“SPLICE-FeDRE: a SPL Domain Requirements Specification Tool”

By

Karla Malta Amorim da Silva

B.Sc. Dissertation

Federal University of Bahia
ceagmat@ufba.br
wiki.dcc.ufba.br/DCC/

SALVADOR, November/2015

Federal University of Bahia
Computer Science Department
Bachelor's Degree in Computer Science

Karla Malta Amorim da Silva

"SPLICE-FeDRE: a SPL Domain Requirements Specification Tool"

A B.Sc. Dissertation presented to the Computer Science Department of Federal University of Bahia in partial fulfillment of the requirements for the degree of Bachelor in Computer Science.

Advisor: *Eduardo Santana de Almeida*
Co-Advisor: *Raphael Pereira de Oliveira*

SALVADOR, November/2015

I dedicate this dissertation to my family, friends and professors who gave me all necessary support to get here.

Blessed is the man who finds wisdom, the man who gains understanding.

—PROVERBS 3:13

Resumo

Linha de Produto de Software (LPS) é uma metodologia para o desenvolvimento de uma diversidade de produtos de software relacionados e sistemas com uso intensivo de software. Durante o desenvolvimento de uma LPS, uma ampla variedade de artefatos é criada para ser reusável ao longo do desenvolvimento de cada sistema da linha de produto.

Requisitos são um exemplo destes artefatos reusáveis que podem ser instanciados e adaptados para derivar os requisitos de produtos específicos. Gerir requisitos em LPS é uma tarefa árdua porque eles são complexos, interligados, e divididos em comuns, variáveis e requisitos de um produto específico. Assim, o processo de engenharia de requisitos deve ter suporte ferramental para controlar a complexidade e o grande volume de requisitos elicitados.

Neste trabalho, propomos uma ferramenta de suporte para realizar a especificação dos requisitos em LPS de forma sistemática, através do uso de diretrizes, mostrando passo a passo como a especificação deve ser feita.

Palavras-chave: linha de produto de software, especificação de requisitos, ferramenta

Abstract

Software Product Line (SPL) is a methodology for developing a diversity of related software products and software-intensive systems. During the development of a SPL, a wide range of artifacts are created to be reusable throughout the development of each system within the product line.

Requirements are an example of these reusable artifacts that can be instantiated and adapted to derive the requirements for individual products. Managing SPL requirements is a hard task because the are complex, interlinked, and divided into common, variable and product-specific requirements. Thus, the requirements engineering process must be tool-supported to handle complexity and the huge volume of elicited requirements.

In this work, we propose a support tool for performing the specification of the SPL requirements in a systematic way through the use of guidelines, showing step by step how the specification should be done.

Keywords: software product line, requirements specification, tool

Contents

List of Figures

List of Tables

List of Acronyms

CAD	Core Asset Development
CRUD	"Create, Read, Update and Delete"
FeDRE	Feature-Driven Requirements Engineering
FeDRE^2	Feature-Driven Requirements Engineering Evolution
RE	Requirements Engineering
RiSE	Reuse in Software Engineering
ORM	Object-relational mapping
PD	Product Development
SPL	Software Product Line
SPLE	Software Product Line Engineering
SE	Software Engineering
SPLICE	Software Product Line Integrated Construction Environment
VCS	Version control systems
VM	Variability Management

1 Introduction

A Software Product Line (SPL) is outlined as a collection of similar software intensive systems that share a set of common features satisfying the wants of specific customers, market segments or mission. Those similar software systems are developed from a set of core assets, comprised of documents, specifications, components, and other software artifacts that may be reusable throughout the development of each system within the product line (Capilla *et al.*, 2013).

Requirements are typical assets in SPL. They are specified in reusable models, in which commonalities and variabilities are documented explicitly. Thus, these requirements can be instantiated and adapted to derive the requirements for an individual product (Cheng and Atlee, 2007). New products in the SPL will be much simpler to specify, because the requirements are reused and tailored (Clements and Northrop, 2002).

Requirements Engineering (RE) in SPL has an additional cost. Many SPL requirements are complex, interlinked, and divided into common, variable and product-specific requirements (Birk *et al.*, 2003; de Oliveira *et al.*, 2014). The requirements engineering process must be tool-supported to handle complexity and the huge volume of elicited requirements (Birk *et al.*, 2003).

The focus of this dissertation is to provide a support tool for performing the specification of the SPL requirements in a systematic way through the use of guidelines, showing step by step how the specification should be done.

This chapter contextualizes the focus of this dissertation and starts by presenting its motivation in Section 1.1 and a clear definition of the problem in Section 1.2. A brief overview of the proposed solution is presented in Section 1.3, while Section 1.4 describes some aspects that are not directly addressed by this work. Section 1.5 presents the main contributions, Section 1.6 presents the research design and, finally, Section 1.7 outlines the structure of this dissertation.

1.1 Motivation

Within the SPL paradigm, it is very important to perform a good requirements engineering phase, because it is the basis of the SPL paradigm. However, existing tools are not designed to support the requirements engineering process for software product lines. Existing tools support only single product development and therefore lack support for modeling commonalities and variabilities as well as variation points in requirements (Birk *et al.*, 2003).

Some approaches have been proposed to perform the specification and evolution of the SPL requirements in a systematic way through the use of guidelines: Feature-Driven Requirements Engineering (FeDRE) and Feature-Driven Requirements Engineering Evolution (FeDRE^2). These approaches are considered easy to use and useful, however, they do not have a support tool. The lack of tool support can lead to mistakes during the manual execution of the guidelines, moreover, without a tool support these approaches can have problems with scalability.

In this sense, a SPL Requirements Engineering tool is proposed to automatize the SPL requirements specification activities according to the FeDRE approach. This tool is an extension of the tool Software Product Line Integrated Construction Environment (SPLICE) (Cabral *et al.*, 2014), which is an integrated tool for developing SPL.

1.2 Problem Statement

This work investigates the problems of complexity and scalability in SPL requirements specification phase to understand its activities in order to improve automation of these activities. This work promotes effort and mistakes reduction during SPL requirements specification by poviding a SPL Requirements Engineering tool .

1.3 Related Work

Feature-Driven Requirements Engineering (FeDRE) (de Oliveira *et al.*, 2014) was defined and evaluated to aid developers in the Requirements Engineering (RE) activity for SPL development. The FeDRE focus is the requirements specification in the Domain Engineering activity. FeDRE realizes chunks of features from a feature model into functional requirements, which are then specified by use cases. Also, it provides detailed guidelines on how to specify the requirements. A first evaluation of FeDRE was performed through an empirical study within a SPL project, where FeDRE was perceived as easy to learn and useful by the participants.

Software Product Line Integrated Construction Environment (SPLICE) is a web-based SPL life-cycle management tool that provides traceability and variability management and supports most of the SPL process activities such as scoping, testing, version control, evolution, management and agile practices (Vale *et al.*, 2014). SPLICE is part of the Reuse in Software Engineering (RiSE) (Almeida *et al.*, 2004), formerly called RiSE Project, whose goal is to develop a robust framework for software reuse in order to enable the adoption of a reuse program.

The tool SPLICE already supports the specification of features and use cases. In order to accomplish the goal of this dissertation, we propose the extension of SPLICE so that it will support the SPL requirements specification activities stablished in the FeDRE approach. The new version of the tool must enable the requirements engineers involved in this phase, to specify the SPL requirements following the gidelines proposed in the FeDRE approach, while providing guidance, and a reduction of effort and mistakes as the SPL scope scales.

1.4 Out of Scope

The following topics are not considered in the scope of this dissertation:

- **SPL Domain Requirements Evolution**

  Although an approach has already been proposed for the SPL domain requirements evolution phase FeDRE[2], we still do not support this approach, but it is certainly a direction we intend to follow in the future.

- **SPL Application Requirements Engineering**

  In this work we do not consider the SPL Application Engineering process, then our contributions do not cover the SPL Application Requirements Engineering.

- **Non-SPL Tools**

  This work is concerned with Software Product Lines development and tools and environments that support the SPL approach. Non-SPL tools are out of scope.

1.5 Statement of the Contributions

As a result of the work presented in this dissertation, the following contribution can be highlighted:

- **Tool support for a SPL domain requirements specification approach (FeDRE)** We extended the tool SPLICE, a SPL lifecycle management tool and automated Feature-Driven Requirements Engineering (FeDRE), thus improving the automation of Software Product Lines (SPL) requirements engineering phase.

1.6 Research Design

The first step of our work was to investigate the software product line area. This informal study also included to understand the requirements engineering phase for single systems and software product lines. As a result, we could write out the second chapter with some foundations on these subjects.

During the informal study we identified the need for tools that appropriately support the domain requirements engineering phase of software product lines. After choosing a requirements specification approach (FeDRE), we extended an existing SPL lifecycle management tool (SPLICE) providing tool support for this approach.

In order to evaluate the proposed tool, we conducted a survey to identify limitations and needed improvements for the tool.

1.7 Dissertation Structure

The remainder of this dissertation is organized as follows:

- **Chapter 2** reviews the essential topics related to this work: Software Product Lines SPL; requirements engineering; SPL requirements engineering; and Software Product Line Engineering (SPLE) tool support.

- **Chapter ??** describes the tool SPLICE, its architeture and the set of frameworks and technologies used during its development. Also, presents the new functional and non-functional requirements proposed for FeDRE implementation based upon SPLICE.

- **Chapter ??** describes an evaluation of FeDRE implementation.

- **Chapter ??** provides the concluding remarks. It discusses our contributions, limitations, threats to validity, and outlines directions for future work.

2

An Overview on Software Product Lines, Requirements Engineering, SPL Requirements Engineering and SPLE Tool Support

This chapter presents fundamental information for the understanding of four topics that are relevant to this work: software product lines, requirements engineering, and SPL requirements engineering. Section 2.1 discusses the motivation, benefits, and the SPL development process. Section 2.2 presents requirements engineering. Section 2.3 presents SPL requirements engineering. Section 2.4 presents SPLE Tool Support. Finally, Section 2.5 presents a summary of this chapter.

2.1 Software Product Lines

2.1.1 Introduction

Nowadays we experience the age of customization, but it was not always like that. There was a time when goods were handcrafted for individual costumers. Over the years, the number of people who could afford to buy several kinds of products has increased (Pohl *et al.*, 2005). In order to meet this rising demand, the production line was invented, which enabled production for a mass market much more cheaply than individual product.

Customers were satisfied with mass produced products for a while (Pohl *et al.*, 2005), however that kind of product lacks sufficient diversification to meet individual customers' wishes. Individualized products also have a drawback; they are a lot more expensive than standardized products. In that context, the industry was challenged to provide customized products at reasonable costs to satisfy the wishes of specific customers and market segments. The combination of mass customization and common platforms was the key to achieve that goal.

