Co, 1978.

Bosch, J. **Design and use of software architectures: adopting and evolving a product-line approach**. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

Bosch, J. Software product lines: organizational alternatives. In: the 23rd International Conference on Software Engineering. **Proceedings of...** . p.91-100, 2001. Washington, DC, USA: IEEE Computer Society.

Brito, K. S. **LIFT: A Legacy InFormation retrieval Tool**. Recife, PE: UFPE, 2007. M.Sc. Thesis, Universidade Federal de Pernambuco, May, 2007.

Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. **Pattern-oriented software architecture: a system of patterns**. New York, NY, USA: John Wiley & Sons, Inc., 1996.

Cavalcanti, Y. C. **BAST: A Bug Report Analysis and Search Tool**. Recife, PE: UFPE, 2009. M.Sc. Thesis, Universidade Federal de Pernambuco, May, 2009.

Chen, J.; Yeap, W. K.; Bruda, S. D. A Review of Component Coupling Metrics for Component-Based Development. In: 2009 WRI World Congress on Software Engineering. **Proceedings of...** . p.65-69, 2009. IEEE.

Chidamber, S.; Kemerer, C. A metrics suite for object oriented design. **IEEE Transactions on Software Engineering**, v. 20, n. 6, p. 476-493, 1994.

Chidamber, S.; Kemerer, C. A metrics suite for object oriented design. **IEEE Transactions on Software Engineering**, v. 20, n. 6, p. 476-493, 1994.

Chung, L.; Nixon, B. A.; Yu, E.; Mylopoulos, J. **Non-Functional Requirements in Software Engineering**. New York: Springer, 1999.

Clements, P. C.; Bachmann, F.; Bass, L.; et al. **Documenting Software Architectures: Views and Beyond**. London, UK: Addison Wesley, 2002.

Clements, P. C.; Kazman, R.; Klein, M. H. **Evaluating software architectures: methods and case studies**. London, UK: Addison-Wesley Professional, 2006.

Clements, P. C.; Northrop, L. M. **Software Product Lines: Practices and Patterns**. London, UK: Addison-Wesley Professional, 2001.

DeBaud, J.; Flege, O.; Knauber, P. PuLSE-DSSA --- a method for the development of software reference architectures. In: Third international workshop on Software architecture (ISAW '98). **Proceedings of...** . p.25-28, 1998. New York, NY, USA: ACM.

Dijkstra, E. W. Chapter I: Notes on structured programming. In: **Structured Programming**. p.1-82, 1972. London, UK: Academic Press Ltd.

Easterbrook, S.; Singer, J.; Storey, M.; Damian, D. Selecting Empirical Methods for Software Engineering Research. In: F. Shull; J. Singer; D. I. Sjøberg; **Guide to Advanced Empirical Software Engineering**, 2008. London: Springer London.

Etxeberria, L.; Mendieta, G. S.; Belategi, L. Modelling Variation in Quality Attributes. In: First International Workshop on Variability Modelling of Software-intensive Systems. **Proceedings of...** . p.51-59, 2007.

Etxeberria, L.; Sagardui, G. Product-Line Architecture: New Issues for Evaluation. In: J. H. Obbink; K. Pohl; Software product lines: 9th international conference, SPLC 2005. **Proceedings of...** , Lecture Notes in Computer Science. v. 3714, p.174-185, 2005. Springer.

Etxeberria, L.; Sagardui, G. Evaluation of Quality Attribute Variability in Software Product Families. In: 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2008). **Proceedings of...** . p.255-264, 2008. IEEE.

Etxeberria, L.; Sagardui, G.; Belategi, L. Quality aware software product line engineering. **Journal of the Brazilian Computer Society**, v. 14, n. 1, 2008.

Fenton, N. Software measurement: a necessary scientific basis. **IEEE Transactions on Software Engineering**, v. 20, n. 3, p. 199-206, 1994.

Folmer, E.; van Gurp, J.; Bosch, J. Scenario based assessment of software architecture usability. In: ICSE 2003 Bridging the Gaps Between Software Engineering and Human-Computer Interaction workshop. **Proceedings of...** , 2003.

Fowler, M.; Scott, K. **UML distilled: a brief guide to the standard object modeling language**. 2nd ed. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

Gacek, C.; Anastasopoules, M. Implementing product line variabilities. **ACM SIGSOFT Software Engineering Notes**, v. 26, n. 3, p. 109-117, 2001. New York, NY, USA: ACM.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. **Design patterns: elements of reusable object-oriented software**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

Garcia, V. C.; Lisboa, L. B.; Durão, F.; Almeida, E. S.; Meira, S. R. A lightweight technology change management approach to facilitating reuse adoption. In: 2nd Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS'08). **Proceedings of...** . p.116, 2008. Porto Alegre, Brazil.

