"TITLE"

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

"TITLE"

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.

Agradecimentos

Agradeço a Deus por ter me dado oportunidade de realizar este trabalho, saúde e energia para superar todas as dificuldades. Também por cada uma das pessoas citadas abaixo, que de alguma forma contribuíram para que essa conquista fosse possível.

Agradeço aos meus pais, José Carlos e Suely, por tudo que me ensinaram, por acreditarem em mim e por todo suporte e incentivo para a minha formação pessoal e profissional. À minha irmã, Karoline, pela sinceridade e amizade. Ao meu namorado, Heitor, pela cumplicidade, carinho e companhia.

Agradeço ao meu orientador, Dr. Eduardo Almeida, por investir em mim e guiar o meu trabalho. Agradeço também ao meu co-orientador, Dr. Raphael Oliveira, pela disponibilidade e por compartilhar seus conhecimentos e experiências de vida.

Agradeço a todos os docentes do Departamento de Ciência da Computação com quem tive a oportunidade de aprender durante os anos do curso de Ciência da Computação, especialmente à Dra. Aline Andrade, pelos seus conselhos e orientação. Agradeço ao Dr. Luciano Oliveira, pela excelência do seu trabalho como professor e coordenador do curso. Agradeço também ao Dr. John McGregor por me receber no seu grupo de pesquisa e orientar durante o meu estágio de verão na Universidade de Clemson (USA), onde fiz minha graduação sanduíche.

Agradeço aos meu amigos que influenciaram positivamente e participaram direta ou indiretamente da minha formação: Nanci Bomfim, Melissa Wen, Laiza Camurugy, Gabriel Uaquim, Rodrigo Vieira, Félix Farias, Caio Almeida e Diego Arize. Agradeço também à minha amiga Marie Cudd por me adotar em Clemson e marcar minha vida de uma forma extraordinária.

Agradeço à Universidade Federal da Bahia por me proporcionar esta formação, à Fapesb, pela bolsa de Iniciação a Extensão, à Capes pela bolsa do programa Ciência sem Fronteiras, e às instituições onde estagiei, EcGlobalPanel, IFBA, TecnoTrends e Ericsson Inovação.

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

—PROVERBS 3:13

Resumo

A engenharia de software é uma disciplina que integra processos, métodos e ferramentas para o desenvolvimento de programas de computador. Para apoiá-la existem sistemas de gerenciamento de ciclo de vida de aplicativos (ALM), que são ferramentas de software que auxiliam na organização e moderação de um projeto ao longo de seu ciclo de vida. Existe um número infinito de possibilidades para definir o ciclo de vida de um projeto, e as ferramentas necessárias para ajudá-la a conclusão. Consequentemente, uma enorme variedade de sistemas de gestão estão disponíveis no mercado, especificamente concebidos de acordo com um número diversificado de metodologias de gestão. No entanto, a maioria dos sistemas são proprietários, bastante especializados e com pouca ou nenhuma capacidade de personalização, tornando muito difícil encontrar um que se encaixam perfeitamente às necessidades do seu projeto.

Um tipo diferente de desenvolvimento de sistemas de software é a Engenharia de Linha de Produtos de Software - ELPS. 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 vasta gama de artefatos devem ser criados e mantidos para preservar a consistência do modelo de família durante o desenvolvimento, o e que é importante para controlar a variabilidade SPL e a rastreabilidade entre os artefatos. No entanto, esta é uma tarefa difícil, devido à heterogeneidade dos artefatos desenvolvidos durante engenharia de linha de produto. Manter a rastreabilidade e artefatos atualizados manualmente é um processo passível de erro, demorado e complexo. Utilizar uma ferramenta para apoiar essas atividades é essencial.

Neste trabalho, propomos o Ambiente Construção Integrado de Linha de Produto de Software (SPLICE). Essa é uma ferramenta online de gerenciamento de ciclo de vida que gerencia, de forma automatizada, as atividades da linha de produtos de software. Esta iniciativa pretende apoiar a maior parte das atividades do processo de LPS, como escopo, requisitos, arquitetura, testes, controle de versão, evolução, gestão e práticas ágeis.

Nós apresentamos um metamodelo leve, que integra o processo de ciclo de vida das Linhas de Produtos com práticas ágeis, a implementação de uma ferramenta que utiliza o metamodelo proposto , e um estudo de caso que reflete a viabilidade e flexibilidade desta solução especialmente para diferentes cenários e processos.

Palavras-chave: ferramenta, busca linha de produtos de software, métodos ágeis, LPS , sistema de gerenciamento de ciclo de vida de aplicativos , ferramenta, metamodelo

Abstract

Software engineering is a discipline that integrates process, methods, and tools for the development of computer software. To support it, Application lifecycle management systems are software tools that assist in the organization and moderation of a project throughout its life cycle. There is an infinite number of possibilities to define a project life cycle, and the needed tools to help it completion. Consequently, a huge variety of management systems are available on today's market, specifically designed to conform to a diverse number of management methodologies. Nevertheless, most are proprietary, very specialized, with little to no customization, making very hard to find one that perfectly fit one's needs.

A different type of software systems development is Software Product Line Engineering – SPLE. 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 needs to be created and maintained to preserve the consistency of the family model during development, and it is important to manage the SPL variability and the traceability among those artifacts. However, this is a hard task, due to the heterogeneity of assets developed during product line engineering. Maintaining the traceability and artifacts updated manually is error-prone, time consuming and complex. Utilizing a tool for supporting those activities is essential.

In this work, we propose the Software Product Line Integrated Construction Environment (SPLICE). That is a web-based life cycle management tool for managing, in an automated way, the software product line activities. This initiative intends to support most of the SPL process activities such as scoping, requirements, architecture, testing, version control, evolution, management and agile practices. This was archived with the integration of a framework around an established tool, providing an easy way for handling the usage of different metamodels.

We present a lightweight metamodel which integrates the processes of the SPL lifecycle agile practices, the implementation of a tool that uses the proposed metamodel, and a case study that reflect the feasibility and flexibility of this solution especially for different scenarios and processes

Keywords: software product line, agile, SPL, Application lifecycle management , tool, metamodel

Contents

List of Figures

List of Tables

List of Acronyms

ALM	Application Lifecycle Management
AP	Agile Planning
BTT	Bug Report Tracker Tool
C.E.S.A.R.	Recife Center For Advanced Studies and Systems
CAD	Core Asset Development
CASE	Computer-Aided Software Engineering
CRUD	Create, read, update and delete
RE	Requirements Engineering
RiSE	Reuse in Software Engineering
RQ	Requirements
ORM	Object-relational mapping
PD	Product Development
SPL	Software Product Line
SPLE	Software Product Line Engineering
SE	Software Engineering
SW	Software
SC	Scoping
SPLICE	Software Product Line Integrated Construction Environment
SWIG	Simplified Wrapper and Interface Generator
TE	Tests
OT	Others
VCS	Version control systems

1 Introduction

Drop by drop is the water pot filled. Likewise, the wise man, gathering it little by little, fills himself with good.

—BUDDHA (Dhammapada)

Nowadays, there are a rising demand for individualized products. Those changes affects directly software-intensive companies that are faced with pressure to innovate and develop adaptations to their products, often big and complex. This has lead to a situation where many companies are constantly struggling with an increasing cost of developing new products due to increased size and complexity. On the other hand, the number of products and customer-specific adaptations required increases constantly (Capilla *et al.*, 2013).

Successful introduction of a software product line provides a significant opportunity for a company to improve its competitive position, and there are many reports documenting the significant achievements and experience gained by introducing Software Product Lines in the software industry (Pohl *et al.*, 2005). However, managing a Software Product Line (SPL) is not so simple, since it demands planning and reuse, adequate management and development techniques, and also the ability to deal with organizational issues and architectural complexity (Cavalcanti *et al.*, 2012).

According to Capilla *et al.* (2013) , the lack of systematic software variability management was the root cause or most of failures in Software Product Line (SPL) adoption. Recently, the concept of Application Lifecycle Management (ALM) has emerged to indicate the coordination of activities and the management of artifacts (e.g., requirements, source code, test cases) during the software product's lifecycle (Schwaber, 2006).

The focus of this dissertation is to provide a support tool for managing the Software

Product Line (SPL) life-cycle by maintaining the variability and traceability among artifacts, being extensible and easily modifiable for changes in the used metamodel. We also developed and implemented a metamodel using the tool.

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 and, finally, Section 1.6 outlines the structure of this dissertation.

1.1 Motivation

Software Product Line (SPL) is considered one of the most popular technical paradigm and emerging methodology in developing software products and is proven to be a successful approach in many business environments (Pohl *et al.*, 2005).

Software product lines are often developed and maintained using model-based approaches (Dhungana *et al.*, 2011). Modeling is used as a support mechanism to define and represent the variability involved in a SPL in a controlled and traceable way, as well as the mappings among the artifacts and elements that compose a SPL. (Cavalcanti *et al.*, 2012).

During the development of a SPL, a wide range of artifacts needs to be created and maintained to preserve the consistency of the family model during development, and it is important to manage the SPL variability and the traceability among those artifacts. However, this is a hard task, due to the heterogeneity of assets developed during product line engineering. To maintain the traceability and artifacts updated manually is error-prone, time consuming and complex. Therefore, using a Project management system for supporting those activities is essential.

A huge number of Computer-Aided Software Engineering (CASE) tools exists in the market for assisting Software Engineering activities and there are also specific tools for Software Product Lines engineering. However, during the development of a customer-oriented mobile application, the actual SPL development process and support tools where complex and too formal, imposing a strict and heavy process.

Application Lifecycle Management (ALM) aims to provide integrated tools and practices that support project cooperation and communication through a project's lifecycle. ALM are mainly provided by commercial vendors (Schwaber, 2006) and have a gap with the SPL practices regarding the lack of evidence in the literature.

During a development lifecycle, was identified that for mostly all projects, a set of

CASE tools were frequently used such as:

- Project management(Trac, DotProject, Redmine)
- Versioning control (Git, SVN, CVS)
- Issue Tracking (Bugzilla, Mantis)

By using separated tools, we lost an opportunity to automatize trace links, providing traceability among the artifacts. Software engineers also frequently have to provide the installation, maintainability and user management for a number of tools themselves, or rely on a external person for those tasks not directly related to the product development.

1.2 Problem Statement

This work investigates the problem of traceability and variability management during the Software Product Line (SPL) lifecycle characterizing it empirically to understand its causes and consequences, and provides a tool for SPL lifecycle management tool to support and reduce the effort spent in the traceability maintenance and assets management

1.3 Overview of the Proposed Solution

In order to accomplish the goal of this dissertation, we propose the Software Product Line Integrated Construction Environment (SPLICE). This tool supports the Software Product Line (SPL) process activities in order to assist engineers in the traceability, variability management and maintenance activity. The remainder of this section presents the context where it was developed and the outline of the proposed solution.

1.3.1 Context

This dissertation describes a tool that 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. RiSE Labs it is influenced by a series of areas, such as software measurement, architecture, quality, environments and tools, and so on, in order to achieve its goal. The influence areas can be seen in Figure 1.1.

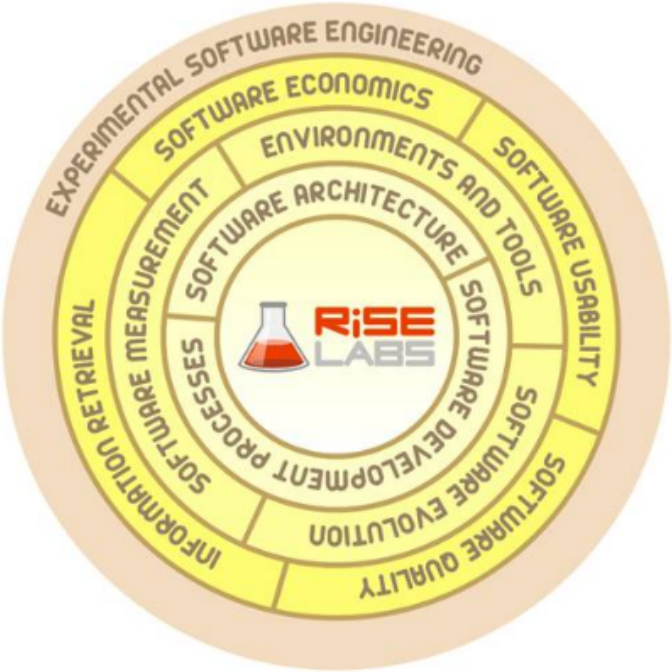


Figure 1.1 RiSE Labs Influences

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

- **RiSE Framework:** Involves reuse processes (Almeida *et al.*, 2004; Nascimento, 2008), component certification (Alvaro *et al.*, 2006) and reuse adoption process (Garcia *et al.*, 2008).