Mass customization is the large-scale production of goods tailored to individual customers' needs. It requires a higher technological investment which leads to higher prices for the individualized products and/or to lower profit margins for the company. The platform approach though, enables manufacturers to offer a larger variety of products and to reduce costs at the same time. A platform is defined as a base of technologies on which other technologies or processes are built. The combination of mass customization and a common platform allows us to reuse a common base of technology and to bring out products in close accordance with customers' wishes (Pohl *et al.*, 2005).

In the software domain, that combination resulted in a software development paradigm called Software Product Line Engineering (SPLE). A Software Product Line (SPL) is a set of software-intensive systems that share a common, managed feature set, satisfying a particular market segment's specic needs or mission and that are developed from a common set of core assets in a prescribed way (Clements and Northrop, 2002).

2.1.2 The Benefits

Developing software under the Product Line Engineering paradigm offers many benefits for a company, some examples follow:

- **Reduction of Development Costs**

  A good reason for applying the Product Line Engineering paradigm is the reduction of costs as the reuse of assets increases. Through the reuse of artifacts from the platform in different systems, the development of each of these systems becomes cheaper. First, the company has to invest in the development of the platform. Also, the way in which the artefacts from the platform will be reused has to be well planned beforehand. Then, from a certain point, called break-even point, the initial investment will be paid off. The precise location of this point is influenced by many characteristics of the company, the market it has envisaged, its customers, expertise, kinds of products, the way the product line is created and others.

  Figure 2.1 shows that the costs to develop a few systems in an SPL approach are higher than in a single systems approach. However, using product line engineering, the costs are significantly lower for larger systems quantities.

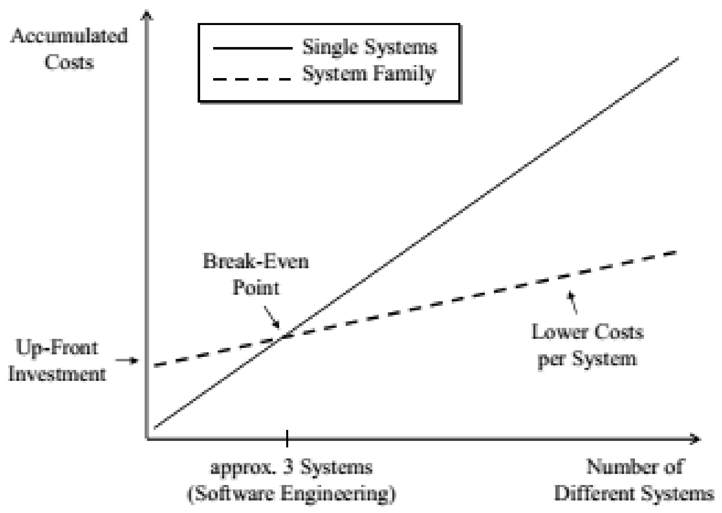


Figure 2.1 Costs for developing systems as single systems compared to product line engineering (Pohl *et al.*, 2005)

- **Quality improvement**

  Creating products under the SPL paradigm improves the quality of all products of a product family. The shared components from the platform are reviewed and tested in many products. They have to work properly in more than one kind of product. The extensive quality assurance indicates a significantly higher opportunity of detecting faults and correcting them, thereby improving the quality of all products (Pohl *et al.*, 2005).

- **Reduction of Time-to-market**

  Another very important success factor for a product is the time to market. SPL engineering demands a high upfront investment, which makes time to market initially higher if compared with to single-systems engineering. However, as the reuse of artefacts grow, the time to market is significantly shortened for new products, as can be seen in Figure 2.2.

- **Reduction of Maintenace Effort**

  When a reusable asset from the platform is changed, this change may be propagated to all products in which it is being used. It usually leads to a simpler and cheaper maintenance and evolution, if compared to maintain and evolve a bunch of single products in a separate way.

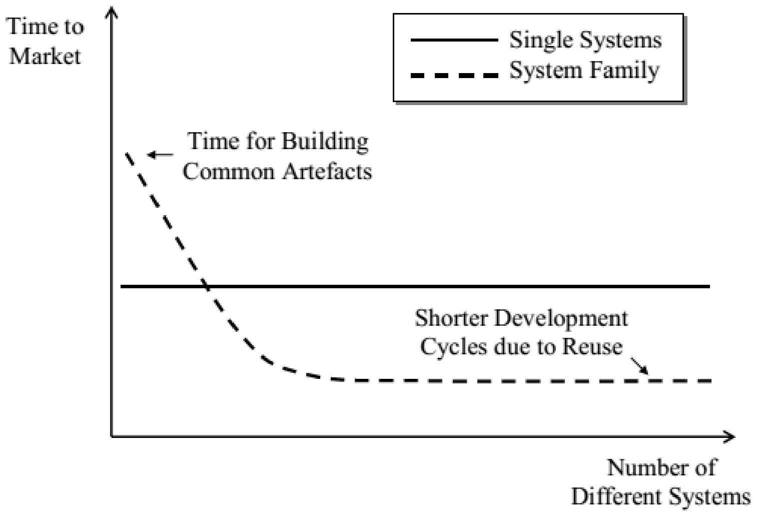


Figure 2.2 Comparison of time to market with and without product line engineering (Pohl *et al.*, 2005)

- **Benefits for the Customers**

  The benefits for the customers are higher quality products at reasonable prices because the production costs become lower in SPL engineering. Besides, products are adapted to their real needs and wishes.

2.1.3 The SPL Development Process

There are a number of different definitions for the Software Product Line (SPL) Development Process on the literature. (Pohl *et al.*, 2005) introduced a framework for SPLE paradigm, shown in Figure 2.3. This framework is divided in two processes:

- **Domain engineering:** This is the process that aims to establish a reusable platform and define the commonality and the variability of the product line. Domain Engineering is composed of five sub-processes: domain requirements, domain design, domain realization, domain testing, and product management (Pohl *et al.*, 2005).

- **Application engineering:** This process is responsible for deriving product line applications from the platform created in domain engineering, where the previously developed components are assembled to compose a product. The application engineering is composed of four sub-processes: application requirements engineering, application design, application realization, and application test (Pohl *et al.*, 2005).

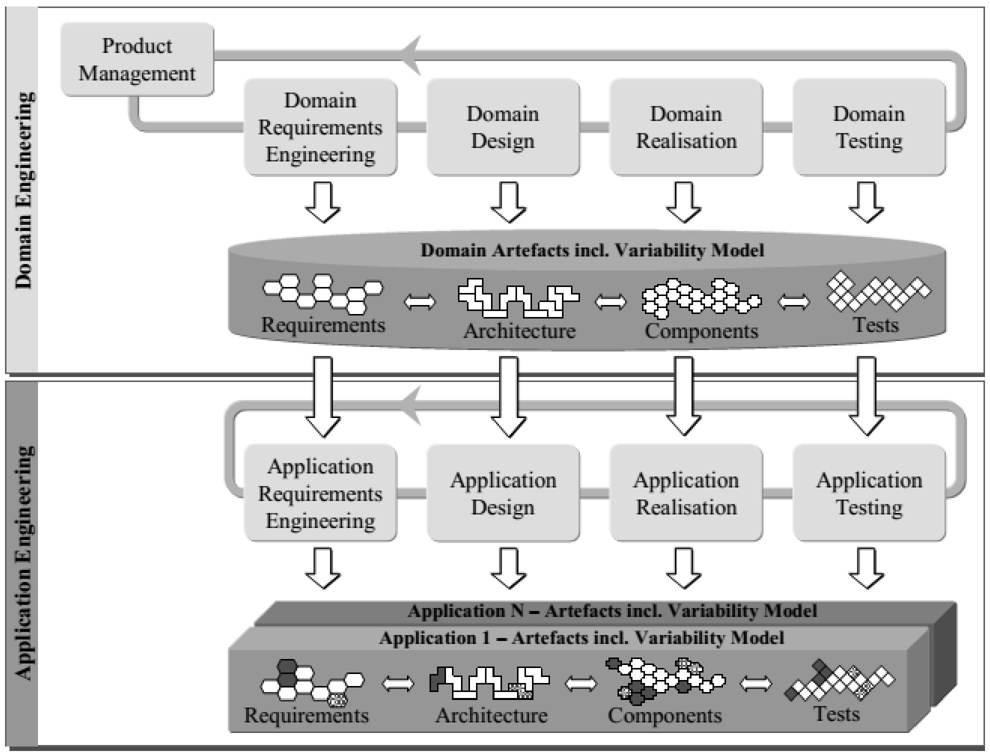


Figure 2.3 The software product line engineering framework (Pohl *et al.*, 2005)

Another popular definition of the Software Product Line (SPL) Development Process can be related to the aforementioned approach. (Clements and Northrop, 2002) defined three essential activities to Software Product Lines: **Core Asset Development (CAD)**, **Product Development (PD)** and **Management activity**, ilustrated in Figure 2.4. In essence, Core Asset Development (CAD) activity is the Domain engineering process, and the Product Development (PD) activity is the Application engineering process. The main difference between these approaches is the Management activity, which is not considered as a process in the first mentioned approach (Pohl *et al.*, 2005).

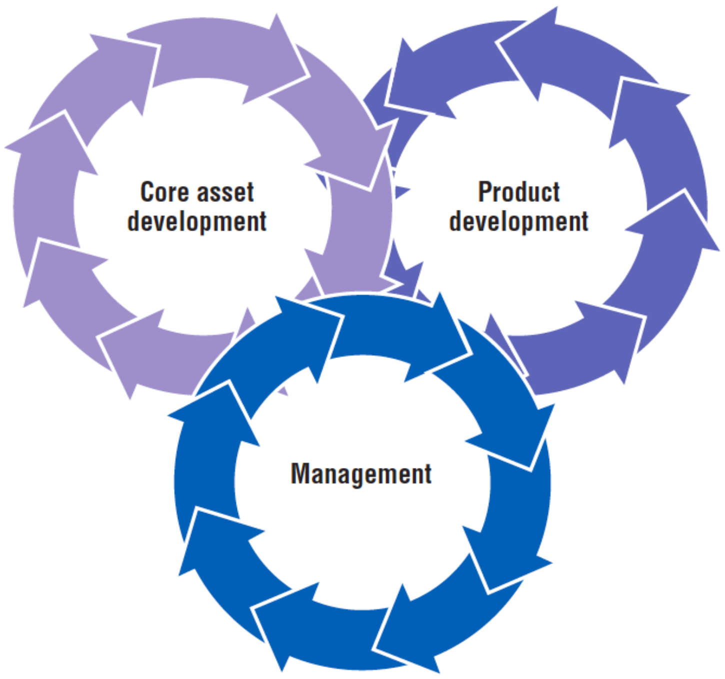


Figure 2.4 SPL Activities (Clements and Northrop, 2002)

Core Asset Development (Domain Engineering)

Core Asset Development (CAD), also called by (Pohl *et al.*, 2005) as domain engineering, is an activity that aims to develop assets to be further reused in other activities. In Figure 2.5, it is shown the core asset development activity, which is interactive, and its inputs and outputs influence each other. The inputs of this activity are product constraints; production constraints; architectural styles; design patterns; application frameworks; production strategy and preexisting assets. This phase is composed of the following sub processes (Pohl *et al.*, 2005):

- **Product Management** deals with the economic aspects associated with the software product line and in particular with the market strategy.
- **Domain Requirements Engineering** involves all activities for eliciting and documenting the common and variable requirements of the product line.
- **Domain Design** encompasses all activities for defining the reference architecture of the product line,
- **Domain Realization** deals with the detailed design and the implementation of reusable software components.
- **Domain Testing** is responsible for the validation and verification of reusable components.