Garlan, D.; Shaw, M. **An Introduction to Software Architecture**. Pittsburgh, PA, USA: School of Computer Science, Carnegie Mellon University, 1994. School of Computer Science, Carnegie Mellon University.

Gomaa, H. **Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures**. London, UK: Addison Wesley Professional, 2004.

González-Baixauli, B.; Laguna, M. A.; do Prado Leite, J. C. Using Goal-Models to Analyze Variability. In: First International Workshop on Variability Modelling of Software-intensive Systems, January 2007, Limerick, Ireland. **Proceedings of...** , Lero Technical Report. v. 2007-01, p.101-107, 2007.

Hallsteinsen, S.; Fægri, T. E.; Syrstad, M. Patterns in Product Family Architecture Design. In: Software product-family engineering: 5th international workshop, PFE 2003. **Proceedings of...** . p.261-268, 2003.

IEEE Computer Society. **IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology**. New York, NY, USA, 1990.

International Organization For Standardization/International Electrotechnical Commission. ISO/IEC 5807:1985, Information processing – Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts. , 1985.

International Organization for Standardization/International Electrotechnical Commission. ISO/IEC 9126-1:2001: Software Engineering - Product Quality - Part 1: Quality Model. , 2001. Geneva, Switzerland.

Jarzabek, S.; Yang, B.; Yoeun, S. Addressing quality attributes in domain analysis for product lines. **IEE Proceedings - Software**, v. 153, n. 2, p. 61, 2006.

Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S. Feature-Oriented Domain Analysis (FODA) Feasibility Study. , 1990.

Kang, K. C.; Kim, S.; Lee, J.; et al. FORM: A feature-; oriented reuse method with domain-; specific reference architectures. **Annals of Software Engineering**, v. 5, n. 1, p. 143–168, 1998. Springer.

Kazman, R.; Klein, M. H.; Clements, P. C. **ATAM: Method for architecture evaluation**. 2000. Software Engineering Institute, Carnegie Mellon University.

Kim, M.; Park, S.; Lee, J. An Approach to Dynamically Achieving Quality Requirements Change in Product Line Engineering. In: Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. **Proceedings of...** . v. 2, p.41-50, 2007.

Kim, S.; Kim, D.; Lu, L.; Park, S. Quality-driven architecture development using architectural tactics. **Journal of Systems and Software**, v. 82, n. 8, p. 1211-1231, 2009. Elsevier Inc.

Kitchenham, B.; Pickard, L. M.; Pfleeger, S. L. Case studies for method and tool evaluation. **IEEE Software**, v. 12, n. 4, p. 52-62, 1995.

Klein, M. H.; Kazman, R.; Bass, L. J.; et al. Attribute-Based Architecture Styles. In: First Working IFIP Conference on Software Architecture (WICSA1). **Proceedings of...** . p.225-244, 1999. Deventer, The Netherlands, The Netherlands: Kluwer, B.V.

Kolb, R.; Mcgregor, J. D.; Muthig, D. Introduction to quality assurance in reuse contexts. In: R. Kolb; J. D. McGregor; D. Muthig; First International Workshop on Quality Assurance in Reuse Contexts (QUARC). **Proceedings of...** , 2004. Fraunhofer IESE.

Krueger, C. W. Software reuse. **ACM Computing Surveys**, v. 24, n. 2, p. 131-183, 1992. New York, NY, USA: ACM.

Krueger, C. W. Easing the Transition to Software Mass Customization. In: PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering. **Proceedings of...** . v. 1, p.282-293, 2002. London, UK: Springer-Verlag.

Laprie, J. C. **Dependability: Basic Concepts and Terminology**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1992.

Lassing, N. Experiences with ALMA: Architecture-Level Modifiability Analysis. **Journal of Systems and Software**, v. 61, n. 1, p. 47-57, 2002.

Lee, K.; Kang, K. C. Feature Dependency Analysis for Product Line Component Design. In: ICSR 2004 : international conference on software reuse : methods, techniques, and tools. **Proceedings of...** . p.69-85, 2004.

Maccari, A. Experiences in assessing product family software architecture for evolution. In: Proceedings of the 24th international conference on Software engineering - ICSE '02. **Proceedings of...** . p.585, 2002. New York, New York, USA: ACM Press.