- **RiSE Tools:** Research focused on software reuse tools, such as the Admire Environment (Mascena, 2006), the Basic Asset Retrieval Tool (B.A.R.T) (Santos *et al.*, 2006), which was enhanced with folksonomy mechanisms (Vanderlei *et al.*, 2007), semantic layer (Durao, 2008), facets (Mendes, 2008) and data mining (Martins *et al.*, 2008), and the Legacy InFormation retrieval Tool (LIFT) (Brito, 2007), the Reuse Repository System (CORE) (Melo, 2008), and the Tool for Domain Analysis (ToolDAy) (Lisboa, 2008). This dissertation is part of the RiSE tools;

- **RiPLE:** Stands for RiSE Product Line Engineering Process and aims at developing a methodology for Software Product Lines, composed of scoping (Moraes, 2010), requirements engineering (Neiva, 2009), design (de Souza Filho, 2010;

Cavalcanti *et al.*, 2011a) , implementation, test (da Mota Silveira Neto, 2010; do Carmo Machado, 2010), and evolution management (de Oliveira, 2009).

- **SOPLE:** Development of a methodology for Service-Oriented Product Lines, based on the fundamentals of the RiPLE (Ribeiro, 2010).
- **MATRIX:** Investigates the area of measurement in reuse and its impact on quality and productivity;
- **BTT:** Research focused on tools for detection of duplicate bug reports, such as in Cavalcanti *et al.* (2008, 2012).
- **Exploratory Research:** Investigates new research directions in software engineering and its impact on reuse;
- **CX-Ray:** Focused on understanding the Recife Center For Advanced Studies and Systems (C.E.S.A.R.), and its processes and practices in software development.

This dissertation is part of the RiSE Tools project. It was conducted in collaboration with researchers in software reuse , to solve the problem of traceability during the life-cycle of aSoftware Product Line (SPL) development.

Figure 1.2 RiSE Labs Projects

1.3.2 Outline of the Proposal

This work defines the requirements, design and implementation of a software product lines 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. In order to address it, we propose a metamodel that covers the SPL lifecycle, and develop a solution that consists in a Web based, extensible SPL lifecycle management tool, implementing this metamodel.

The tool must enable the engineers involved in the process, to automatize the assets creation and maintenance, while providing traceability and variability management between them, providing detailed reports and enable the engineers to easily navigate between the assets using the traceability links. It must also provide a basic infrastructure for development, and a centralized point for user management among different tools.

1.4 Out of Scope

- **Full Application Engineering Support** Application engineering is the process of software product line engineering in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability (Pohl *et al.*, 2005). Although the SPLICE architecture is flexible enough for it, we still do not support code binding, and cannot perform automatic product derivation.

- **Risk Management.** Our metamodel do not support risk management assets yet.

- **Type of users.** The subjects of this work are developers and testers who will develop the system or test it. It also applies for stakeholders with some software engineering background, who understand the impact of their changes. Thus, it is out of scope to provide a tool that supports the following types of users: business experts and end-users.

1.5 Statement of the Contributions

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

- **A study on SPL lifecycle management tools**, which can provide the research and professional community an overview of the state-of-the-art tools in the field considering the literature and tools on the market.

- **A metamodel for Software Product Lines tools** that cover the activities of a SPL development with agile practices and represent the interactions among the assets of a SPL, as a way to provide traceability and variability, based on the metamodel developed in the RiSE Labs (Cavalcanti *et al.*, 2011b)

- **An application lifecycle management system** to support the suggested SPL metamodel. A web-based, collaborative tool which acts as a centralized location to user-management, provide an infrastructure and support for the SPL lifecycle steps.

1.6 Dissertation Structure

The remainder of this dissertation is organized as follows:

- **Chapter 2** reviews the essential topics used throughout this work: Software Product Lines and CASE tools. It also contains a comprehensive revision similar tools.

- **Chapter 3** describes the functional and non-functional requirements proposed for the SPLICE tool as well as architecture, the set of frameworks,the SPL metamodel and technologies used during the SPLICE implementation.

- **Chapter 4** describes an academic case study conducted to evaluate the tool.

- **Chapter 5** provides the concluding remarks. It discusses our contributions, limitations, threats to validity, and outline directions for future work.

2

An Overview on Software Product Lines, Requirements Engineering, SPL Requirements Engineering and SPL Tools

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 SPL Tools. 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.

- **Quality improvement**

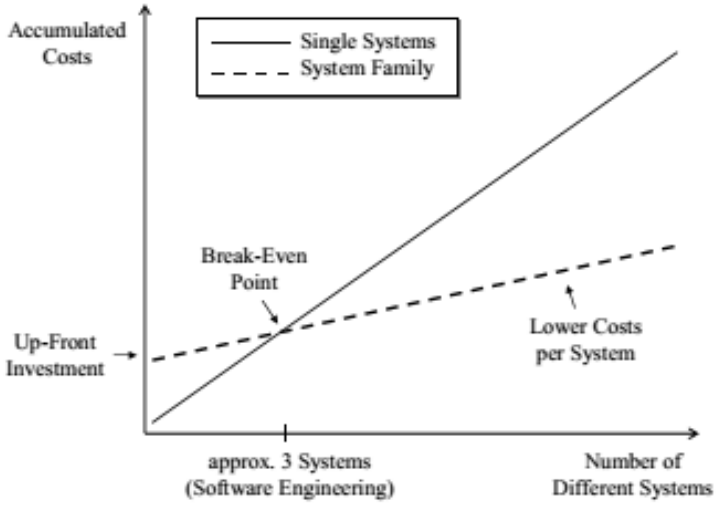


Figure 2.1 Costs for developing systems as single systems compared to product line engineering (Pohl *et al.*, 2005)

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.

- **Benefits for the Customers**

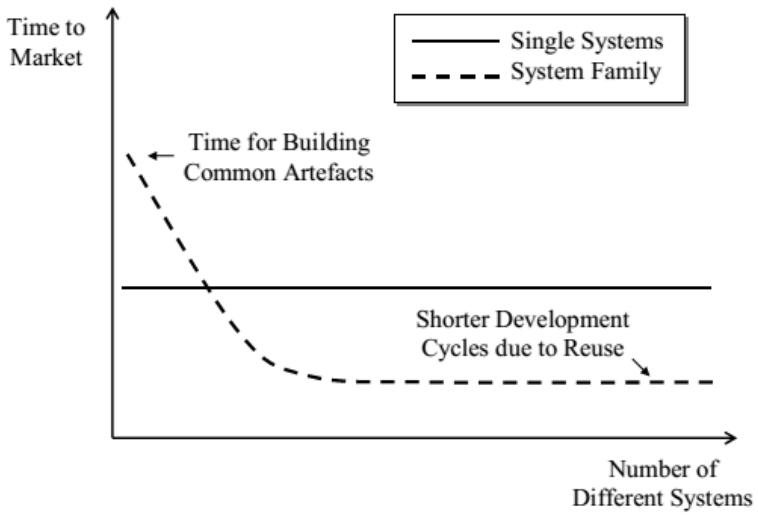


Figure 2.2 Comparison of time to market with and without product line engineering (Pohl *et al.*, 2005)

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).

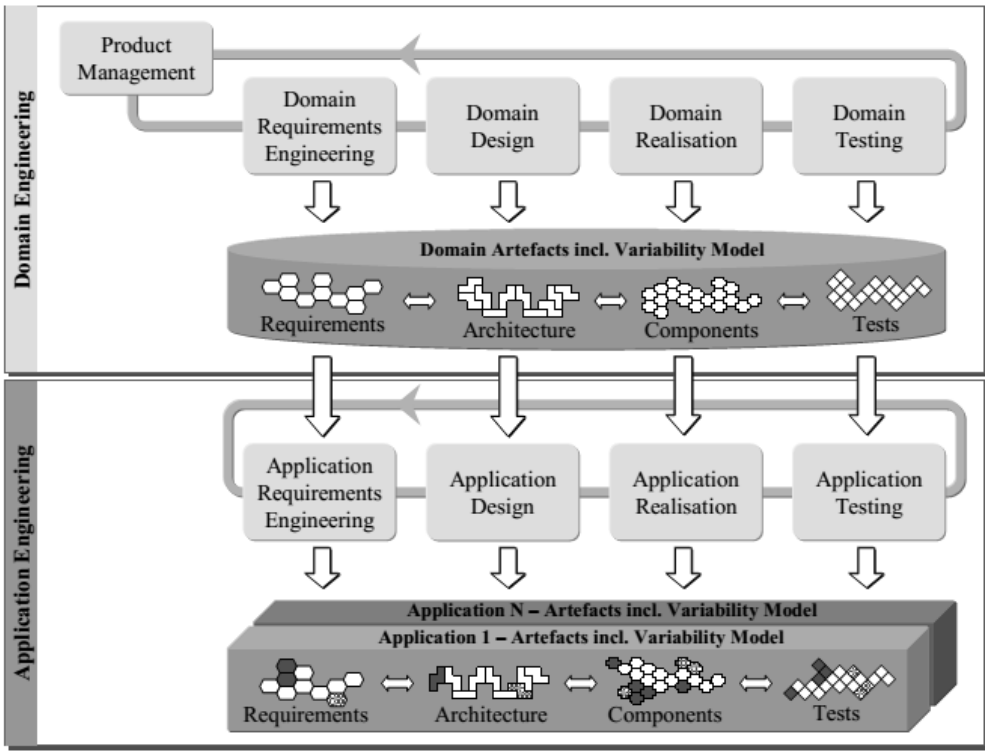


Figure 2.3 The software product line engineering framework (Pohl *et al.*, 2005)

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).

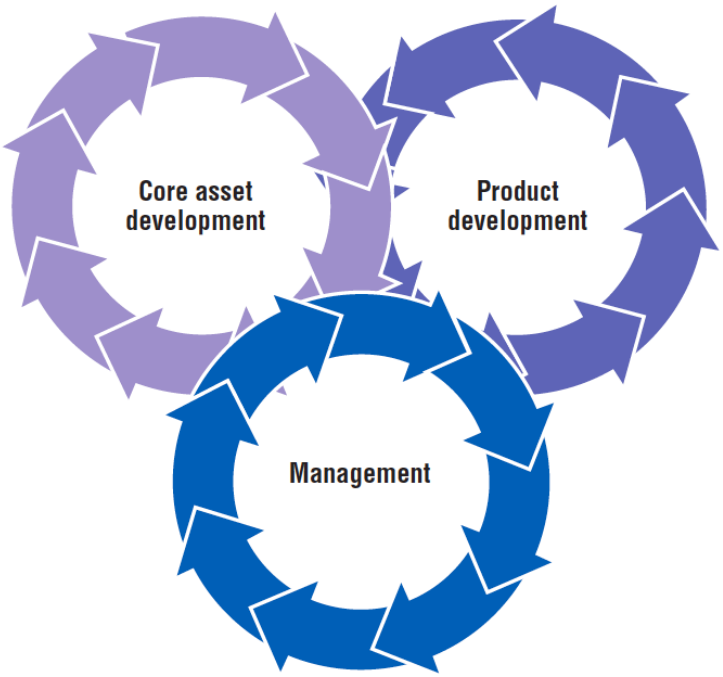


Figure 2.4 SPL Activities (Clements and Northrop, 2002)

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.