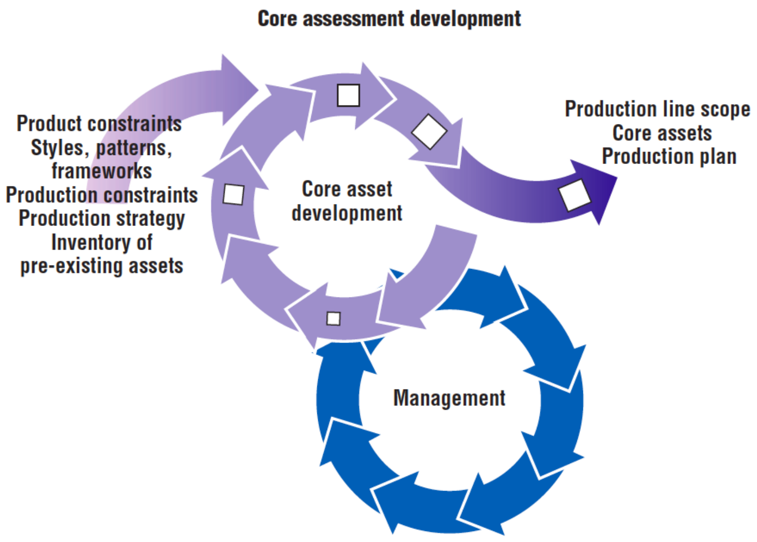


Figure 2.5 Core Asset Development (Clements and Northrop, 2002)

This activity have three outputs: **Product Line Scope**, **Core Assets** and **Production Plan**. The Product Line Scope describes the products that will compose the product line or that the product line can include. This description is recommended to be detailed and well specified, for example, including market analysis activities in order to determine the product portfolio and to encompass which assets and products will be part of the product line. This specification must be driven by economic and business reasons to keep the product line competitive (Capilla *et al.*, 2013).

Core assets are the basis for production of products in the product line. It includes an architecture that will fulfill the needs of the product line, specify the structure of the products and the set of variation points required to support the spectrum of products. It may also include components and their documentation (Clements and Northrop, 2002).

Lastly, the production plan describes how products are produced from the core assets. It details the overall scheme of how the individual attached processes can be fitted together to build a product (Clements and Northrop, 2002). It is what links all the core assets together, guiding the product development within the constraints of the product line.

Product Development (Application Engineering)

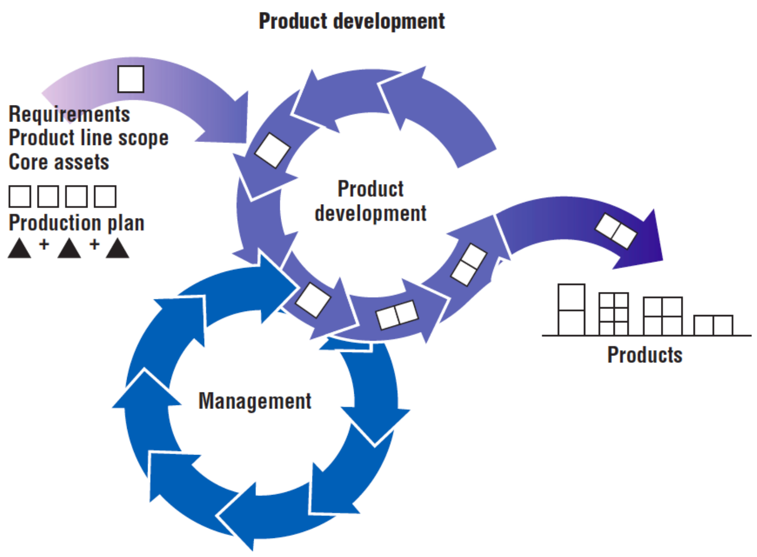


Figure 2.6 Product Development (Clements and Northrop, 2002)

The inputs for this activity are the outputs of the core asset development activity (product line scope, core assets, and production plan) and the requirements specification for individual products as seen in Figure 2.6. The production plan guides how individual products within a product line are constructed using the core assets.

The outputs from this activity should be analyzed by the software engineer and the corrections must be fed back to the Core Asset Development (CAD) activity. During the product development process, some insights happen and it is important to report problems and faults encountered to keep the core asset base healthy.

Management

The management activity is responsible for the production strategy and is vital for success of the product line (Pohl *et al.*, 2005). It is performed in two levels: technical and organizational. The technical management supervise the CAD and PD activities by certifying that both groups that build core assets and products are focused on the activities they are supposed to, and follow the process. The organizational management must ensure that the organizational units receive the right resources in sufficient amounts (Clements and Northrop, 2002).

2.2 Requirements Engineering

Software requirements are descriptions of what the system is expected to do, the services that it must provide and the constraints it must satisfy (Sommerville, 2011). Software requirements are usually classified in a classic way as functional and non-functional. Functional requirements describe what the system must do and non-functional requirements place constraints on how these functional requirements are implemented (Sommerville, 2005).

According to (Sommerville and Kotonya, 1998), Requirements Engineering (RE) is the process by which the software requirements are defined. They state that a process is an organized set of activities that transforms inputs to outputs. Thus, a complete description of a RE process should include what activities are carried out, the structuring or schedule of these activities, who is responsible for each activity and the tools used to support the RE activities.

The RE lifecycle includes requirements elicitation, analysis, negotiation, specification, verification, and management, where (Clements and Northrop, 2002; Sommerville, 2005):

- **Elicitation** identifies sources of requirements information and discovers the users' needs and constraints for the system.
- **Analysis** understands the requirements, their overlaps, and their conflicts.
- **Negotiation** reaches agreement to satisfy all stakeholders, solving conflicts that are identified.
- **Specification** documents the user's needs and constraints clearly and precisely.
- **Verification** checks if the requirements are complete, correct, consistent, and clear.

- **Management** controls the requirements changes that will inevitably arise.

2.3 SPL Requirements Engineering

Requirements are typical assets in SPL. They are specified in reusable models, in which commonalities and variabilities are documented explicitly. Thus, these requirements can be instantiated and adapted to derive the requirements for an individual product (Cheng and Atlee, 2007). During product derivation, for each variant asset, it is decided whether the asset is (or is not) supported by the product to be built. When a domain requirement is instantiated, it can become a concrete product requirement. Thus, new products in the SPL will be much simpler to specify, because the requirements are reused and tailored (Clements and Northrop, 2002).

Deciding which products to build depends on business goals, market trends, technological feasibility, and so on. On the other hand, there are many sources of information to be considered and many trade-offs to be made. The SPL requirements must be general enough to support reasoning about the scope of the SPL, predicting future changes in requirements and anticipated SPL growth.

In practice, establishing the requirements for an SPL is an iterative and incremental effort, covering multiple requirements sources with many feedback loops and validation activities (Chastek *et al.*, 2001). Thus, Requirements Engineering (RE) in SPL has an additional cost. Many SPL requirements are complex, interlinked, and divided into common, variable and product-specific requirements (Birk *et al.*, 2003; de Oliveira *et al.*, 2014). Regarding to single systems, RE for SPL has some differences, such as (Clements and Northrop, 2002; Pohl *et al.*, 2005; Thurimella and Bruegge, 2007):

- **Elicitation** captures anticipated variations over the foreseeable life-cycle of the SPL. RE must anticipates prospective changes in requirements, such as laws, standards, technology changes, and market needs for future products. Thus, its sources of information are probably larger than for single-system requirements elicitation.

- **Analysis** identifies variations and commonalities, and discovers opportunity for reuse.

- **Negotiation** solves conflicts not only from a logical viewpoint, but also taking into consideration economical and market issues. The SPL requirements may require sophisticated analysis and intense negotiation to agree on both common requirements and variation points that are acceptable for all the systems.

- **Specification** documents a SPL set of requirements. Notations are used to represent the product line variabilities and enable the product instantiation.
- **Verification** checks if the SPL requirements can be instantiated for the products, ensuring the reusability of the requirements.
- **Management** must provide a systematic mechanism for proposing changes, evaluating how the proposed changes will impact the SPL, specifically its core asset base. Evolution can affect the reuse and customization, therefore, appropriate mechanisms must be used o manage the variabilities.

In SPL, RE also has influence of several stakeholders that participate of the SPL. Identifying stakeholders that directly influence the RE is essential to define the requirements negotiation participants. They are responsible for resolving conflicts and providing information.

Each stakeholder plays a role with respect to the SPL. Many of the stakeholders that help to define the requirements also use them. These users have different expectations of the outputs of SPL analysis. Some may simply want to confirm that their interests have been represented (e.g., marketers, domain expert and analyst domain). Others (e.g., architects and developers) may want to describe proposed functional and non-functional capabilities, and their commonality and variability across the SPL, thus, those decisions about architectural solutions and asset construction should be taken into account (Chastek *et al.*, 2001).

Several approaches to deal with the definition and specification of functional requirements in SPL development have been proposed over the last few years. Some approaches specify the SPL requirements through features and use cases (Griss *et al.*, 1998; Bayer *et al.*, 2000; Moon *et al.*, 2005; Eriksson *et al.*, 2005; Bonifácio and Borba, 2009; Alférez *et al.*, 2011; Mussbacher *et al.*, 2012; Shaker *et al.*, 2012; de Oliveira *et al.*, 2014). A SPL functional requirement represented as an use case has at least the following fields: identifier, name, description, associated feature(s), pre and post-conditions, and the main success scenario , as shown in Table 2.1. It may also have alternative scenarios, includes/extends relationships, and so on. The feature associated to the use case handles the variability within the SPL.

Table 2.1 SPL Use Case Example (Addapted from (de Oliveira *et al.*, 2014))

***ID:**	Use case identifier		
***Name:**	Use case name		
***Description:**	Use case description		
***Associated feature:**	Feature associated to the use case	**Actor(s) [0..*]:**	Actor associated to the use case
***Pre-condition:**	Use case pre-condition	***Post-condition:**	Use case post-condition
***Main Success Scenario**			
Step	**Actor Action**	**Blackbox System Response**	
Step represented by a number	Actor action	System response	

*Mandatory Fields

However, most of the approaches for specifying SPL functional requirements do not propose guidelines, showing step by step how the specification should be done. This lack of guidelines may lead to some challenges and risks (de Oliveira *et al.*, 2014).