Machado, I. C. **RiPLE-TE: A Software Product Lines Testing Process**. Recife, PE: UFPE, 2010. M.Sc. Thesis, Universidade Federal de Pernambuco, August, 2010.

Martin, R. C. **Agile Software Development, Principles, Patterns, and Practices**. Prentice Hall, 2002.

Martins, A. C.; Garcia, V. C.; Almeida, E. S.; Meira, S. R. Enhancing components search in a reuse environment using discovered knowledge techniques. In: 2nd Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS'08). **Proceedings of...** , 2008. Porto Alegre, Brazil.

Mascena, J. C. **ADMIRE: Asset Development Metric-based Integrated Reuse Environment**. Recife, PE, Brazil, 2006. M.Sc. Thesis, Universidade Federal de Pernambuco, May, 2006.

Matinlassi, M. Comparison of software product line architecture design methods: COPA, FAST, FORM, KobrA and QADA. In: 26th International Conference on Software Engineering. **Proceedings of...** . v. 0, p.127-136, 2004. Los Alamitos, CA, USA: IEEE Comput. Soc.

Matinlassi, M. Quality-Driven Software Architecture Model Transformation. In: 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05). **Proceedings of...** . p.199-200, 2005. Washington, DC, USA: IEEE.

Matinlassi, M.; Niemelä, E.; Dobrica, L. Quality-driven architecture design and quality analysis method: A revolutionary initiation approach to a product line architecture. **Vtt Publications**, 2002.

McIlroy, M. D. Mass-produced Software Components. In: J. M. Buxton; P. Naur; B. Randell; Software Engineering Concepts and Techniques. **Proceedings of...** . p.79-85, 1968. NATO Science Committee.

Medeiros, F. M. **SOPLE-DE : An Approach to Design Service-Oriented Product Line Architectures**. Recife, PE: UFPE, 2010. M.Sc. Thesis, Universidade Federal de Pernambuco, May, 2010.

Mendes, R. C. **Search and Retrieval of Reusable Source Code using Faceted Classification Approach**. Recife, PE, Brazil: UFPE, 2008. M.Sc. Thesis, Universidade Federal de Pernambuco, March, 2008.

Mili, H.; Mili, A.; Yacoub, S.; Addy, E. **Reuse-based software engineering: techniques, organization, and controls**. New York, NY, USA: Wiley-Interscience, 2001.

Moraes, M. B. **RiPLE-SC: An Agile Scoping Process for Software Product Lines**. Recife, PE, Brazil: UFPE, 2010. M.Sc. Thesis, Universidade Federal de Pernambuco, August, 2010.

Myllärniemi, V.; Männistö, T.; Raatikainen, M. Quality Attribute Variability within a Software Product Family Architecture. In: Conference on the Quality of Software Architectures. **Proceedings of...** , 2006.

Myllärniemi, V.; Prehofer, C.; Raatikainen, M.; Gurp, J.; Männistö, T. Approach for Dynamically Composing Decentralised Service Architectures with Cross-Cutting Constraints. In: 2nd European conference on Software Architecture. **Proceedings of...** . p.180-195, 2008. Berlin, Heidelberg: Springer-Verlag.

Myllärniemi, V.; Raatikainen, M.; Männistö, T. Inter-organisational Approach in Rapid Software Product Family Development - A Case Study. In: M. Morisio; Reuse of Off-the-Shelf Components, 9th International Conference on Software Reuse, ICSR 2006, Turin, Italy, June 12-15,2006, Proceedings. **Proceedings of...** . p.73-86, 2006. Springer.

Myllärniemi, V.; Raatikainen, M.; Männistö, T. KumbangSec: An Approach for Modelling Functional and Security Variability in Software Architectures. In: First International Workshop on Variability Modelling of Software-intensive Systems, January 2007, Limerick, Ireland. **Proceedings of...** . p.61-70, 2007.

Nascimento, L. M. **Core Assets Development in SPL: Towards a Practical Approach for the Mobile Game Domain**. Recife, PE, Brazil, 2008. M.Sc. Thesis, Universidade Federal de Pernambuco, March, 2008.

Neiva, D. F. **A Requirements Engineering Process for Software Product Lines**. Recife, PE, Brazil: UFPE, 2009. M.Sc. Thesis, Universidade Federal de Pernambuco, August, 2009.