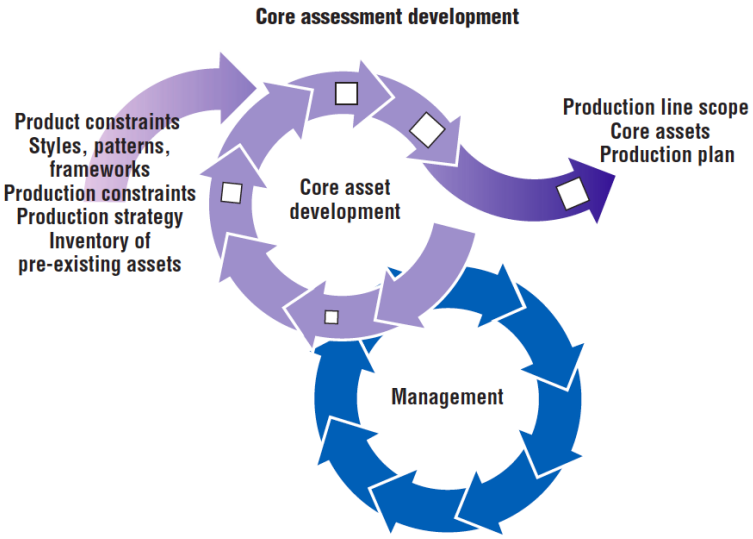


Figure 2.5 Core Asset Development (Clements and Northrop, 2002)

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

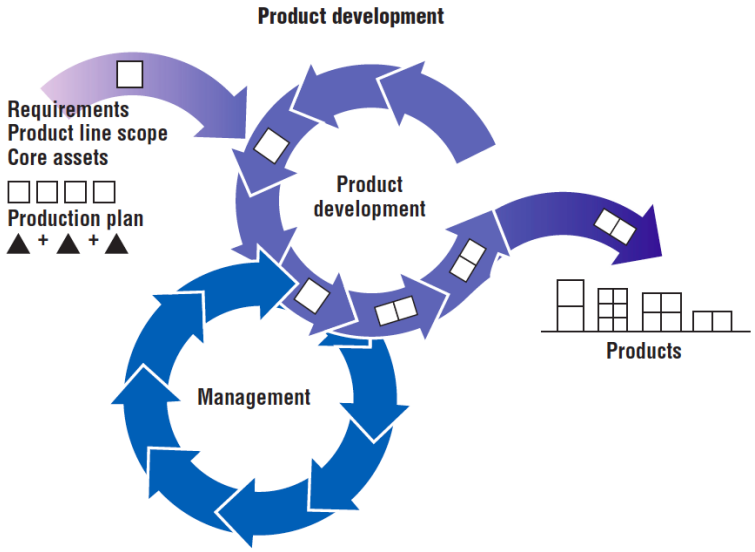


Figure 2.6 Product Development (Clements and Northrop, 2002)

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 be 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, Requirement 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; 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** 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.

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., 1999a; Moon et al., 2005; Eriksson et al., 2005; Bonifácio and Borba, 2009; Alférez et al., 2011; Mussbacher et al.,2012; Shaker et al., 2012). An 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.

However, the approaches for specifying SPL functional requirements do not propose guide- lines, showing step by step how the specification should be done. This lack of guidelines may led to some challenges and risks.

2.4 SPL Tools

2.5 Summary

In this chapter, we discussed about important concepts to this work: the area of Software Product Line (SPL), Computer-Aided Software Engineering (CASE) tools and Application Lifecycle Management (ALM) tools, including motivations, benefits and definitions. We also made a analysis of features and comparison of tools available in market.

Next chapter presents the Software Product Line Integrated Construction Environment

(SPLICE), a web-based, collaborative support for the SPL lifecycle steps. It will be discussed the requirements, architecture, implementation and other aspects.

3

SPLICE - Software Product Line Integrated Construction Environment

You alone are the author of your future. At every moment, you have the opportunity to change — to alter your thoughts, your speech, your actions.

—HENEPOLA GUNARATANA (Eight Mindful Steps to Happiness)

3.1 Introduction

The development of software products is a highly complex endeavor, which includes numerous interdependent activities, various different roles and a wide range of artifacts spanning over all phases and activities (Lacheiner and Ramler, 2011).

In this chapter, we present functional and non-functional requirements for a tool, it architecture, involved frameworks and technologies, and its implementation. The tool is called 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.

We also present a metamodel, which proposes a representation of the SPL artifacts, based on agile methods and variability management amount the assets (Features, Requirements, Use Case, etc...). This metamodel is implemented by SPLICE.

The remainder of this chapter is organized as follows: Section 3.2 presents the requirements of the tool; Section 3.3 describes the created SPL metamodel; Section 3.4

shows its general architecture; details of the implementation are discussed in Section 3.5; Section 3.6 shows the tool in operation; and, finally, Section 3.7 presents the summary of the chapter.

3.2 Tool Requirements

3.2.1 Functional Requirements

According to Sommerville (2007) , Functional Requirements are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. In the SPLICE specification, the following functional requirements were defined:

- **FR1 - Traceability of lifecycle artifacts.** The tool should identify and maintain relationships between managed lifecycle artifacts. It is important not just for reporting but also for enabling change impact analysis and information visibility through the development lifecycle.

- **FR2 - Reporting of lifecycle artifacts.** It must use lifecycle artifacts and traceability information to generate needed reports from the lifecycle product information. Thus, supporting the Software (SW) development and management and easily producing ready to consume documentation for the stakeholders.

- **FR3 - Metamodel Implementation.** The SPLICE should implement entities and relationships described in a defined metamodel. The metamodel created comprises the relationships among the SPL assets, allowing traceability and facilitating the evolution and maintenance of the SPL. This requirement is very demanding and complex, and include a big number of features to correctly represent the metamodel presented, such as *Product Map*, *Features Selection* and *Evolution management*.

- **FR4 - Issue Tracking.** Issue Tracking play a central role in software development, and the tool must support it. It is used by developers to support collaborative bug fixing and the implementation of new features. In addition, it is also used by other stakeholders such as managers, QA experts, and end-users for tasks such as project management, communication and discussion, code reviews, and story tracking (Baysal *et al.*, 2013).

- **FR5 - Agile Planning.** In the software industry, there is a strong shift from traditional phase-based development towards agile methods and practices. Some of agile characteristics includes: customer collaboration, small self-organizing teams, emphasis on coding, minimal bureaucracy and documentation effort, test-driven development, and incremental delivery (Hochmüller, 2011).

- **FR6 - Configuration management.** For evolution management, the tool must support change management across all the managed artifacts. It must also support creating and controlling the mainstream version management systems such as CVS, SVN and GIT.

- **FR7 - Unified User management.** As an integrated environment, with the plan to cover all the lifecycle activities, the tool can use a number of external tools, taking advantage of the vibrant community and quality present in some open-source/freely available tools. For convenience, it must provide a unified user management between all the tools.

- **FR8 - Collaborative documentation.** Wiki is a collaborative authoring system for collective intelligence which is quickly gaining popularity in content publication and it is a good candidate for documentation creation. Wikis collaborative features and markup language make it very appropriate for documentation of Software Development tasks.

- **FR9 - Artifacts search.** All artifacts managed by the tool must support keyword search, and present the results in an adequate way.

3.2.2 Non-Functional Requirements

Sommerville (2011) define Non-Functional Requirements as requirements that are not directly concerned with the specific services delivered by the system to its users. They place restrictions on the product and the development process, and specify constraints that the product must meet. Non-functional requirements of the tool is described as follows:

- **NFR1 - Easy access.** The tool must be easily accessible and a web-based tool enables a high level of interoperability between different systems. It allows engineers and stakeholders to collaborate and access the system independently of their location.

- **NFR2 - Metamodel Flexibility.** There is no perfect process, and metamodels changes and evolves quickly. The tool must be implemented in a way to easily allow metamodel changes.

- **NFR3 - Extensibility.** The SPLICE must have a plugin infrastructure for allowing external developers to interact with it using an API. It is planned that in the future, the SPLICE could become a platform which enable undergraduate students to build and performs Software Engineer experiments on it.

- **NFR4 - Usability.** As an integrated environment, it will aggregate a plethora of different tools. It is important to have a consistent User Experience between them. The tool must also conform with the latest trends in web-design, and must provide a modern and minimalist interface.

- **NFR5 - Accountability.** All users actions must be logged, for evolution management, security and accountability.

- **NFR6 - Transparency.** The user must identify the collection of tools as a single tool. The tool must permit that all the integrated tools be accessible with just one login.

- **NFR7 - Security.** Being web-based and publicly accessible, it is necessary to keep a tight access control in the application in order to guarantee the privacy and confidentiality of the projects.

3.3 A lightweight SPL metamodel

A Software Product Line (SPL) process has to handle interconnected and complex models such as feature and architecture models. Furthermore, traceability is fundamental to ensure that they are consistent. An efficient process require mature software engineering, planning and reuse, adequate practices of management and development, and also the ability to deal with organizational issues and architectural complexity. (Birk and Heller, 2007).

To address the previously defined function and non-functional requirements, we decided to use a model-driven approach to represent all the information, activities and connections among artifacts. With a well-defined metamodel we can provide automation and interactive tool support to aid the corresponding activities.

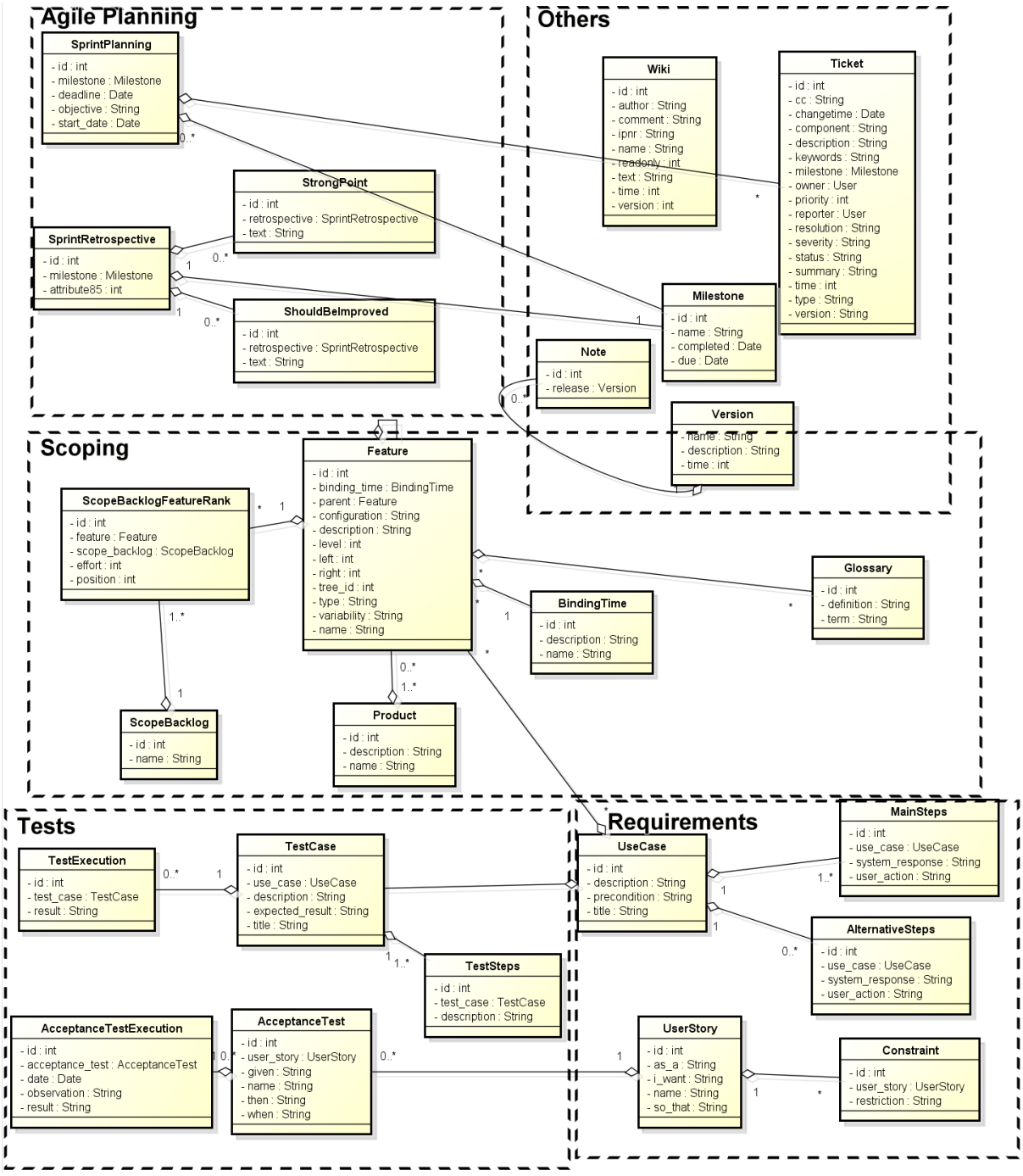


Figure 3.1 SPLICE metamodel

A huge number of metamodels have been proposed in the literature in the past (Schubanz *et al.*, 2013; Araújo *et al.*, 2013; Anquetil *et al.*, 2010; Bayer and Widen, 2002; Buhne *et al.*, 2005; Seidl *et al.*, 2012), including one done by a RiSE researcher (Cavalcanti *et al.*, 2011b). However, we found that all the metamodels propose a heavyweight and formal process, and do not fit with a more lightweight and informal methodology such as agile methods.