2.3.1 Risks and Challenges

A key RE challenge for SPL development includes strategic and effective techniques for analyzing domains, identifying opportunities for SPL, and identifying the commonalities and variabilities of an SPL (Cheng and Atlee, 2007). Another challenge related to RE is that the applicability of more systematic techniques and tools is limited, partly because such techniques are not yet designed to cope with SPL development's inherent complexities (Birk *et al.*, 2003).

Regarding to the risks associated with RE for SPL, the major risk is failure to capture the right requirements, and their variabilities, over the life of the SPL (Clements and Northrop, 2002). Documenting the wrong or inappropriate requirements, failing to keep the requirements up-to-date, or failing to document the requirements at all, may affects the subsequent activities (architecture, implementation, tests, and so on). They will be unable to produce systems that satisfy the customers and fulfill the market expectations. Moreover, inappropriate requirements can result from the following (Clements and Northrop, 2002):

- **Failure in the communication between core assets requirements development and product requirements development.** The core asset builders need to know the requirements they must build, while the product-specific software builders must know what is expected of them. The lack of communication between these two development stages may lead to inconsistent requirements or even unnecessary variabilities in the requirements.

- **Insufficient generality.** Insufficient generality in the requirements leads to a design that is too fragile to deal with the change actually experienced over the life-cycle of the SPL.

- **Excessive generality.** Excessive generality on requirements leads to excessive effort in producing both core assets (to provide that generality) and specific products (which must turn that generality into a specific instantiation).

- **Wrong variation points.** Incorrect determination of the variation points results in inflexible products and the inability to respond rapidly to customer needs and market shifts.

- **Failure to account for qualities other than behavior.** SPL requirements (and software requirements in general) should capture requirements for quality attributes such as performance, reliability, and security.

2.4 SPLE Tool Support

Since the early days of computer programming, software engineers use a variety of tools to support software development. Software Engineering (SE) tools and environments are becoming progressively important as the demand for software, its diversity and complexity increases. The computer industry is a competitive industry and there is a pressure to produce software at lower costs and faster because time-to-market is a decisive factor for success. Thus, modern software engineering cannot be accomplished without reasonable tool support (Ossher *et al.*, 2000).

The commercial potential of the SPL approach has already been demonstrated in numerous case studies. While product line development is increasingly accepted, professional tool support is still insufficient and represents a key challenge for future research (Pohl *et al.*, 2005; Schmid *et al.*, 2006).

Software Product Line Engineering (SPLE) tool support focuses almost exclusively on a single, cross-cutting aspect of SPLE: variability management Variability Management (VM), or making software and artifacts (such as requirements, tests, and documentation) configurable in a way that they can be developed together, while each product still receives its specifically adapted version (Schmid and Santana de Almeida, 2013). Thus, an effective and efficient variability management VM is the base of the successful reuse of development artifacts (Boutkova, 2011).

Variability Management (VM) tools support four main activities: modeling variability, modeling the relationship between variability and a generic artifact, supporting configuration of generic artifacts, and deriving customized products (Schmid and Santana de Almeida, 2013).

The requirements engineering process must be tool-supported to handle the huge volume of elicited requirements. There are several differences between a single product development and a product line development and therefore a tool must be capable to support that development, including the additional activities that must be performed in the requirements engineering phase. However, existing tools are not designed to support the requirements engineering process for software product lines. Existing tools support only single product development and therefore lack support for modeling commonalities and variabilities as well as variation points in requirements (Birk *et al.*, 2003).

2.5 Summary

In this chapter, we discussed about important concepts to this work: the area of Software Product Line (SPL), Requirements Engineering (RE) , SPL Requirements Engineering and SPLE tool support.

Next chapter presents an extension of Software Product Line Integrated Construction Environment (SPLICE), a web-based, collaborative support tool for the SPL lifecycle steps.

3 SPLICE-FeDRE: a SPL Domain Requirements Specification Tool

3.1 Introduction

In this chapter, we present functional and non-functional requirements for a tool we call SPLICE-FeDRE, and its implementation. The tool is an extension of Software Product Line Integrated Construction Environment (SPLICE), built in order to support and integrate SPL activities, such as, requirements management, architecture, coding, testing, tracking, and release management, providing process automation and traceability across the process.

The remainder of this chapter is organized as follows: Section 3.2 presents the FeDRE approach; Section 3.3 describes the tool SPLICE; Section 3.4 presents the requirements of SPLICE-FeDRE; details of the implementation of SPLICE-FeDRE are discussed in Section 3.5; Section 3.6 shows the tool SPLICE-FeDRE in operation; and, finally, Section 3.7 presents the summary of the chapter.

3.2 FeDRE Overview

The Feature-Driven Requirements Engineering (FeDRE) approach (de Oliveira *et al.*, 2014) for SPL has been defined by considering the feature model as the main artifact for specifying SPL requirements. The aim of the approach is to perform the requirements specification by systematically utilizing the features identified in the SPL domain through the use of guidelines that establish traceability links between features and requirements.

The main activities of the FeDRE approach are: Scoping and Requirements Specification for Domain Engineering. Figure 3.1 shows the activities of FeDRE, which are detailed in this chapter. The following roles are involved in these activities: Domain Analyst, Domain Expert, Market Expert and the Domain Requirements Analyst.

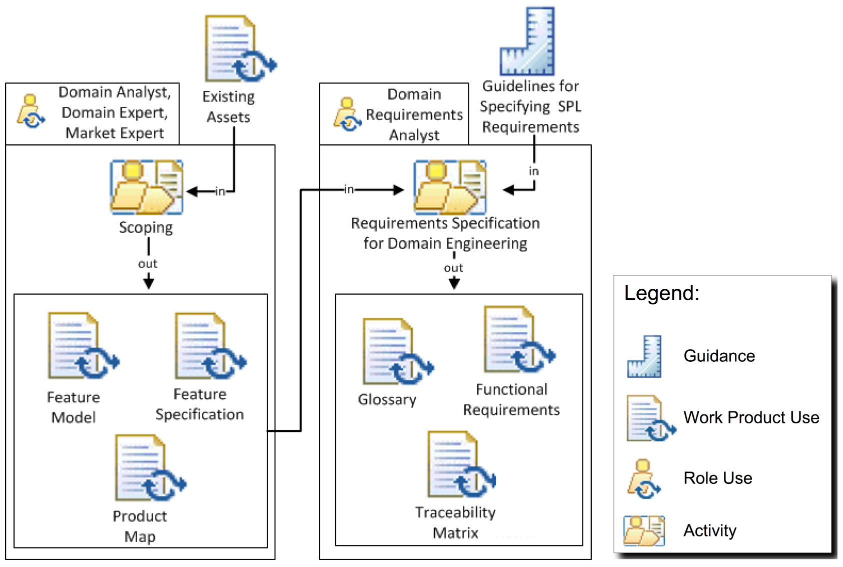


Figure 3.1 Overview of the FeDRE approach (de Oliveira *et al.*, 2014)

3.2.1 Scoping

The first activity performed in FeDRE is the Scoping. This determines not only what products to include in an SPL but also whether or not an organization should launch the SPL. Three main artifacts are produced as a result of the Scoping activity: the Feature Model, the Feature Specification, and the Product Map, using the Existing Assets (if any) as the input artifact. These three artifacts will drive the SPL requirements specification for domain engineering. Each of these artifacts (input and outputs) is detailed below.

Existing Assets

Existing assets (e.g., user manual or existing systems) help the Domain Analyst and the Domain Expert to identify the features and products in the SPL. When they do not exist, a proactive approach can be followed to build the SPL from scratch.

Feature Model

Feature modeling is a technique that is used to model common and variable properties, and can be used to capture, organize and visualize features in the SPL.

Feature Specification

The Domain Analyst is responsible for specifying the features using a feature specification template. This template captures the detailed information of the features and maintains traceability with all the artifacts involved.

Product Map

Each of the identified features is assigned to the corresponding products in the SPL. The set of relationships among features and products produces the Product Map artifact, which describes all the features that are required to build a specific product in the SPL. All these artifacts are the input for the Requirements Specification for Domain Engineering activity, which is described next.

3.2.2 Requirements Specification for Domain Engineering

This activity specifies the SPL requirements for domain engineering. These requirements allow realization of the features and desired products identified in the Scoping activity. The steps required to perform this activity are described in the Guidelines for Specifying SPL Functional Requirements, Sub-section 3.2.3 below.

This activity, seen in Figure 3.1, uses the Feature Model, Feature Specification and Product Map as input artifacts and produces the Glossary, Functional Requirements and Traceability Matrix as output artifacts. Each of these output artifacts is detailed below.

Glossary

The Glossary describes and explains the main terms in the domain in order to provide the stakeholders with a common vocabulary and avoid misconceptions.

Functional Requirements

This artifact contains all the functional requirements identified (common or variable), for the family of products that constitute the SPL. Use cases are used to specify the SPL functional requirements. Each functional requirement has a unique Use case id, a Name, a Description, Associated Feature(s), Pre and Post-Conditions, and the Main Success Scenario. A functional requirement can also be related to an Actor and may have Include and/or Extend relationships with other use case(s).

Traceability Matrix

The Traceability Matrix is a matrix that contains the links among features and the functional requirements.

3.2.3 Guidelines for Specifying SPL Functional Requirements

The purpose of the guidelines is to guide the Requirements Analyst in the specification of SPL functional requirements for domain engineering. The guidelines have been structured to specify functional requirements by addressing the following questions: i) Which features or set of features will be grouped to be specified by use cases? ii) What are the specific use cases for the feature or set of features? iii) Where should the use cases be specified? (when there is a set of features in a hierarchy, do we specify the use cases for each individual feature or only for the parent features?) and iv) How is the use case specified in terms of steps?

Activities, tasks and steps are used in the process of specifying requirements for SPL. Figure 3.2 shows the guidelines with the detailed steps of each task for specifying SPL functional requirements.

3.3 SPLICE Overview

Software Product Line Integrated Construction Environment (SPLICE) (Cabral *et al.*, 2014) is an open source (GNU General Public License), Python, web-based software product line lifecycle management tool, providing traceability and variability management and supporting most of the SPL process activities such as scoping, requirements, architecture, testing, version control, evolution, management and agile practices. This tool assists the engineers involved in the process, with the assets creation and maintenance, while providing traceability and variability management, as well offering detailed reports and enabling engineers to easily navigate between the assets using the traceability links.