Niemelä, E.; Immonen, A. Capturing quality requirements of product family architecture. **Information and Software Technology**, v. 49, n. 11-12, p. 1107-1120, 2007. Newton, MA, USA: Butterworth-Heinemann.

Northrop, L. M. **Achieving Product Qualities Through Software Architecture Practices**. Pittsburgh, PA, USA, 2004. Software Engineering Institute, Carnegie Mellon University.

Obbink, H.; Müller, J.; America, P.; van Ommering, R. C. **COPA A Component-Oriented Platform Architecting Method for Families of Software-Intensive Electronic Products**. 2000.

Oliveira, T. H. **RiPLE-EM: A Process to Manage Evolution in Software Product Lines**. Recife, PE, Brazil: UFPE, 2009. M.Sc. Thesis, Universidade Federal de Pernambuco, August, 2009.

Olumofin, F. G.; Misic, V. B. Extending the ATAM Architecture Evaluation to Product Line Architectures. In: 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05). **Proceedings of...** . p.45-56, 2005. IEEE.

Parnas, D. L. On the Design and Development of Program Families. **IEEE Transactions on Software Engineering**, v. SE-2, n. 1, p. 1-9, 1976.

Parnas, D. L.; Siewiorek, D. P. Use of the concept of transparency in the design of hierarchically structured systems. **Communications of the ACM**, v. 18, n. 7, p. 401-408, 1975. New York, NY, USA: ACM.

Perepletchikov, M.; Ryan, C.; Frampton, K.; Tari, Z. Coupling Metrics for Predicting Maintainability in Service-Oriented Designs. **2007 Australian Software Engineering Conference (ASWEC'07)**, p. 329-340, 2007. Ieee.

Pohl, K.; Günter Böckle; Linden, F. J. **Software Product Line Engineering: Foundations, Principles and Techniques**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

Robak, S.; Franczyk, B.; Pilitowicz, K. EXTENDING THE UML FOR MODELLING VARIABILITY FOR SYSTEM FAMILIES. **Int. J. Appl. Math. Comput. Sci.**, v. 12, n. 2, p. 285-298, 2002.

Rossel, P. O.; Perovich, D.; Bastarrica, M. C. Feature Model to Product Architectures: Applying MDE to Software Product Lines. In: Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on. **Proceedings of...** , 2009.

Rossel, P. O.; Perovich, D.; Bastarrica, M. C. Reuse of Architectural Knowledge in SPL Development. In: ICSR '09: Proceedings of the 11th International Conference on Software Reuse. **Proceedings of...** . p.191-200, 2009. Berlin, Heidelberg: Springer-Verlag.

Sametinger, J. **Software engineering with reusable components**. New York, NY, USA: Springer-Verlag New York, Inc., 1997.

Santos, E. C.; Durão, F.; Martins, A. C.; et al. Towards an effective context-aware proactive asset search and retrieval tool. In: 6th Workshop on Component-Based Development (WDBC’06). **Proceedings of...** . p.105–112, 2006. Recife, PE, Brazil.

Schmidt, D. C.; O'Ryan, C. Patterns and performance of distributed real-time and embedded publisher/subscriber architectures. **Journal of Systems and Software**, v. 66, n. 3, p. 213-223, 2003.

Shaw, M.; Garlan, D. **Software architecture: perspectives on an emerging discipline**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

Silveira Neto, P. A. **A Regression Testing Approach for Software Product Lines Architectures**. Recife, PE, Brazil: UFPE, 2010. M.Sc. Thesis, Universidade Federal de Pernambuco, July, 2010.

Sinnema, M.; Deelstra, S.; Nijhuis, J.; Bosch, J. COVAMOF: A Framework for Modeling Variability in Software Product Families. In: Software product lines: Third International Conference, SPLC 2004. **Proceedings of...** . v. 3154, p.197-213, 2004.

Snell, J. **Automating business processes and transactions in Web services**. 2002. IBM. Available at: <*http://www.ibm.com/developerworks/webservices/library/ws-autobp/*>. Accessed in: 7/18/2009.

Souza Filho, E. D. **RiPLE-DE: An Approach to Design Software Product Lines Architecture**. Recife, PE, Brazil: UFPE, 2010. M.Sc. Thesis, Universidade Federal de Pernambuco, August, 2010.

Souza Filho, E. D.; Almeida, E. S.; Meira, S. R. **Towards an Approach for Software Product Lines Domain Design**. Recife, PE, Brazil, 2008.