In the Figure 3.1 we propose a lightweight metamodel, adapted from Cavalcanti *et al.* (2011b), representing the interactions among the assets of a SPL, developed in order to provide a way of managing traceability and variability, however it explicitly removes, for clarity, the representation of user-management, authentication, Version control, Issue Tracking, Logging and internal models.A more complete diagram is depicted on Figure 3.2. The proposed metamodel represent diverse reusable assets involved in a SPL project, and there are five sub-models:

- **Scoping Module** In this module, is located the core of the metamodel, 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*.

  The *feature* model is represented as a Modified Preorder Tree Traversal, which allow representing hierarchies. This is used to model relationships between features. It also permit *required* and *excluded* features relationships. *Feature* also have a *Glossary*, containing the definition of terms used. Additionally it have *BindingTime* and *VariabilityType* values associated with it. Kang *et al.* (1990) describes examples of *binding time*, such as *before compilation*, *compile time*, *link time*, *load time*, *run-time*; and examples of variability as *Mandatory*, *Alternative*, *Optional*, and *Or*.

  The Scoping module also have a *ScopeBacklog* model, that permit to classify the features by order of importance.

- **Requirements Module** The metamodel also involves the requirement engineering traceability and interactions issues, considering the variability and commonality in the SPL products (Cavalcanti *et al.*, 2012). The main object of this SPL phase is *Use Case*. Differently from Cavalcanti *et al.* (2012) metamodel, we following an agile development process, which has been designed to cope with requirements that changes during the development process. In these processes, when a user proposes a requirements change, this change does not go through a formal change management process.

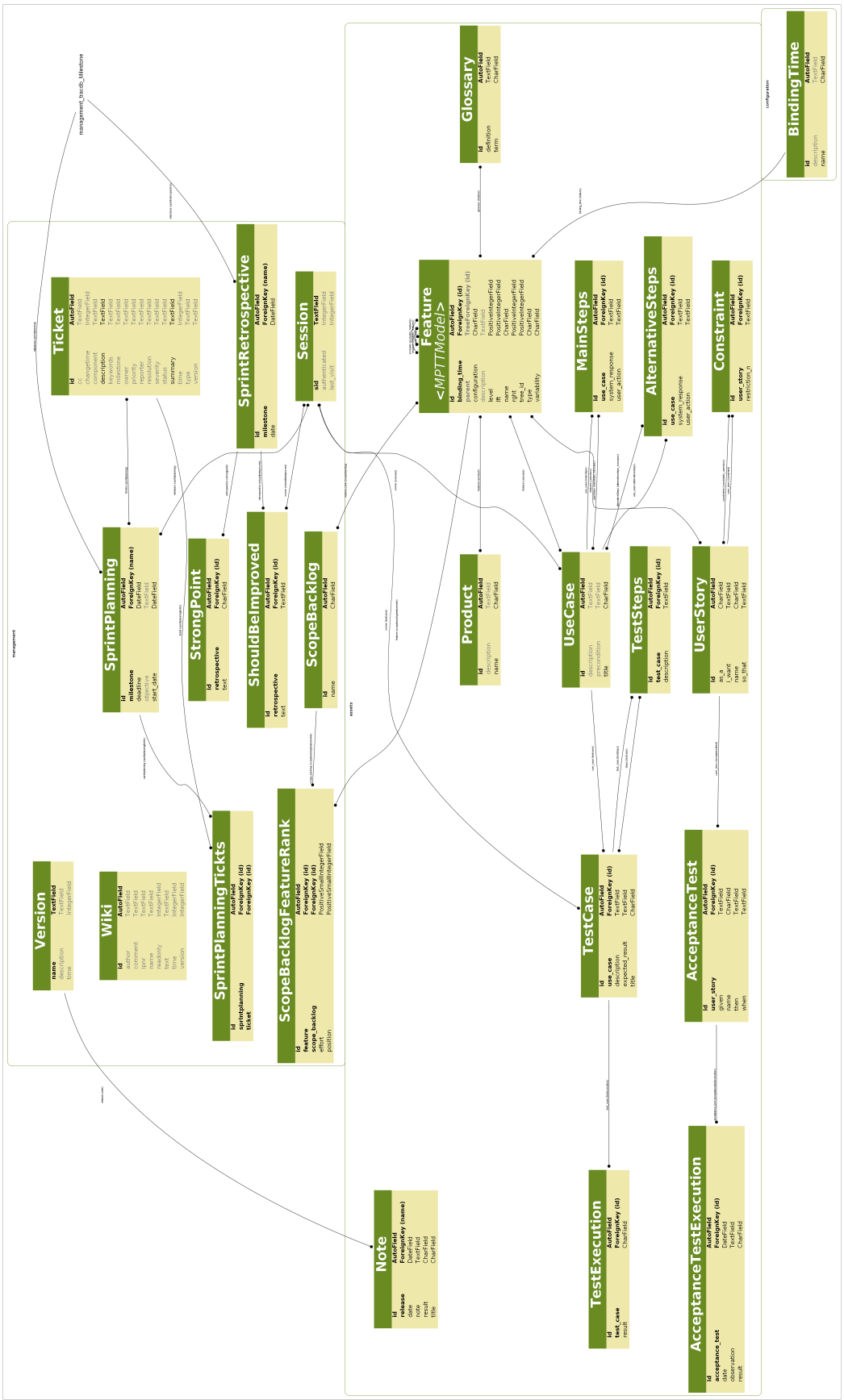


Figure 3.2 Django generated meta-model

The *Use Case* model is composed by *description*, *precondition*, *title* and a number of *MainSteps* and *AlternativeSteps* 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. It is composed by a *name* and the associated template : *As a*, *I want* and *So that*.

- **Testing Module** An agile project is centered on short iterations in which each iteration can be viewed as a small project of its own. Therefore, one characteristic of an agile project is the high degree of rework that mandates a comprehensive set of regression tests (Goeschl *et al.*, 2010).

  According to Shaye (2008), test automation plays the greatest role in agile testing, with it is possible to keep testing and development in synchronization. In the metamodel proposed we represent the usual test workflow in agile environments. The *Test Case* model is related to one *Use Case* and 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 represent the acceptance testing with the *Acceptance Test* and *Acceptance Test Execution*. An acceptance test is a formal description of the behavior of a software product, usually expressed as an example or a usage scenario. In principle, the customer or his representative is given the role of expressing requirements as input to the software paired with some expected result (Shaye, 2008). The *Acceptance Test* model is based on the *Given-When-Then* template. (Given) some context, (When) some action is carried out, (Then) a particular set of observable consequences should obtain. It also have a number of *Acceptance Test Execution* associated with it. The *Acceptance Test Execution* is composed of the date it was executed, test result and the observations.

- **Agile Planning Module** There are several product development approaches, such as agile, interactive, incremental, and phased approaches. This metamodel is more closely related to an agile methodology, which is an incremental and iterative style of development, with focus on customer collaboration. In agile software development, the project is developed in a series of relatively small tasks, visualized, planned and executed by producing a shippable incremental product for the customer (Uikey and Suman, 2012).

  The agile interactions are called *sprints*, and there are an upfront planning of it,

represented in the metamodel by the *Sprint Planning* model. This planning is 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.

- **Others Module** This module is composed of all the models that did not fit in the other sections. It includes *Wiki* model for a collaborative document creation; *Ticket* model for Issue tracking; and *Milestone*, *Version*, *Note* for release management.

3.4 SPLICE Architecture and Technologies

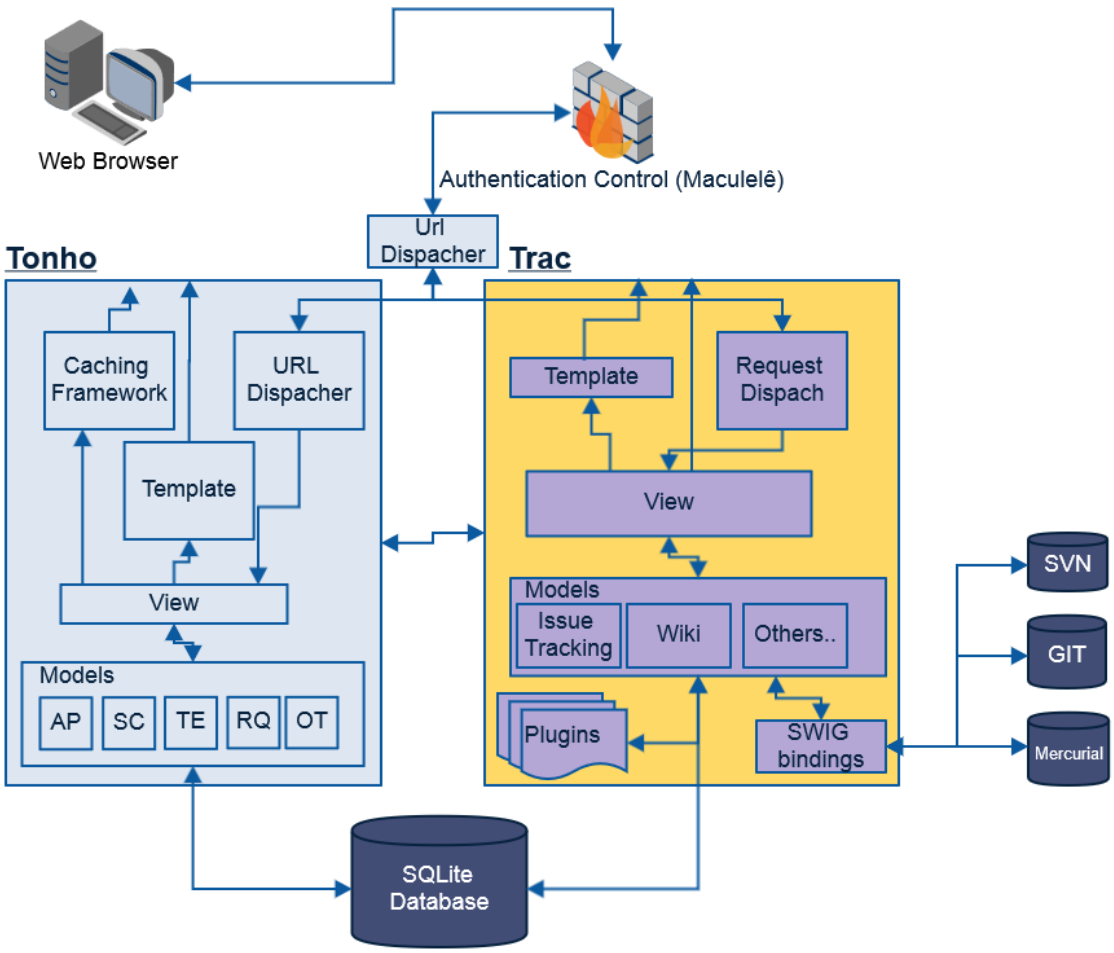


Figure 3.3 SPLICE architecture overview

3.4.1 Architecture Overview

The SPLICE architecture was planned to cover all the essential steps of the Software Product Line (SPL) process, coping with the defined functional and non-functional requirements and acting as a Application Lifecycle Management (ALM) tool. During the evaluation of existing tools in Section **??**, we found in that the tool *Trac* [1] was a perfect candidate to be the base of our product. *Trac* is an open source, Web-based project management and bug tracking system .It has been chosen for a number of reasons:

- **Stable and Established tool.** *Trac* initial release was on October 1, 2006, and the version 1.0, defined as stable, was released 7 years after. Its source has been deeply scrutinized for issues and a number of high-value users have public accessible instances of *trac*. Trac is reported to have more than 450 major installations worldwide [2].Some notable users includes NASA's Jet Propulsion Laboratory[3] , Nginx[4] and WordPress [5]. Thus, we have good indicators of stability and security.