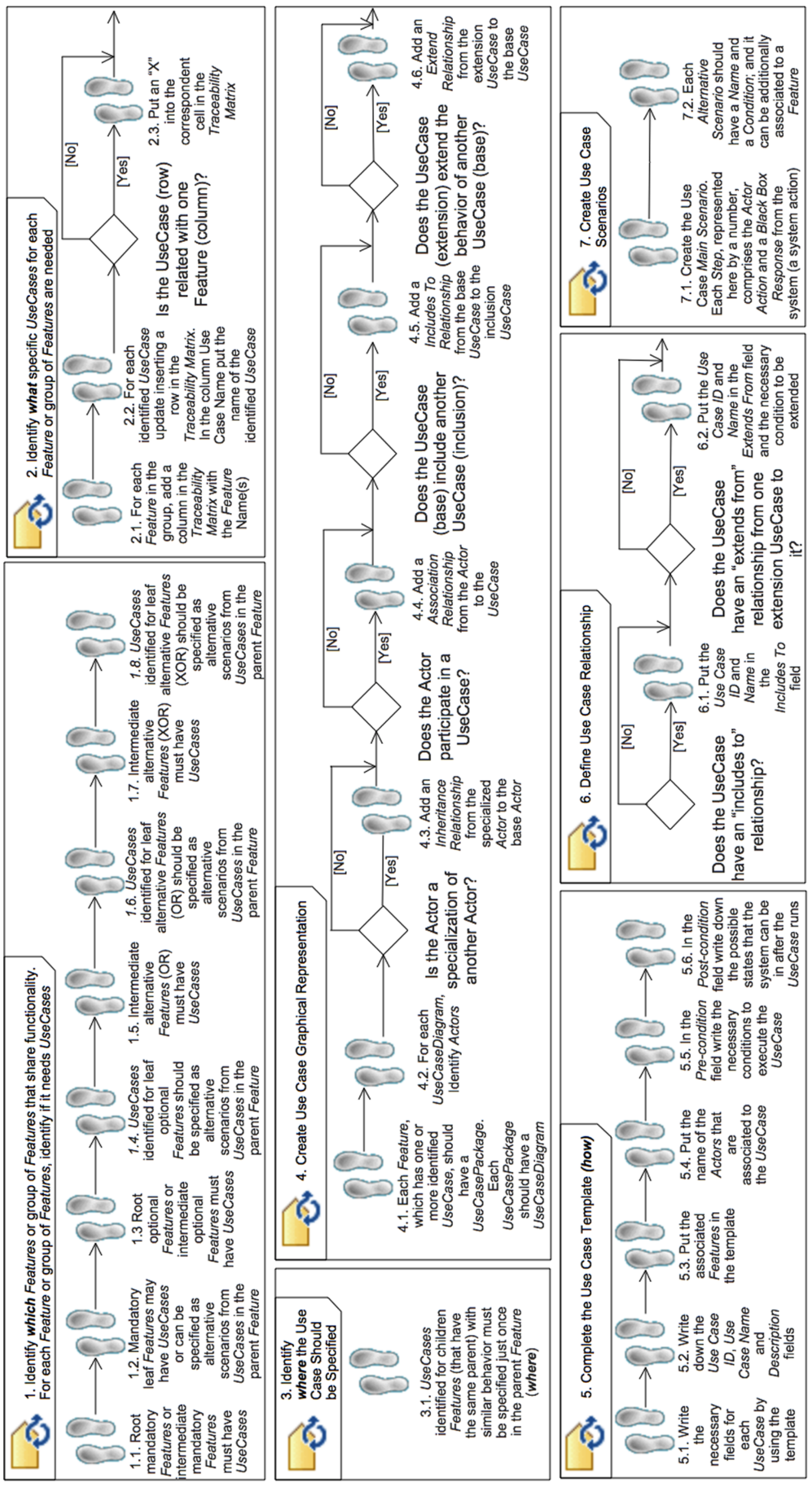

1. Identify which Features or group of Features that share functionality. For each Feature or group of Features, identify if it needs UseCases
1.1. Root mandatory Features or intermediate mandatory Features must have UseCases
1.2. Mandatory leaf Features may have UseCases or can be specified as alternative scenarios from UseCases in the parent Feature
1.3 Root optional Features or intermediate optional Features must have UseCases
1.4. UseCases identified for leaf optional Features should be specified as alternative scenarios from UseCases in the parent Feature
1.5. Intermediate alternative Features (OR) must have UseCases
1.6. UseCases identified for leaf alternative Features (OR) should be specified as alternative scenarios from UseCases in the parent Feature
1.7. Intermediate alternative Features (XOR) must have UseCases
1.8. UseCases identified for leaf alternative Features (XOR) should be specified as alternative scenarios from UseCases in the parent Feature
2. Identify what specific UseCases for each Feature or group of Features are needed
2.1. For each Feature in the group, add a column in the Traceability Matrix with the Feature Name(s)
2.2. For each identified UseCase update inserting a row in the Traceability Matrix. In the column Use Case Name put the name of the identified UseCase
Is the UseCase (row) related with one Feature (column)?
[No]
[Yes]
2.3. Put an "X" into the correspondent cell in the Traceability Matrix
3. Identify where the Use Case Should be Specified
3.1. UseCases identified for children Features (that have the same parent) with similar behavior must be specified just once in the parent Feature (where)
4. Create Use Case Graphical Representation
4.1. Each Feature, which has one or more identified UseCase, should have a UseCasePackage. Each UseCasePackage should have a UseCaseDiagram
4.2. For each UseCaseDiagram, identify Actors
Is the Actor a specialization of another Actor?
[No]
[Yes]
4.3. Add an Inheritance Relationship from the specialized Actor to the base Actor
Does the Actor participate in a UseCase?
[No]
[Yes]
4.4. Add a Association Relationship from the Actor to the UseCase
Does the UseCase (base) include another UseCase (inclusion)?
[No]
[Yes]
4.5. Add a Includes To Relationship from the base UseCase to the inclusion UseCase
Does the UseCase (extension) extend the behavior of another UseCase (base)?
[No]
[Yes]
4.6. Add an Extend Relationship from the extension UseCase to the base UseCase
5. Complete the Use Case Template (how)
5.1. Write the necessary fields for each UseCase by using the template
5.2. Write down the Use Case ID, Use Case Name and Description fields
5.3. Put the associated Features in the template
5.4. Put the name of the Actors that are associated to the UseCase
5.5. In the Pre-condition field write the necessary conditions to execute the UseCase
5.6. In the Post-condition field write down the possible states that the system can be in after the UseCase runs
6. Define Use Case Relationship
Does the UseCase have an "includes to" relationship?
[No]
[Yes]
6.1. Put the Use Case ID and Name in the Includes To field
Does the UseCase have an "extends from" relationship from one extension UseCase to it?
[No]
[Yes]
6.2. Put the Use Case ID and Name in the Extends From field and the necessary condition to be extended
7. Create Use Case Scenarios
7.1. Create the Use Case Main Scenario. Each Step, represented here by a number, comprises the Actor Action and a Black Box Response from the system (a system action)
7.2. Each Alternative Scenario should have a Name and a Condition; and it can be additionally associated to a Feature

Figure 3.2 Guidelines For Specifying SPL Functional Requirements (de Oliveira *et al.*, 2014)

3.3.1 Metamodel

SPLICE proposes a lightweight metamodel, representing the interactions among SPL assets, developed in order to provide a way of managing traceability and variability. The proposed metamodel represents the reusable assets involved in a SPL project, and simplified description of the models is presented next.

- **Scoping Module** comprises the Feature and the Product Model. Many artifacts relates directly with the Feature Model including Use Case, Glossary, User Story and Scope Backlog. A Product is composed of one or more Features.
- **Requirements Module** involves the requirements engineering traceability and interactions issues, considering the variability and commonality in the SPL products. The main object of this SPL phase is Use Case. The concept of User stories is used in this metamodel to represent what a user does or needs to do as part of his or her job function.
- **Testing Module** is composed of a name, description, the Expected result and a set of Test Steps. One Test Case can have many Test Execution that represent one execution of it. The reasoning for the Test Execution is to enable a test automation machinery. The metamodel also represents the acceptance testing with the Acceptance Test and Acceptance Test Execution.
- **Agile Planning Module** contains Sprint Planning models, which are composed of a number of Tickets, a deadline, an objective and a start date. At the end of the sprint, it happens a retrospective, represented in the model by Sprint Retrospective, that contains a set of Strong Points and Should be Improved models that express what points in the spring was adequate, and what needs improvement.

3.3.2 Main Functionalities

The main functionalities of SPLICE include:

- **Metamodel Implementation.** All the screens are completely auto-generated based on the models descriptions, allowing the Software Engineer to easily modify the process. For every model, a complete "Create, Read, Update and Delete" (CRUD) system is created. The SPLICE also provides advanced features such as filtering and classification.

- **Issue Tracking.** SPLICE has a full-featured Issue Tracking. It was extended to implement SPL specific features and to provide traceability between other assets.

- **Traceability.** SPLICE provides total traceability for all assets in the metamodel, and is able to report direct and indirect relations between them. In reports, assets have hyperlinks, enabling the navigation between them.

- **Custom SPL Widgets.** SPLICE has a set of custom widgets to represent specific SPL models. Such as Feature Map, Product Map, and Agile Poker planning.

- **Change history and Timeline.** SPLICE has a rich set of features to visualize how the project is going, where the changes are happening, and who did it. For every Issue or Asset, a complete Change history is recorded.

- **Unified Control Panel.** The tool aggregates the configuration of all external tools in a unified interface. With the same credentials, the user is able to access all SPLICE features, including external tools as Version control systems (VCS).

- **Agile Planning.** The SPLICE supports a set of Agile practices such as effort estimation, where team members use effort and degree of difficulty to estimate their own work. The Features can be dragged by the mouse, and their position is updated in accordance.

- **Automatic reports generation.** SPLICE has the ability of creating reports, including PDFs. The generated report includes a cover, a summary and the set of the chosen artifact related to the product. This format is suitable for the requirements validation by stakeholders. The tool is also able to collect all reports for a given Product, and create a compressed file containing the set of generated reports.

3.4 SPLICE-FeDRE Requirements

In the SPLICE-FeDRE specification, the following functional requirements were defined:

- **FR1 - FeDRE Feature Specification.** The tool should provide a complete CRUD (Create, Read, Update and Delete) for the model Feature that satisfies the FeDRE approach needs. The model Feature should include a unique Feature id, name, description, priority (high, medium or low), type (abstract or concrete), variability (mandatory, optional, OR ou XOR), binding time (compile, runtime), parent feature, glossary, use case diagram, similar feature(s), required feature(s) and excluded feature(s).

- **FR2 - FeDRE Use Case Specification.** The tool should provide a complete CRUD (Create, Read, Update and Delete) for the model Use Case that satisfies the FeDRE approach needs. The model Use Case should include a unique Use case id, name, description, associated feature(s), pre-conditions, post-conditions, and the main success scenario. A Use Case can also be related to an actor and may have include and/or extend relationships with other use case(s), and alternative scenarios.

- **FR3 - FeDRE Guidelines.** The tool shoul implement the FeDRE guidelines for specifying SPL functional requirements. The Features must be stored hierarchically in order to enable FeDRE guidelines implementation. This should be done by storing Features as an n-ary tree data structure that represents the Feature Model.