Souza Filho, E. D.; Almeida, E. S.; Meira, S. R. Experimenting a process to design product line architectures. In: EASA'09: Workshop on Empirical Assessment in Software Architecture. **Proceedings of...** , 2009. Cambridge, UK.

Souza Filho, E. D.; Cavalcanti, R. O.; Neiva, D. F.; et al. Evaluating Domain Design Approaches Using Systematic Review. In: ECSA. **Proceedings of...** . p.50-65, 2008.

Svahnberg, M.; Bosch, J. Issues Concerning Variability in Software Product Lines. **Software Architectures for Product Families**, v. 1951, p. 146-157, 2000.

System Security Study Committee; Computer Science and Telecommunications Board; Commission on Physical Sciences Mathematics and Applications; National Research Council. **Computers at risk: safe computing in the information age**. Washington, DC, USA: National Academy Press, 1991.

Voas, J. M.; Miller, K. W. Software Testability: The New Verification. **IEEE Software**, v. 12, n. 3, p. 17-28, 1995.

Weiss, D. M. **Software product-line engineering: a family-based software development process**. Addison-Wesley Longman Publishing Co., Inc., 1999.

Wohlin, C.; Runeson, P.; Host, M.; et al. **Experimentation in software engineering: an introduction**. Boston, MA, USA: Kluwer Academic Publishers, 2000.

van Der Linden, F.; Schmid, K.; Rommes, E. **Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

Appendix A

ARCHITECTURE DOCUMENT TEMPLATE

As part of the Enhanced RiPLE-Design, detailed in Chapter 3, the architecture document template from RiPLE was also modified with the purpose of facilitating the documentation of the product line architectures with variable quality attributes. The next sections list all the information that should be documented.

1 Introduction

<<Here it will be defined a brief introduction of the purpose of this document and its main objective. Scope, definitions, acronyms and abbreviation belong here too. >>

2 References

<<Here, describe the referenced documents, if any>>

3 Technologies Description

<<Describe the technologies (api's, frameworks, libraries) that will be used in the domain development, and will drive the component definition. For example, the use of EJB will drive the components to have entity beans and session beans. The description should be written according to the following format: >>

<<API1 name>>: <<Rationale for the selection>>

<<API2 name>>: <<Rationale for the selection>>

4 Architectural Drivers

<<Describe the functional and non-functional features that will be the most important ones and will drive the architecture (architectural drivers). The effort for adding features that are not part of architectural drivers should be evaluated in order to decide if they should really be added or not. The complete list of non-functional requirements must be provided by the quality scenarios document>>

5 View Documentation

5.1 Structural View

5.1.1 View Description

<<Present a brief description of the current view. Define the types of elements, the relations among them, the significant properties they exhibit, and the constraints they obey for views conforming to this viewpoint >>

<<Also describe the stakeholders and their concerns that this view is intended to address. It should also describe the level of detail this view will use in order to satisfy each specific stakeholder>>

5.1.2 Module Presentation

<<Present the modules and the relationship among them>>

5.1.3 Architectural Styles and approaches

<<Describe the chosen architectural styles and approaches. A rationale for each architectural style must be provided>>

5.1.4 Variability Guidelines

<<Document optional and alternative architectural styles and strategy following the summarized table: >>

<<Strategy>>		**Strategy**	**Feature**	**Affects quality attributes**
Stimulus	**Response**			

5.1.5 Modules Catalog

<<Describe each module in a specific subsection.>>

5.1.5.1 <<Module Name>>

Description:	<<Briefly describe the element's>>
Related Features:	<<Provide the features related with the element>>

5.1.5.2 <<Module Name>> Component Presentation

<<Present the module diagram and description for each module previously elaborated. Use a subsection for each component set >>

5.1.5.1.1 <<Component Name>>

<<Present the component diagram and describe each component in a subsection. Follow the template below>>

Description	<<Give a brief description of the component emphasizing its purpose and its structure>>
Related Features	<<Provide the features related with the component>>
Variability guidelines	<<Describe the patterns and strategies used in the component to handle variability. This section must be provided for each defined component>>

5.2 Behavioral View

<<This section provides a description of the behavior of the domain, using diagrams for representing the iteration among domain classes its variable messages. Diagram should be made for each feature that represents a complete use case>>

5.2.1 View Description

<<Present a brief description of the current view. Define the types of elements, the relations among them, the significant properties they exhibit, and the constraints they obey for views conforming to this viewpoint >>

<<Also describe the stakeholders and their concerns that this view is intended to address. It should also describe the level of detail this view will use in order to satisfy each specific stakeholder>>