- **Extensible with a thriving ecosystem.** *Trac* is extensible via plugins. Plugins can be used to add functionality that was not previously possible without extensive modification to the source. The website Trac-Hacks [6] indexes more than 600 plugins, that includes a wide range of functionality, such as: extended milestone facilities; multi-project functionality; integration of automated build systems; dependency graphs and Distributed Peer Review. This makes very likely that for many popular requests, may already exist a plugin that provides the wanted functionality.

- **Set of features.** After examination if other open-source solutions, we found that *Trac* had a good number of features needed to build our tool. *Trac* defines[7] itself as an "enhanced wiki and issue tracking system for software development projects". It provides: an interface to Subversion and Git (or other version control systems), an integrated Wiki and convenient reporting facilities. Trac also allows wiki markup in issue descriptions and commit messages, creating links and seamless references between bugs, tasks, changesets, files and wiki pages. Futhermore, *trac* have a

[1] http://trac.edgewall.org
[2] http://trac.edgewall.org/wiki/TracUsers
[3] http://www.jpl.nasa.gov/
[4] http://trac.nginx.org/nginx/
[5] https://core.trac.wordpress.org/
[6] http://trac-hacks.org/wiki/HackIndex
[7] http://trac.edgewall.org/

timeline showing all current and past project events in a temporal order, making the acquisition of an overview of the project and tracking progress very easy.

- **Liberal license.** *Trac* uses the Revised BSD License [8] , a very permissive free software license, imposing minimal restrictions on the redistribution of covered software. This enable us to modify and distribute the code without any restriction.

An overview of the architecture of SPLICE can be seen in Figure 3.3. Using the *Trac* tool as foundation to the SPLICE tool, two separated modules were built to provide the missing functionality. *Tonho*, is our main module, where all the metamodel and functionality not provided by *Trac* is implemented and the *Authentication manager* is the module who provides the *Single sign-on* property among the tools, supplying unified access control with a single login.

All the bridge between *Tonho* and *Trac* is made using plugins, a shared database and templates. Absolutely no modification was done to *Trac* core. This solution permits to easily upgrade *Trac* in the future taking advantage of new features and security fixes. All modules of the architecture share the same Database and have the same template design. Each component of the architecture is detailed in the following section.

3.4.2 Architecture Components

The architecture of the tool is composed of the *Trac*, a core (*Tonho*) module, a database module, an Authentication Control (*Maculelê*) module and a set of versioning and revision control tools.

Figure 3.3 illustrates a simplified organization of the architecture of the application: the ***Authentication Control*** validates the request and enable access to the tools if the user have the right credentials; ***Trac*** is responsible for Issue Tracking, managing the versioning and revision control tools, collaborative documentation and plugin extensibility; the main module **(*Tonho*)**,it is where the metamodel is implemented and has sub-modules for each metamodel module, such as: Scoping (SC), Requirements (RQ), Tests (TE), Agile Planning (AP) and Others (OT); ***Versioning and revision control*** is part of software configuration management, and is composed of the set of external Version control systems (VCS) tools such as *SVN*, *GIT* and *Mercurial*. It tracks and provides control over changes to source code, documents and other collections of information; and finally, the ***Database*** stores in an organized way all the asserts and do the persistence of the data among the tools. Next, it will be provided more details for each component.

[8] http://opensource.org/licenses/BSD-3-Clause

Trac. *Trac* is a minimalistic web-based software project management and bug/issue tracking system. It is the foundation and it development is completely external to the SPLICE tool development. It was the initial module important to the architecture and provides an interface to the revision control systems, an integrated wiki, flexible issue tracking and convenient report facilities. The communication with the rest of the modules is done by the shared database, and a set of plugins developed to make a bridge between the rest of the architecture. Internally *Trac* the following technologies to archive it goal:

- **Genshi.** Genshi is a Python library that provides an integrated set of components for parsing, generating, and processing HTML, XML or other textual content for output generation on the web. [9] It is used in *Trac* to provide templating and representing any information to the user.

- **SWIG bindings .** Many popular Version control systems (VCS) are set of C libraries, for example Subversion and Git. As *Trac* is developed using the Python scripting language, it cannot directly use those libraries. It need a tool to allow calling native functions. Simplified Wrapper and Interface Generator (SWIG) is an open source software tool used to connect computer programs or libraries written in C or C++ with scripting languages such as Python and is used in *Trac* to enable access to popular VCS such as *SVN*, *GIT* and *Mercurial*.

SPLICE Tonho. *Tonho* is the internal codename for the core module of SPLICE, where all the functionality missing from *Trac* and the proposed metamodel is implemented. The non-functional requirement NFR2, required Metamodel Flexibility, and this module was carefully designed to be flexible and easily allow future metamodel alterations.

The idea is at the begin of each project, the Software Engineer would describe the metamodel he wanted for that specific project. He may choose to use a predefined metamodel, such as the one presented here, or to customize it adding, removing or modifying assets.

An overview of the solution can be seen in Figure 3.4. The SPLICE will ship a public accessible set of classes that represent the metamodel, and the engineer can modify the metamodel, by changing the objects fields names, types and descriptions.

A Object-relational mapping (ORM) module will then convert the class model and create a database representing all the relations in a SQL format. Finally a tool will automatically read the metadata in the models and using the created database, will provide a powerful and production-ready "Create, read, update and delete (CRUD)"

[9] http://genshi.edgewall.org/

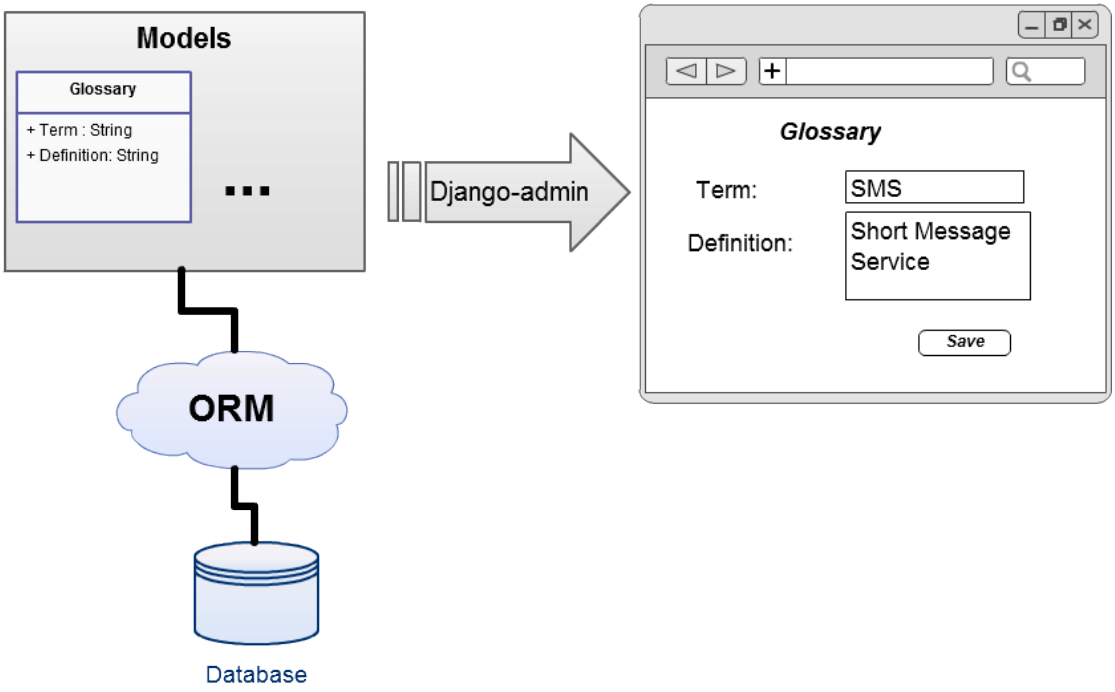


Figure 3.4 SPLICE model transformation

```
class Glossary(models.Model):
    term = models.CharField(max_length=200)
    definition = models.TextField(max_length=500)
```

Figure 3.5 SPLICE model class

interface. Figure 3.5 exemplify how this a class look like. This gives unprecedented level of flexibility, as the metamodel can be changed without any other effort then direct modifying the models. The set of technologies used to possibility this were:

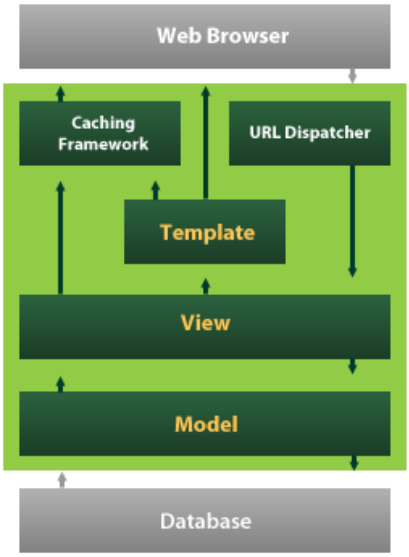


Figure 3.6 Django Architecture

- **Django.** For creating the web-application, we choose the Django Framework. It is a high-level Python Web framework that encourages rapid development and pragmatic design that follows the Model-View-Controller (MVC) pattern. Between a number of frameworks available for *Python* web-development, Django was chosen because it employs and enforces a model-view-controller (MVC) methodology, a common development pattern for separating presentation from application logic, encouraging good development practices and produces a more maintainable code. An overview of Django architecture can be seen in Figure 3.6. Django is also the most widely used Python web framework, with a very well established community of developers.

  The SPLICE also relies on Django Object-relational mapping (ORM), where the metamodel is mapped as entities and their relationship into Python classes and the ORM automatically create a relational database.

- **Bootstrap.** The interface and design were build using the Bootstrap Framework[10]. Bootstrap is a free collection of tools for creating websites and web applications. It contains HTML and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions[11].

SPLICE Database The database is a central point of the whole architecture. It is shared between two major modules, and permit global access to assets and informations. This characteristic is very important to provide traceability and enable creation of detailed and complete reports among a variety of assets, produced by different modules. The technology used to implement the database was:

- **SQLite.** To implement the database of the application, it was chosen the SQLite database. It is a powerful, embedded relational database management system in a compact C library. It offers support for a large subset of SQL92, multiple tables and indexes, transactions, views, triggers and a vast array of client interfaces and drivers. The library is self-contained, fast, efficient and scalable.

SPLICE Authentication Control (*Maculelê*) This module is responsible for coordinating the authentication and control access between the modules, satisfying the requirement *usability*. It have a Single sign-on property and the SPLICE tool provides a control panel that manages all users accounts.

The user-session is stored as a cookie sent to the user browser, it is composed of the user-metadata and a US Secure Hash Algorithm 1 (SHA1) hash of the user meta-data with a secret code stored on the server, protecting from forgery while giving persistence of credentials.