3.5 SPLICE-FeDRE Implementation

The tool SPLICE was implemented using the Django Framework and the Python programming language. According to (Python Software Foundation, 2014), “Python is a dynamic object-oriented programming language that is used in a wide variety of application domains. It has a very clear, readable syntax, offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days. Many Python programmers report substantial productivity gains and feel the language encourages the development of higher quality, more maintainable code”. Other languages used to developed the tool were JavaScript, CSS, HTML, XML, YAML and make.

According to the SPLICE developer, this choice was motivated by the unprecedented flexibility that the Django Object-relational mapping (ORM) empowers it users, making the metamodel changes effortless. Also, Python is becoming the introductory language for a number of computer science curriculums (Sanders and Langford, 2008). Python is also frequently used on many scientific workflows (Bui *et al.*, 2010) making the project attractive for future data scientists experiments and for undergraduate projects.

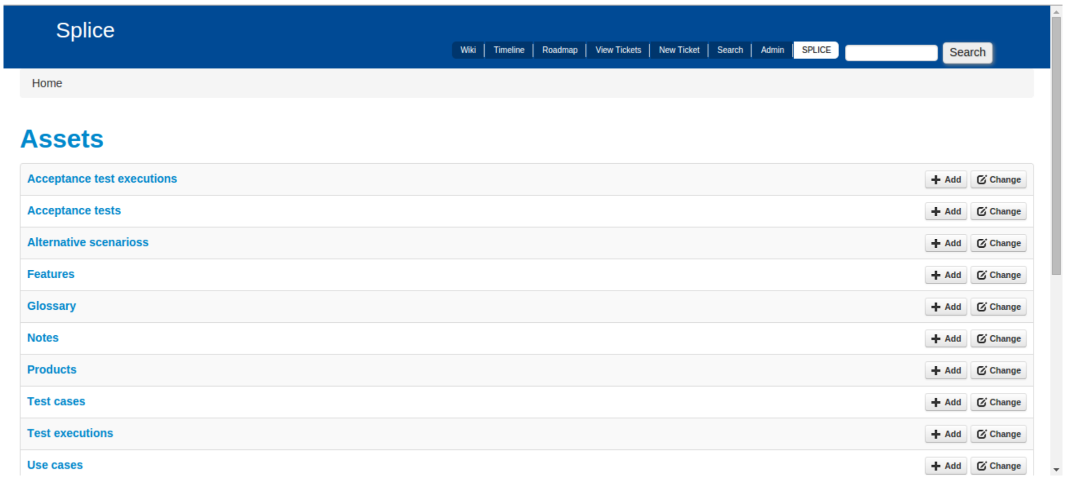
The implementation of the new version called SPLICE-FeDRE adopted the same set of programming languages and framework for development. In SPLICE-FeDRE, the features are stored hierarchically using a modified preorder tree traversal algorithm. We can think of the feature model as an n-ary tree of features, where the root node is a special node that represents de product line. This tree is traversed using a depth-first search algorithm, then the FeDRE flow of activities, tasks and steps is executed for each subtree of the root node, one at a time.

3.6 SPLICE-FeDRE in action

In order to demonstrate how the tool SPLICE-FeDRE works, this section shows the operation of selected features, with a brief description.

3.6.1 Feature Specification

In the home page of SPLICE-FeDRE, seen in Figure 3.3, an assets menu can be used to manage the assets available.



Assets

Acceptance test executions	Add Change
Acceptance tests	Add Change
Alternative scenarioss	Add Change
Features	Add Change
Glossary	Add Change
Notes	Add Change
Products	Add Change
Test cases	Add Change
Test executions	Add Change
Use cases	Add Change

Figure 3.3 Assets screen

By clicking in Features, an user can have access to a complete "Create, Read, Update and Delete" (CRUD), which is also available for all the models listed in the assets menu. Figure 3.4 shows part of the form used to add a new Feature while specifying the Features. It implements the functional requirement *FR1- Feature Specification*. The model Feature includes a unique Feature id, name, description, priority (high, medium or low), type (abstract or concrete), variability (mandatory, optional, OR ou XOR), binding time (compile, runtime), parent feature, glossary, use case diagram, similar feature(s), required feature(s) and excluded feature(s).

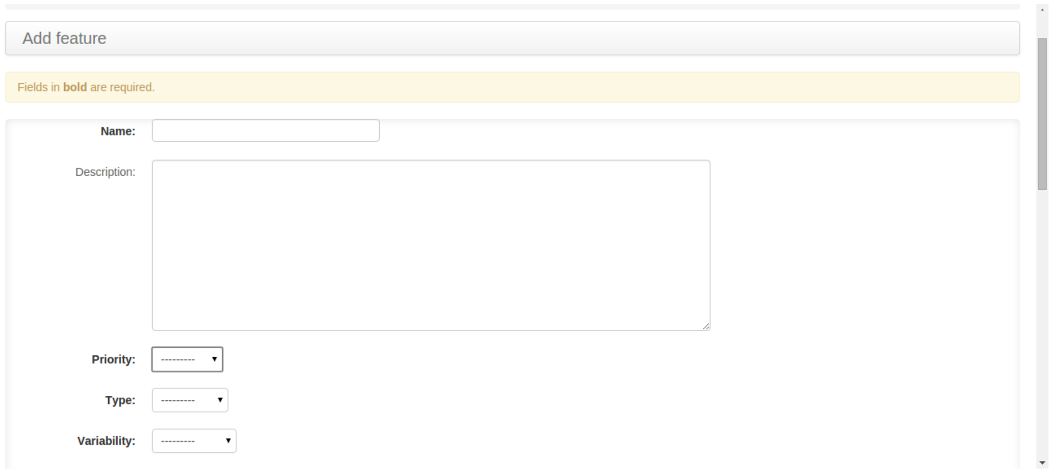


Figure 3.4 Add feature form

3.6.2 FeDRE Guidelines

The FeDRE page is depicted in Figure 3.5. Once the features specification is finished, the user can start the flow of FeDRE guidelines in order to specify the requirements of each subtree of features, one by one. It implements the functional requirement *RF3 – FeDRE Guidelines*.

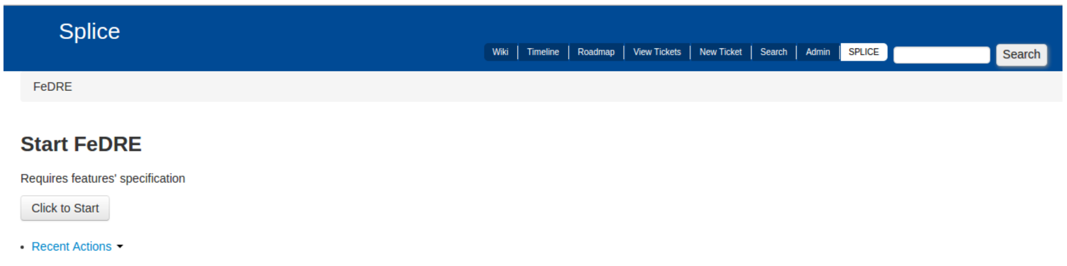
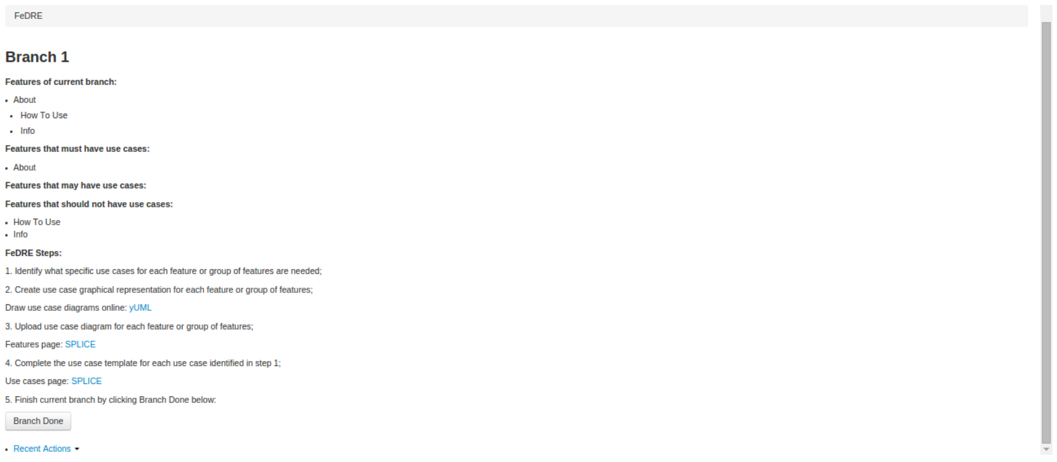


Figure 3.5 FeDRE initial screen

For each branch, the user can see a hierarchy of this branch features, and lists of the features that must have use cases, the features that may have use cases and the features that should not have use cases. Also, it is shown a list of steps to be accomplished before moving to the next branch, as seen in Figure 3.6.



FeDRE

Branch 1

Features of current branch:

- About
  - How To Use
  - Info

Features that must have use cases:

- About

Features that may have use cases:

Features that should not have use cases:

- How To Use
- Info

FeDRE Steps:

1. Identify what specific use cases for each feature or group of features are needed;

2. Create use case graphical representation for each feature or group of features;

Draw use case diagrams online: yUML

3. Upload use case diagram for each feature or group of features;

Features page: SPLICE

4. Complete the use case template for each use case identified in step 1;

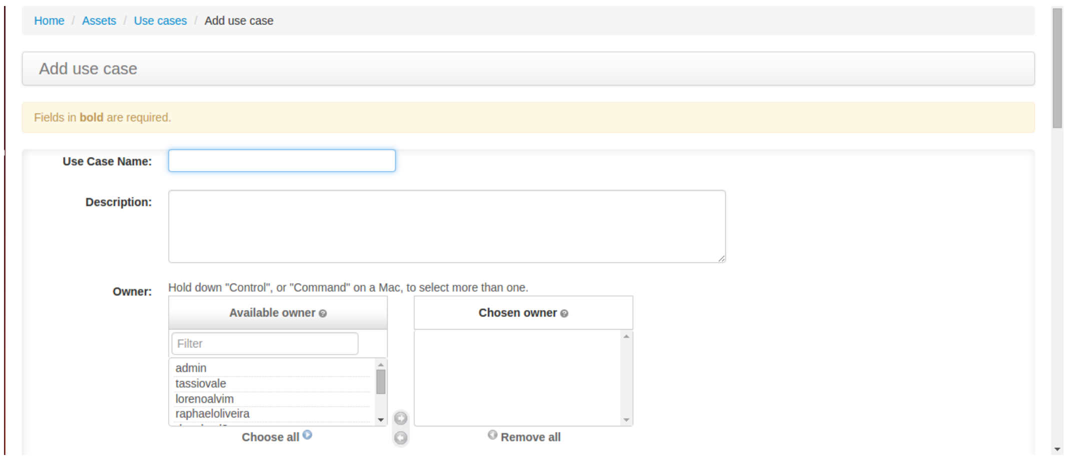
Use cases page: SPLICE

5. Finish current branch by clicking Branch Done below:

Figure 3.6 Branch example

3.6.3 Use Case Specification

As mentioned above, a complete CRUD is also accessible for the model Use Case. Implementing the requirement *FR2 - Use Case Specification*, the model Use Case includes a unique Use case id, name, description, associated feature(s), pre-conditions, post-conditions, and the main success scenario. A Use Case can also be related to an actor and may have include and/or extend relationships with other use case(s), and alternative scenarios. See Figure 3.7 below:



Home / Assets / Use cases / Add use case

Add use case

Fields in **bold** are required.