5.2.2 <<Feature Name>> Behavioral Presentation

<<This section provides a sequence diagram with interaction of the classes responsible for the execution of the feature>>

5.2.2.1 Variability Guidelines

<<This section has the purpose of identifying the variable messages of the feature sequence diagram and relating it with the feature that drives it. It is used for enable the application

architect to decide which message will be instantiated according to the selected features. This section must be done for each sequence diagram defined>>

5.3 Deployment View

5.3.1 View Description

<<Present a brief description of the current view. Define the types of elements, the relations among them, the significant properties they exhibit, and the constraints they obey for views conforming to this viewpoint >>

<<Also describe the stakeholders and their concerns that this view is intended to address. It should also describe the level of detail this view will use in order to satisfy each specific stakeholder>>

5.3.2 Deployment Presentation

<<Present how the deployment diagrams and description about the deployment of the derived applications>>

5.3.4 Variability Guidelines

<<Document decisions in deployment that are relative to variability achievement>>

Appendix B

ARCHITECTURE EVALUATIN REPORT TEMPLATE

As part of the Enhanced RiPLE-Design, detailed in Chapter 3, the architecture evaluation report template was suggested with the purpose of facilitating the documentation of the product line architectures evaluation. The next sections list all the information that should be documented.

1 Architecture Description

<<A brief description about the architecture. Should include module diagrams.>>

2 Architectural Approaches

<<A description of the main architectural approaches used.>>

3 Scenarios

<<A description of the elicited quality attributes scenarios.>>

4 Risks

<<A description of the risks found during the evaluation.>>

5 Non-Risks

<<A description of the non-risks found during the evaluation.>>

6 Sensitivity Point

<<A description of the sensitivity points found during the evaluation.>>

7 Trade-off Point

<<A description of the trade-off points found during the evaluation.>>

8 Evolvability Point

<<A description of the evolvability points found during the evaluation. This section should be ignored when reporting a product architecture evaluation.>>

9 Evolvability Constraints or Guidelines

<<A description of the evolvability constraints or guidelines found during the evaluation. This section should be ignored when reporting a product architecture evaluation.>>

Appendix C

INSTRUMENTS OF THE EXPERIMENTAL STUDY

As part of the experiment instrumentation, detailed in Chapter 5, 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 C-1 and C-2) was intended to collect data about the subject's background, and the second one (detailed in Table C-3) was created with the purpose of collecting information about the use of the Enhanced RiPLE-Design.

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

Table C-1 Questionnaire for Subject's Background (Part 1)

Questionnaire for Subjects Background	
How many SPL projects have you participated? [ ] None. [ ] Academic. [ ] Industrial.	
How do you define your experience with software reuse?	
Industrial: [ ] None. [ ] Low. [ ] Medium. [ ] High.	Academic: [ ] None. [ ] Low. [ ] Medium. [ ] High.
How do you define your experience with design?	
Industrial: [ ] None. [ ] Low. [ ] Medium. [ ] High.	Academic: [ ] None. [ ] Low. [ ] Medium. [ ] High.
How do you define your experience with domain design?	
Industrial: [ ] None. [ ] Low. [ ] Medium. [ ] High.	Academic: [ ] None. [ ] Low. [ ] Medium. [ ] High.
Please, inform which techniques/methods you know in the context of the reuse, design, domain design and software product line.	
Please, inform which disciplines/courses you have attended in the context of the reuse, design, domain design and software product line.	

Table C-2 Questionnaire for Subject's 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 in applying the activity Identify Architectural Drivers in practice? Which one(s)?
Did you have any difficulties during the Represent variable quality attributes in the feature model task? Which one(s)?
Did you have any difficulties in applying the activity Define Architectural Details in practice? Which one(s)?
Did you have any difficulties in applying the activity Represent Architecture in practice? Which one(s)?
Did you have any difficulties during the Select architectural drivers task? Which one(s)?
Did you have any difficulties during the Choose Architectural Styles task? Which one(s)?
Did you have any difficulties during the Document decision guidelines task? Which one(s)?
Did you have any difficulties in applying the activity Identify Design Decisions in practice? Which one(s)?
Do you thing the RiPLE-Design training was efficacious? Please justify. **[ ] Very Good. [ ] Good. [ ] Regular. [ ] Unsatisfactory.**
Do you thing the RiPLE-Design documentation was sufficient? Please justify.
Which improvements would you suggest for the RiPLE-Design?

Table C-3 Questionnaire for Subject's Feedback