Versioning and revision control Version management is an important part of the Configuration management process. It is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used. It also involves ensuring that changes made by different developers to these assets do not interfere with each other. To support version management, you should always use Version control systems (VCS) tools (Sommerville, 2011). VCS are external modules from the architecture with the communication bridged by *SWIG* bindings. SPLICE provides support for two popular VCS systems: *SVN* and *GIT*.

[10] http://getbootstrap.com
[11] http://en.wikipedia.org/wiki/Bootstrap

Language	Files	Blank lines	Comment lines	Code lines
Javascript	105	6746	5064	27421
CSS	59	3021	646	20588
Python	147	3115	3400	13903
HTML	134	836	148	7154
XML	10	8	0	797
YAML	1	0	0	11
make	1	3	1	9
Total	457	13729	9259	69883

Table 3.1 Project statistics

3.5 Implementation

The proposed tool 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 have 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".

The motivation for choosing this stack was because the *trac* was already build with Python and the unprecedented flexibility that the Django Object-relational mapping (ORM) empowers it users, making the metamodel changes effortless. Python is also 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. Other languages used to developed the tool were JavaScript, CSS, HTML, XML, YAML and make.

The SPLICE tool was developed by only one person and it took about 8 months of intensive development (30 hours/week). According to cloc tool [12] , the SPLICE project consists of almost 70k lines of code, excluding comments and blank lines. Table 3.1 shows the code statistics of the project. All the SPLICE code is platform-independent, and has been thoroughly tested on both Windows and Linux environments.

[12] http://cloc.sourceforge.net

3.6 SPLICE in action

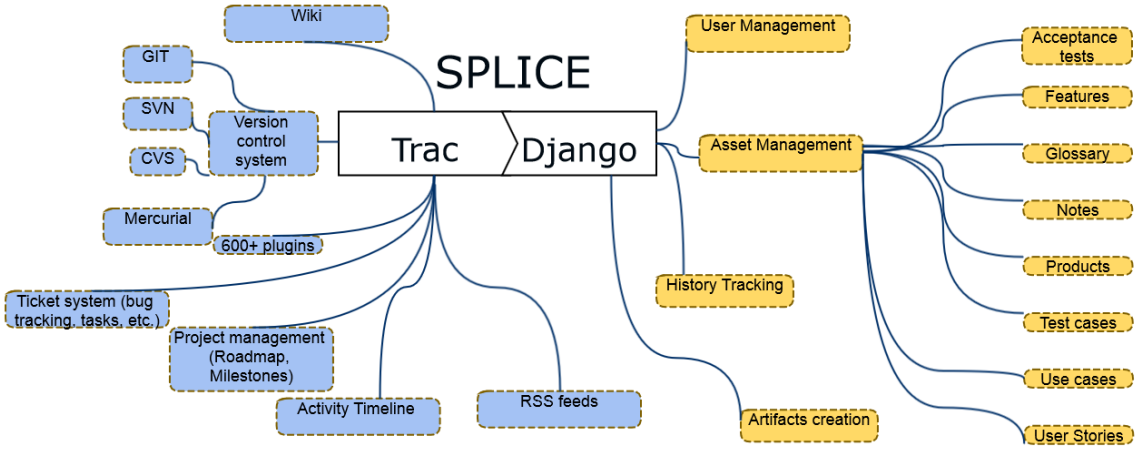


Figure 3.7 SPLICE features

The SPLICE is a complex tool, covering many steps and different process, and work as a façade for a number of external tools. In order to demonstrate how the tool works, this section shows the operation of a number of selected features, with a brief description. Figure 3.7 depicts some features offered by the SPLICE and what came from *trac*. The tool, following the non-functional requirement *NFR1 - Web-based interface*, is a web-based tool.

Main Screen

The main interface can be seen in Figure 3.8, this present the main structure of the SPLICE user interface. The main fields of are:

- 1. **Tool bar.** In this part of the tool are displayed the tools and screens accessible to the user, depending on his credentials. An unlogged user, will access a limited set of options, sticking to *NFR7 - Security* , providing protection to privileged information.

- 2. **Search bar.** It implements the functional requirement *FR9 - Artifacts search* This is the part of the tool in which the user can insert search terms and all the artifacts such as Issue Reports that contains the term will be presented to the user.

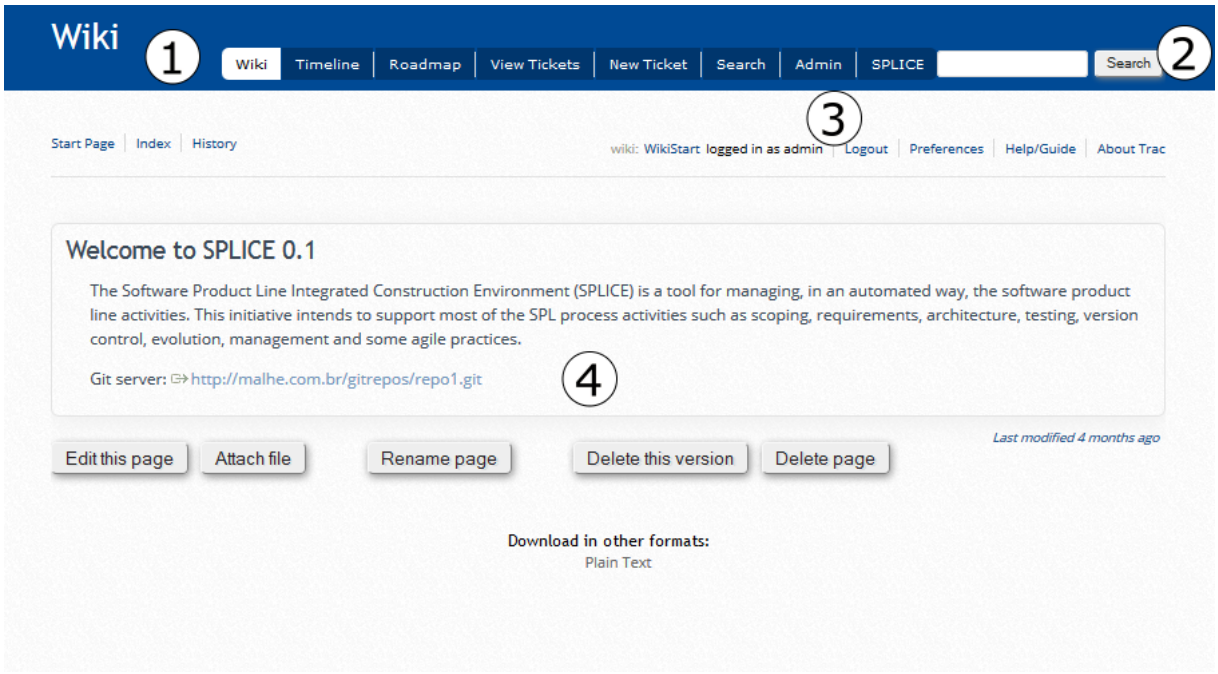


Figure 3.8 SPLICE main screen

- 3. **Breadcrumbs.** In this part of the tool are presented some information about the environment and some utilities. Through the toolbar, users can view information about the logged user and the actual path.

- 4. **Primary content.** This is the core view for content. In the SPLICE, as default, the first screen is a collaborative document (known as *Wiki page*), as shown in Figure 3.8. *Wikis* implements the functional requirement *FR8 - Collaborative documentation*.

Metamodel

The Figure 3.9 shows the Metamodel assets screen, and is where the requirement *FR3 - Metamodel Implementation* is represented. This screen is completely auto-generated based on the models descriptions. For every model, a complete "Create, read, update and delete (CRUD)" system is created, but idiosyncrasies can be easily customized. The SPLICE also provide advanced features such as filtering and classification. In Figure 3.10 is shown the list of indexed features, and in the side-panel is possible to filter features by *Type* and *Variability*.

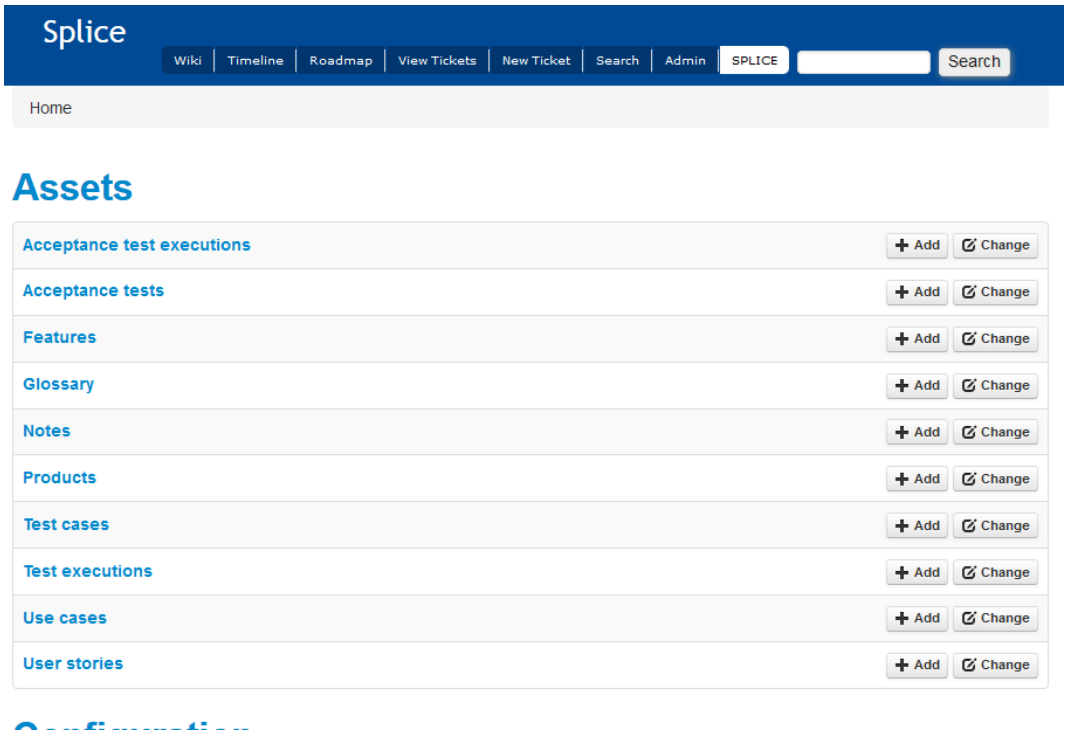


Figure 3.9 SPLICE metamodel assets

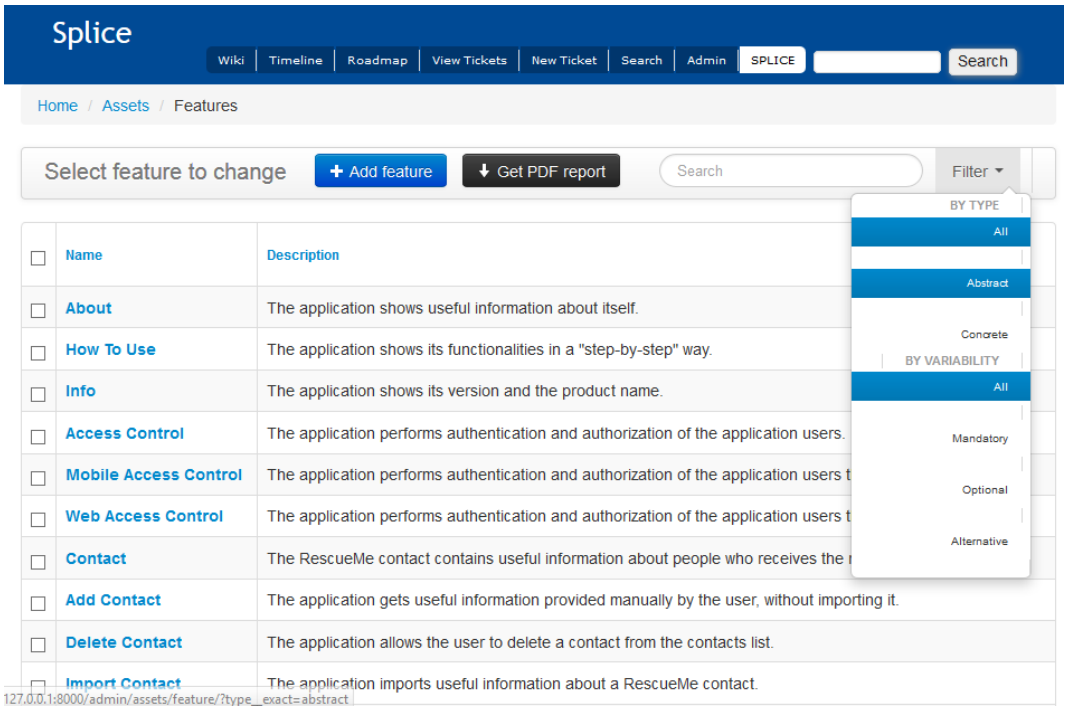


Figure 3.10 SPLICE features filtering

Issue Tracking

Following the functional requirement *FR4 - Issue Tracking*, in Figure 3.11 it is shown a screen of ticket creation. A ticket is defined as a software artifact that describes some defect, enhancement, change request, or an issue in general, that is submitted to an issue tracker. The SPLICE thanks to the *trac* core, have a full-featured Issue Tracking. One important addition to improve traceability in SPLICE is the ability to close and to reference *Tickets* during a commit to the VCS by mentioning it on the commit description. Making possible to trace back the set of changes related to an Issue.