Use Case Name:

Description:

Owner: Hold down "Control", or "Command" on a Mac, to select more than one.

Figure 3.7 Add use case form

3.7 Summary

In this chapter, it was presented the Feature-Driven Requirements Engineering (FeDRE) approach and the Software Product Line Integrated Construction Environment (SPLICE), a web-based tool for SPL lifecycle management, and how it was extended to automate the FeDRE approach. Next chapter presents an evaluation of SPLICE-FeDRE performed during the development of the tool.

4
Case study : RescueME SPL

If we're facing in the right direction, all we have to do is keep on walking.

—JOSEPH GOLDSTEIN (The Experience of Insight)

4.1 Introduction

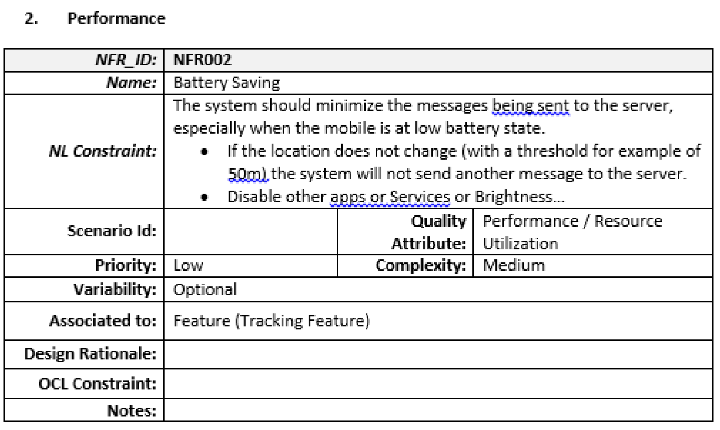
In this chapter, will be described a case study where the SPLICE tool was tested during a real SPL development. The study was conducted inside a research laboratory and included the migration from a manual Software Engineering process to the SPLICE tool and the proposed metamodel. To validate the tool, we proposed some research questions and conducted a survey.

This Chapter is organized as follows: Section 4.2 defines this case study; in Section 4.3 the planning of the case study takes place; Section 4.4 shows the analysis and interpretation of the results; Section 4.5 analyses the possible threats to validity of our study; Section 4.6 describes the lessons learned during this study; and Section 4.7 presents the findings and summarizes this chapter.

4.2 Definition

4.2.1 Context

During the months of June and November 2013, we performed a case study in the "National Institute of Software Engineering (I.N.E.S)", a Software Engineering research laboratory, composed of 11 Ph.D. candidates. The laboratory developed a SPL called RescueMe, which was built following a SPL agile process. The RescueMe is a product line developed in Objective-C for iOS devices. RescueMe is designed to to help its users in dangerous situations. The start screen of the application consists of a button that when pressed send messages to the user contacts asking for help. RescueMe get the contacts from the phone address book or from social networks, such as Facebook and Twitter, depending on the used version.



2. Performance

NFR_ID:	NFR002		
Name:	Battery Saving		
NL Constraint:	The system should minimize the messages being sent to the server, especially when the mobile is at low battery state. • If the location does not change (with a threshold for example of 50m) the system will not send another message to the server. • Disable other apps or Services or Brightness…		
Scenario Id:		Quality Attribute:	Performance / Resource Utilization
Priority:	Low	Complexity:	Medium
Variability:	Optional		
Associated to:	Feature (Tracking Feature)		
Design Rationale:			
OCL Constraint:			
Notes:			

Figure 4.1 Older non-functional requirement

RescueMe had an iterative and incremental development, carried out by four developers, with a face-to-face meeting at the end of each sprint. These meetings were responsible for evaluating results, and planing the next sprint. The group manually maintained the SPL process based on a set of external tools. Before using SPLICE, they used the SourceForge[1] service for issue tracking and Version control systems (VCS). All the requirements artifacts where maintained using text documents questionnaires, as can be seen on Figure 4.1. The SPLICE was introduced to manage the SPL process and all artifacts were migrated to it. After the migration, the development continued to use only the SPLICE to manage the application lifecycle.

4.2.2 Research Questions

The SPLICE is a complete environment for Software Product Lines development, and many aspects can be analyzed. However, in this work the main objective is to analyze the

[1]http://www.sourceforge.net

effectiveness of the traceability through integration of the technologies on the tool, and how the tool influences the Software Product Line (SPL) lifecycle. In order to evaluate these aspects in our proposal, we defined three case study research questions:

- **From the stakeholder perspective, how the traceability is solved with the SPLICE?**

  Rationale: The goal is to verify if the tool can provide traceability support, and from the stakeholder perspective, how it was solved.

- **How positively the tool improved the application lifecycle?**

  Rationale: The tool manages the whole process, and this question verifies if the outcome is positive.

- **How negatively the tool impacted the application lifecycle?**

  Rationale: In this question, the stakeholder evaluates how negatively the SPLICE interfered with the process.

4.3 Data collection

In this case study, we selected surveys as a data collection instrument. Survey is a data-gathering mechanism in which participants answers questions or statements previously developed and according to Kitchenham and Pfleeger (2008) , they are probably the most commonly used instrument to gather opinions from experts. Expert Surveys is a kind of study conducted through a research applied to people who are considered experts in a field, in order to identify speculations, guesses and estimates, which may serve as a cognitive input in some decision process (Chhibber *et al.*, 1992)

The survey design is based on Kitchenham and Pfleeger (2008) guidelines and is composed of a set of personal questions, closed-ended and open-ended questions related to the research questions. The remain of this section contains the overall process applied in this study and the methodology.

4.3.1 Survey Design

In this survey, we used the cross-sectional design, which participants were asked about their past experiences at a particular fixed point in time. This survey was performed as a self-administered printed questionnaire. We collected all data for analysis in January, 2014.

4.3.2 Developing the Survey Instrument

The questionnaire, which composes the survey, was defined based on the steps defined in Kitchenham and Pfleeger (2008) , and although they suggest the use of closed questions in self-administered questionnaires, our question was composed of closed questions with an open field to justification, as the tool usage is something very subjective and we wanted to capture the researcher opinion. The questionnaire was composed of three personal questions, eight closed questions with justification fields, and three open questions. The closed questions were formulated to measure and quality the data, while getting personal feedback. The open questions were built to collect the experts' experiences and the impressions about the tool.

4.3.3 Evaluating the Survey Instrument

After defining the questions for our survey, it is necessary to evaluate it, in order to check whether is enough to address the preliminary stated goals. Evaluation is often called pre-testing and according to Kitchenham and Pfleeger (2008) has several different goals :

- To check that the questions are understandable.
- To assess the likely response rate and the effectiveness of the follow-up procedures.
- To evaluate the reliability and validity of the instrument.
- To ensure that our data analysis techniques match our expected responses.

We validated the questionnaire by asking Software Engineering researchers from the RiSE research group to analyze and suggest modifications. The suggestions where discussed and the questionnaire modified accordingly. It is important to reinforce that the authors were not considered during the pilot test. The questionnaire can be seen in the Appendix 1.

4.3.4 Expert Opinion

An important step on expert opinion survey is looking for expert judgments, since an expert is a knowledgeable authority on the research domain [Chhibber et al., 1992]. Considering this, the subjects should be chosen based on the most relevant expertise, most accurate estimates or judgments. Our selection criteria was: The expert should have a considerable knowledge demonstrated through academic experience or Industry

Name	Occupation	SE experience	SPL experience
Raphael Oliveira	Ph.D. candidate	10 years	6 years
Tassio Vale	Ph.D. candidate	6 years	4 years, on academic and industrial projects.

Table 4.1 Experts Selected

experience. They also must have a strong background in Software engineering and more specifically in Software Product Line (SPL). Another very important and limiting criteria is that the expect should have availability to use the tool and be involved with the specific analyzed case study , the RescueME.

We did not use any sampling method as suggested by Kitchenham and Pfleeger (2008), because the target population was already very small. At the beginning, we selected four SPL experts, as potential candidates. However, one of them did not receive the invitation emails, and another did not answer our invitation. Thus, we had two experts that participated in this survey.

The experts who answered the questionnaire are showed in Table 4.1, all the experts have more than 5 years of experience in Software Engineering, and more than 4 years on SPL specifically . The purpose of this table is to show that the SPL experts are people that have been involved in the SPL community, and their names are showed in order to increase the confidence of our research.

4.3.5 Analyzing the data

In order to collect the data, the experts filled a printed questionnaire. From the three invited researchers only two reported their answers. After designing and running the survey, the next step was to analyze the collected data. The main analysis procedure was to check all responses, tabulate the data, identify the findings and classify the options.

4.4 Results

In this section the analysis of the collected data are presented, discussing the given answer for each question. Three of the fourteen questions were personal questions such as name and experience and where already reveled on the previous section.

Tool usage difficulties

Considering the question **There was any difficult during the execution of some activity in the tool ?**, one expert declared in that did not found any problem using the tool, however another subject indicated that *"The SPLICE assets management should be the initial screen"*. Actually the initial screen is the *Wiki*.

Tool interference in workflow

In the question: **How the usage of the tool altered your workflow?** , one expert pointed that the tool did not altered his workflow, since the tool *"offered the option to specify the needed asserts during the SPL lifecycle."*, interestingly another expect pointed the exactly opposite and complained that *"The tool should provide a way to customize the necessary assets for each project, so it can be used in several SPL projects"*.

Assets creation difficulties

Considering the question **Did you have any problems creating assets ?** , All respondents pointed that they had no problems during the assets creation.

Blame changes

Considering the question **Did the tool gave you enough information to identify who did a specific modification in an asset ?** , both experts declared that this information was clearly presented and complete.

Traceability

In the question **Do you consider that tool aided the traceability among the assets?** all experts replied *"Yes"*. We asked for justification and one said that *"The tool deals in an easy way with the traceability among assets. For example, it is easy to check what products have a determinate Feature, since there is a field in the feature details that shows this information (Products with this Feature)"*. Another respondent stated that (the tool) *"...provides links for different types of assets to be related in the tool, and it provides, for instance, the ability to analyze the impact of changes"*. From the answer of the respondents, we could assume that the tool did address the traceability problem, providing a level of traceability to the managed assets.

Expected traceability links

All respondents indicated in the question **Did you expect any traceability link not available in the tool? If so, which one(s) ?** , that they did not expected any other traceability link, from what is offered by the tool.

Traceability links navigation

Considering the question **What is your opinions on navigating among traceability links ?**, one expert pointed that the Traceability links navigation in SPLICE was intuitive and *"... simple because each traceability link has the 'asset name' on the defined value (i.e. compile time) that lead to more details of the ''asset' or 'define value' when clicked"*.

However, another expert replied that *"The traceability links make software engineers to think the assets as part of a unique SPL. When these links are not available, the assets seems disconnected and I can't have an overall view of the SPL inconsistencies it might cause when specifying new assets."*. This question have a problem, as it should state more clearly that we want to know about the options on navigating among traceability links using the SPLICE tool. We do not know if he is complaining about the tool, or traceability navigation in general. We do not think that is directly related to the too because in a previous question he declared that no traceability link was missing in the SPLICE.

Reporting

In the question **In your opinion, the reports generated were satisfactory ?** , one respondent fully agreed that the generated reports were satisfactory. Conversely, another respondent stated that is missing the *"The analysis of impact for changes in the SPL assets. Based on an asset that is changed, I can visualize the impacted assets and the effort to make modification in the SPL"*. The SPLICE do have impact change analysis, but just for assets deletion, this is a point that can be improved.

Tool Helpfulness

Considering the question **Do you think that the proposed tool would aid you during a SPL process ? Would you spontaneously use the tool hereafter?** , all experts fully agreed that the tool was helpful and they would use in another project. One expert even stated that *"One of the biggest problems within SPL documentation when performed in spreadsheets or documents files is to keep updated the traceability information among the SPL assets after evolving them. The proposed tool helps in dealing with this problem"*.

Positive points

We asked the experts the question **What are the positive points of using the tool?** , and the positive points of the tool according to them are:

- Traceability among assets.
- Version control system.
- Change history.
- Report generation.
- Integration with variability mechanisms.
- Integration of different software engineering support tools.
- Feature oriented approach.

Most of them mentioned positive points are functional requirements of the tool, which could demonstrate that some of our objectives was been fulfilled by the SPLICE.

Negative points

In Contrast with the previous question, we also asked **What are the negative points of using the tool?**. Only two points were mentioned, as follow:

- **Pre-defined set of assets is available**. *"If a SPL manager decides to include a new asset, a new version of the tool must be deployed"*.
- **No variability mechanisms in source code**. *"The integration with variability mechanisms in source code is not available. Then, the derivation is not complete"*.

Suggestions

As a final point, we asked **Please, write down any suggestion you think might be useful**. One expert suggested to *"Turn the tool flexible to include or remove assets for a specific project"*. The other expert suggest that *"The analysis of change impacts can be very useful. You can use the estimatives (Story points, for example) to calculate the effort spent to change a feature, user story and so on"*. Although one of our non-functional requirement is to provide *"Metamodel flexibility"*, we implement this on a compile time. Both suggestion has been noted to future improvement to the tool.

4.5 Threats to validity

There are some threats to the validity of our study, which were briefly described and discussed:

- Research questions: The research questions we defined cannot provide complete coverage of all the features covered by the tool. We considered just some important point: traceability, advantages and disadvantages.

- Sample size: The most obvious threat to internal validity is the sample size, which was very small. We also included a participant who contributed to the tool, and our results may be biased. Fowler (2002) suggests that there is no equation to exactly determine the sample size, but we recognize a more ample study help to generalize the case study results.

4.6 Findings

Analyzing the answers, just one developer reported difficulties during the tool usage. He reported problems with the fact the initial screen is the collaborative document and not the assets screen, which is hidden behind a menu item. We will make the software more customizable in the future. No users reported difficulties creating asserts or identifying who performed modifications to assets. No major usability problem was found, and all were able to use and evaluate the tool without supervision. This can indicate that the tool fulfilled the requirements of Usability and Accountability.

All the experts explicitly declared that the tool was useful, aided on the assets traceability, provided all the traceability links they wanted and offered a valuable set of features. They also stated that would, spontaneously, use the tool in future SPL projects.

The experts also mentioned some points of improvement during the survey. One interesting problem vocally expressed by one expert was inability to configure the process and the metamodel. This is a non-functional requirement of the tool, and the SPLICE architecture is very capable of it. However, this require editing some files manually, so visual editor should be added to the tool to address this problem.

Some other problems includes the need to a better change impact analysis and integration with variability in source code, to perform product line derivation. The latter future was postponed because of time limitations of an undergraduate project, and is planned for a future version.

4.7 Summary

This chapter presented the definition, planning, operation, analysis and interpretation of a case study to evaluate the SPLICE tool. The case study was conducted inside the research laboratory "National Institute of Software Engineering (I.N.E.S)", and a survey was administered to the experts of the laboratory. After concluding the case study and the questionnaires, we gathered information that can be used as a guide to improve the tool, and an indicator about the actual status of the tool. The results of the experiment pointed out that the SPLICE address the traceability problem and was considered useful to all experts. However, some points of improvements were raised, that we plan to fix on future versions. In addition, the case study design and the lessons learned were also presented.

Next chapter presents the concluding remarks and future work of this dissertation.

Bibliography

Alférez, M., Lopez-Herrejon, R. E., Moreira, A., Amaral, V., and Egyed, A. (2011). Supporting consistency checking between features and software product line use scenarios. In *Top Productivity through Software Reuse*, pages 20–35. Springer.

Almeida, E. S., Alvaro, A., Lucrédio, D., Garcia, V. C., and Meira, S. R. L. (2004). Rise project: Towards a robust framework for software reuse. In *IEEE International Conference on Information Reuse and Integration (IRI)*, pages 48–53, Las Vegas, NV, USA.

Bayer, J., Muthig, D., and Widen, T. (2000). Customizable domain analysis. In *Generative and Component-Based Software Engineering*, pages 178–194. Springer.

Birk, A., Heller, G., John, I., Joos, S., Muller, K., Schmid, K., and von der Massen, T. (2003). Report of the gi work group" requirements engineering for product lines.

Bonifácio, R. and Borba, P. (2009). Modeling scenario variability as crosscutting mechanisms. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 125–136. ACM.

Boutkova, E. (2011). Experience with variability management in requirement specifications. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 303–312. IEEE.

Bui, P., Yu, L., and Thain, D. (2010). Weaver: Integrating distributed computing abstractions into scientific workflows using python. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 636–643, New York, NY, USA. ACM.

Cabral, B., Vale, T., and Almeida, E. (2014). Splice: Software product lines integrated construction environment. *V Congresso Brasileiro de Software: Teoria e Prca (CBSoft), Sesse Ferramentas*.

Capilla, R., Bosch, J., and , K. (2013). *Systems and Software Variability Management: Concepts, Tools and Experiences*. SpringerLink : Bücher. Springer.

Chastek, G., Donohoe, P., Kang, K. C., and Thiel, S. (2001). Product line analysis: a practical introduction. Technical report, DTIC Document.

Cheng, B. H. and Atlee, J. M. (2007). Research directions in requirements engineering. In *2007 Future of Software Engineering*, pages 285–303. IEEE Computer Society.

Chhibber, S., Apostolakis, G., and Okrent, D. (1992). A taxonomy of issues related to the use of expert judgments in probabilistic safety studies. *Reliability Engineering System Safety*, **38**(1–2), 27 – 45.

Clements, P. and Northrop, L. (2002). *Software Product Lines: Practices and Patterns*. The SEI series in software engineering. Addison Wesley Professional.

de Oliveira, R. P., Blanes, D., Gonzalez-Huerta, J., Insfran, E., Abrahão, S., Cohen, S., and de Almeida, E. S. (2014). Defining and validating a feature-driven requirements engineering approach. *Journal of Universal Computer Science*, **20**(5), 666–691.

Eriksson, M., Börstler, J., and Borg, K. (2005). The pluss approach–domain modeling with features, use cases and use case realizations. In *Software Product Lines*, pages 33–44. Springer.

Griss, M. L., Favaro, J., and Alessandro, M. D. (1998). Integrating feature modeling with the rseb. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 76–85. IEEE.

Kitchenham, B. and Pfleeger, S. (2008). Personal opinion surveys. In F. Shull, J. Singer, and D. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 63–92. Springer London.

Moon, M., Yeom, K., and Chae, H. S. (2005). An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. *Software Engineering, IEEE Transactions on*, **31**(7), 551–569.

Mussbacher, G., Araújo, J., Moreira, A., and Amyot, D. (2012). Aourn-based modeling and analysis of software product lines. *Software Quality Journal*, **20**(3-4), 645–687.

Ossher, H., Harrison, W., and Tarr, P. (2000). Software engineering tools and environments: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 261–277. ACM.

Pohl, K., Böckle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles, and Techniques*.

Python Software Foundation (2014). Python Programming Language. http://www.python.org. Last access on Jan/2014.

Sanders, I. D. and Langford, S. (2008). Students' perceptions of python as a first programming language at wits. *SIGCSE Bull.*, **40**(3), 365–365.

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

Schmid, K., Krennrich, K., and Eisenbarth, M. (2006). Requirements management for product lines: extending professional tools. In *Software Product Line Conference, 2006 10th International*, pages 10–pp. IEEE.

Shaker, P., Atlee, J. M., and Wang, S. (2012). A feature-oriented requirements modelling language. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pages 151–160. IEEE.

Sommerville, I. (2005). Integrated requirements engineering: A tutorial. *Software, IEEE*, **22**(1), 16–23.

Sommerville, I. (2011). *Software Engineering*. Addison Wesley, 9 edition.

Sommerville, I. and Kotonya, G. (1998). *Requirements engineering: processes and techniques*. John Wiley & Sons, Inc.

Thurimella, A. K. and Bruegge, B. (2007). Evolution in product line requirements engineering: A rationale management approach. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 254–257. IEEE.

Vale, T., Cabral, B., Alvim, L., Soares, L., Santos, A., Machado, I., Souza, I., Freitas, I., and Almeida, E. (2014). Splice: A lightweight software product line development process for small and medium size projects. In *Software Components, Architectures and Reuse (SBCARS), 2014 Eighth Brazilian Symposium on*, pages 42–52. IEEE.

Appendix

A Evaluation Instruments

A.1 Form for Expert Survey

Name:

What is your experience with Requirements Specification (in months/years)?

What is your experience with Software Product Lines (in months/years)?

Did you have any difficulty during the execution of any activity in the tool?

[ ] Yes. [ ] No.

In case you answered Yes, detail the difficulty encountered:

Did you have any problems creating use cases?

[ ] Yes. [ ] No.

In case you answered Yes, describe the problems encountered:

Do you think that the proposed tool would aid you during a SPL Requirements Engineering process? Would you spontaneously use the tool hereafter?

Do you think the proposed tool is useful to handle the complexity of SPL Requirements Engineering process?

Do you think the proposed tool is useful to handle scalability problems during a SPL Requirements Engineering process?

What are the positive points of using the tool?

What are the negative points of using the tool?

Please, write down any suggestion you think might be useful.