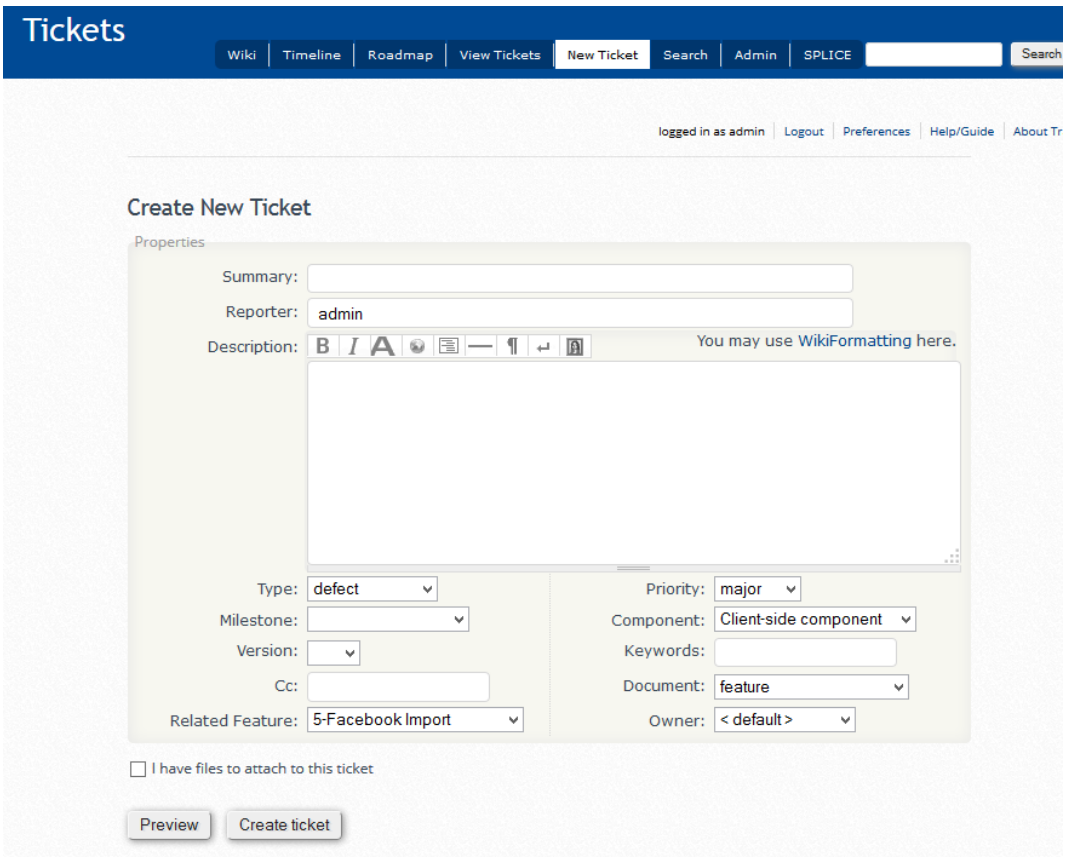


Figure 3.11 SPLICE Issue tracking

Traceability

Implementing the requirement *FR1 - Traceability*, Figure 3.12 depicts a *Feature* report. The SPLICE provides total traceability for all assets in the metamodel, and is able to report direct and indirect relations between them. In reports, asserts have hyperlinks,

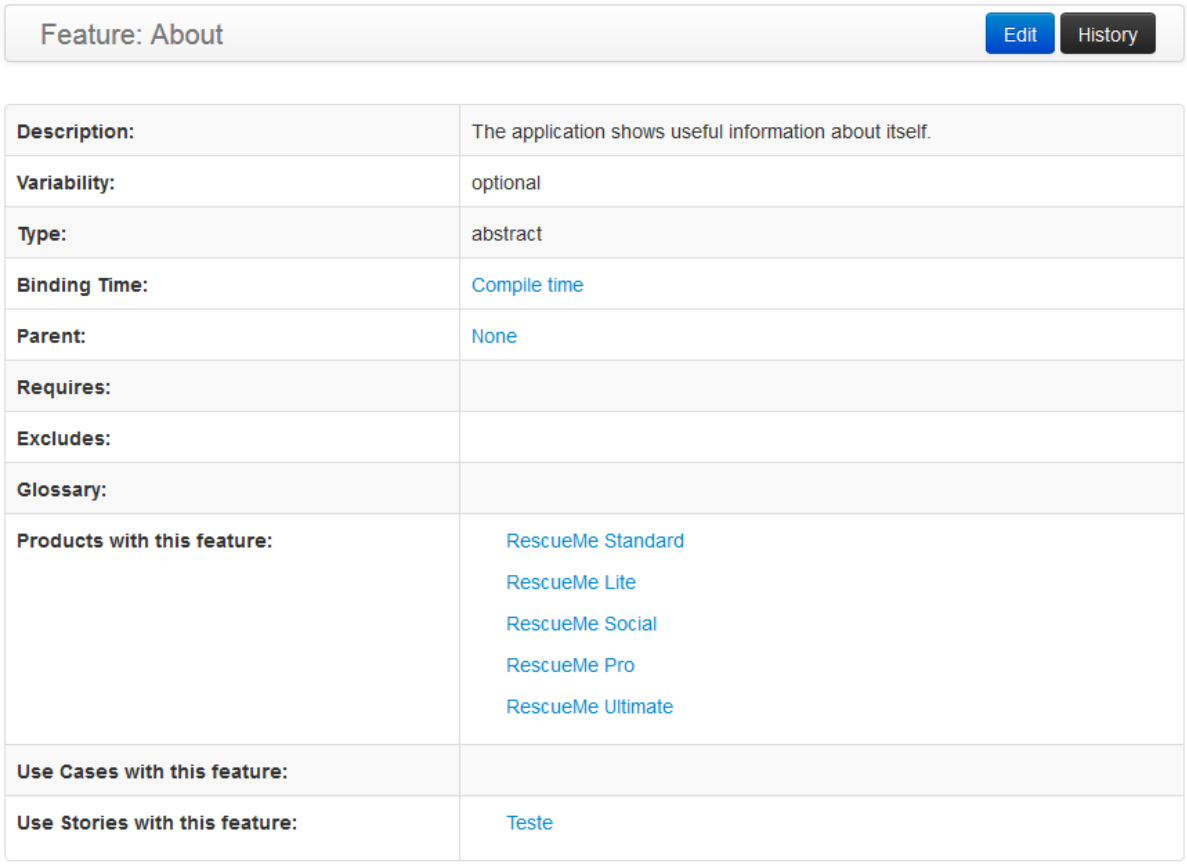


Figure 3.12 SPLICE Feature Traceability

enabling the navigation between them.

Product Selection

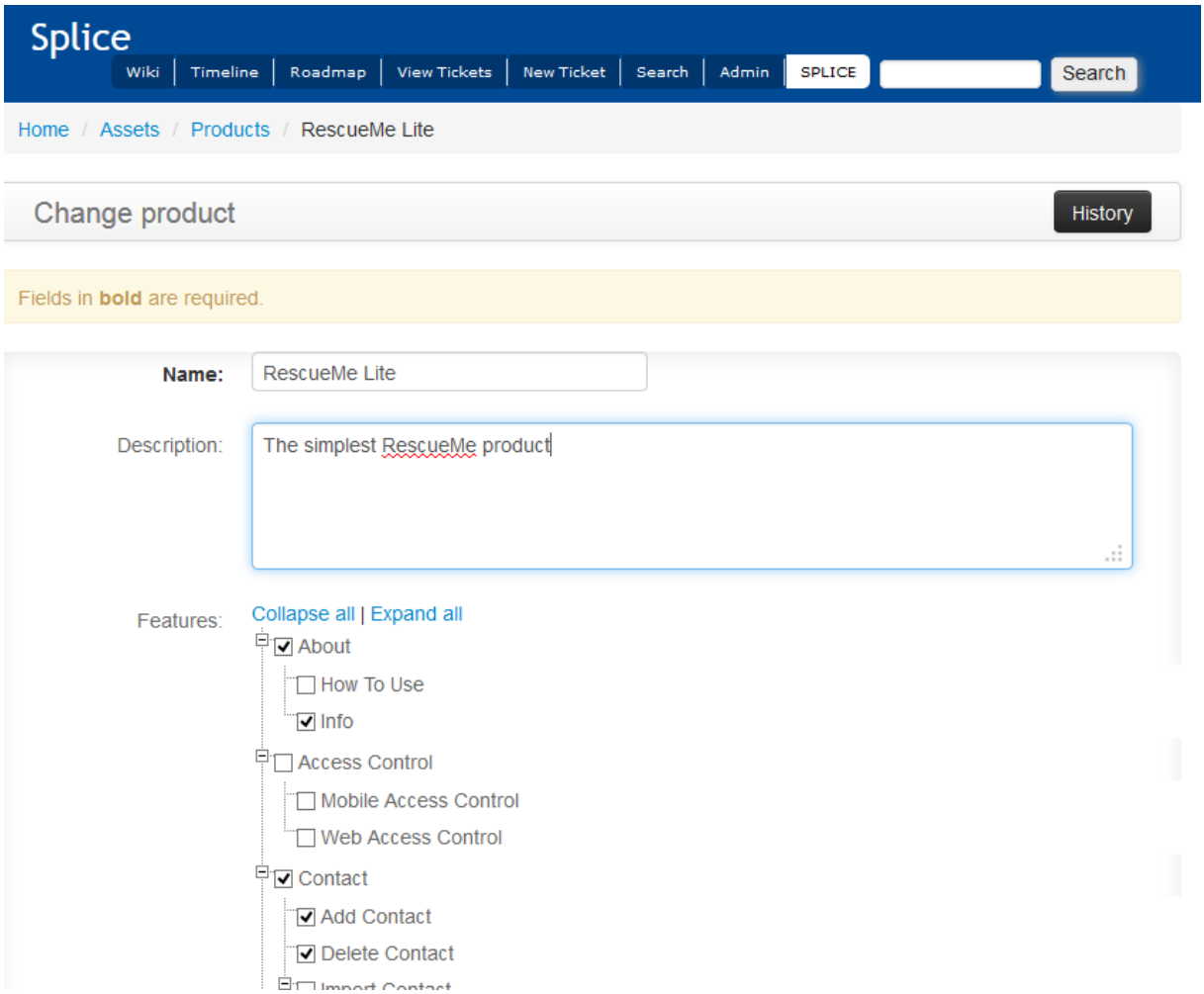


Figure 3.13 SPLICE Product edit

The SPLICE have a set of custom widgets to represent specific SPL models. One example is presented in Figure 3.13. It depicts a modification of a *Product* item. A product according to the metamodel presented in Section 3.3 is composed of a *Name*, *Description* and a set of *Features*. Those *Features* have a number of rules and restriction, such as: "Ownership"; "Dependency"; and "Exclusion". The custom checkbox tree control is able to handle those cases, representing the features as a tree, and automatically checking or unchecking related Features.

Change history and Timeline

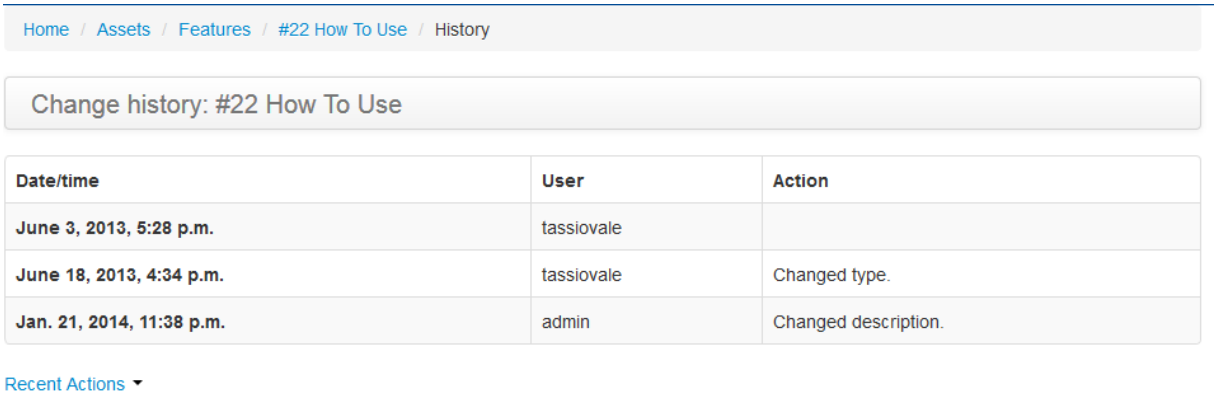


Figure 3.14 SPLICE change history

The SPLICE have a rich set of features to visualize how the project is going, where the changes are happening, and who did it. Following the functional requirement *FR2 - Reporting of lifecycle artifacts*, for every Issue or Asset, a complete Change history is recorded. As Figure 3.14 show, the user, date/time and the performed action is automatically registered. Figure 3.15 shows a timeline that the SPLICE created containing all tickets transactions, VCS Change-sets and documentation changes.

Control Panel

Figure 3.16 shows the SPLICE control panel. As the requirement *FR7 - Unified User management* demands, it aggregates the configuration of all external tools in a unified interface. Figure 3.16 depicts specifically the user-account management. 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 as required by the *FR5 - Agile Planning* requirement and is depicted in Figure 3.17. It shows an 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.

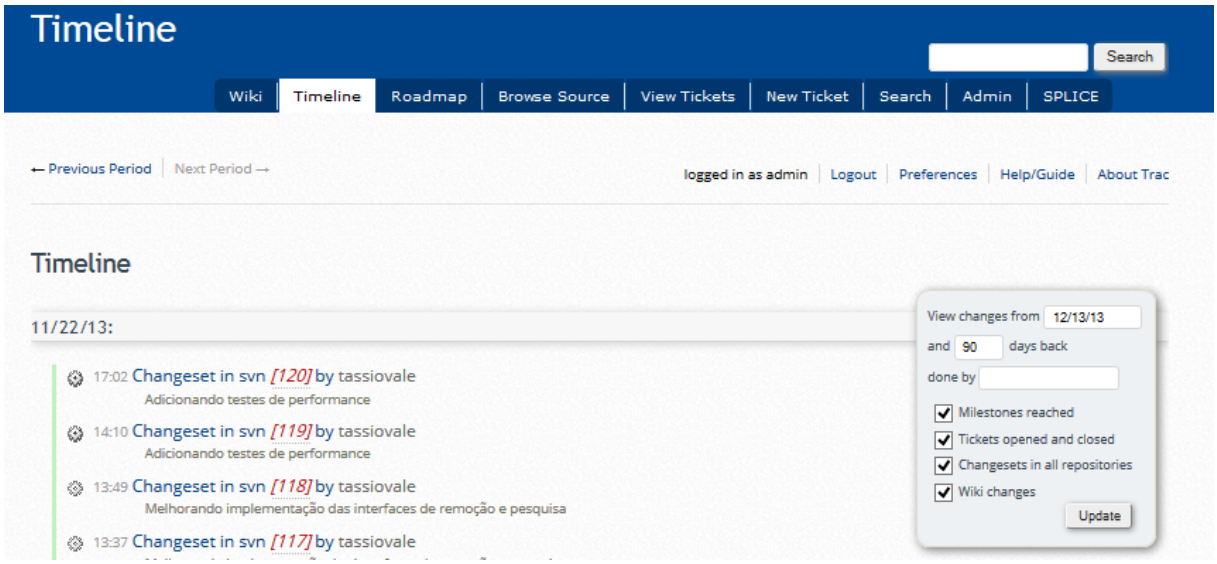


Figure 3.15 SPLICE Timeline

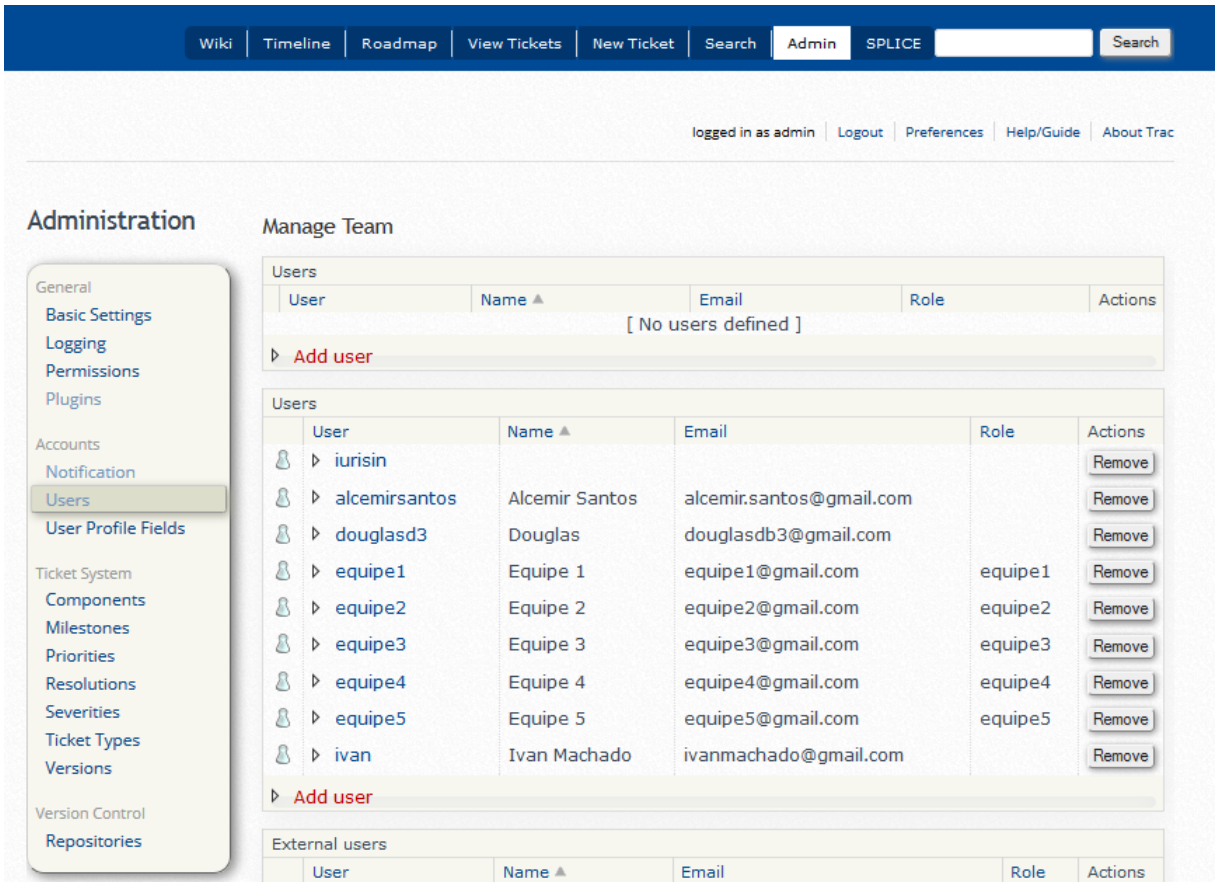


Figure 3.16 SPLICE Control Panel

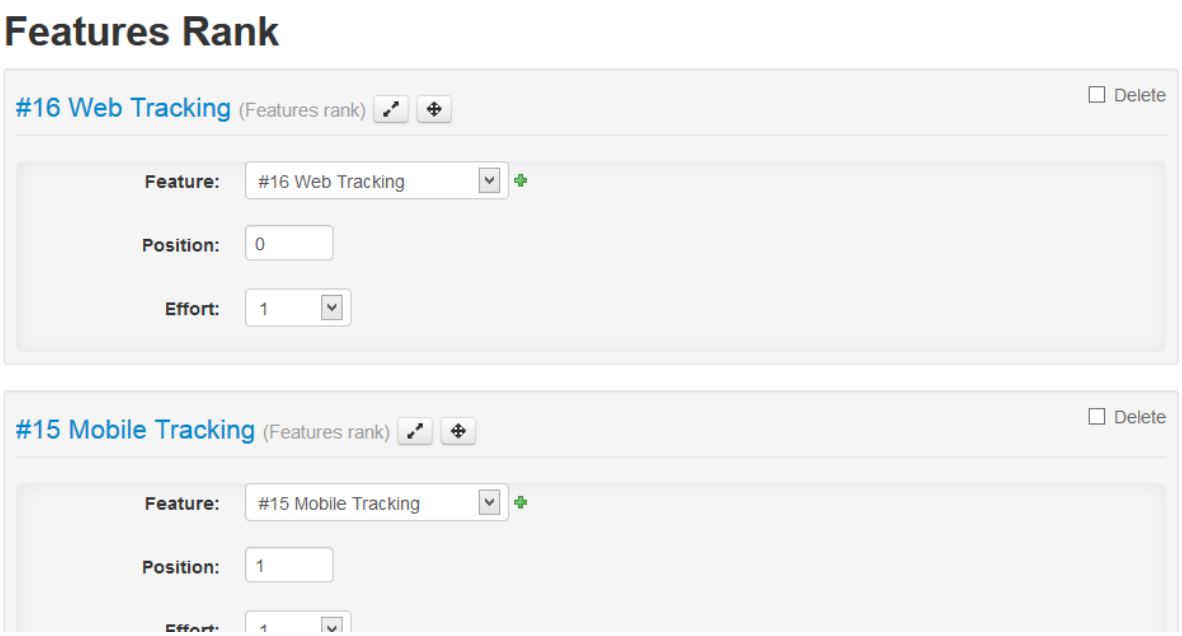


Figure 3.17 SPLICE agile feature rank

Version control systems (VCS) view

Following the FR6 - Configuration management functional requirement specification, in Figure 3.18 can be seen the SPLICE VCS repository browser. In this figure, two different kinds of VCS are running simultaneously, *SVN* and *Git*. In this tool, is possible to view the log of any file or directory or a list of all the files changed, added or deleted in any given revision. Is possible to see the differences (diff) between two versions of a file so as to see exactly what was changed in a particular revision.

Automatic reports generation

Based on the functional requirement *FR2 - Reporting of lifecycle artifacts*, the SPLICE have the capacity of creating reports, including *PDFs*, the *de facto* standard for printable document. The generated report is depicted on Figure 3.20 and includes a cover, a summary and the set of the chosen artifact related to the product. This format is suited for presenting to the stakeholders for requirements validation. The tool is also able to collect all reports for a given Product, and create a compressed file containing the set of generated reports, as can be seen in Figure 3.19.

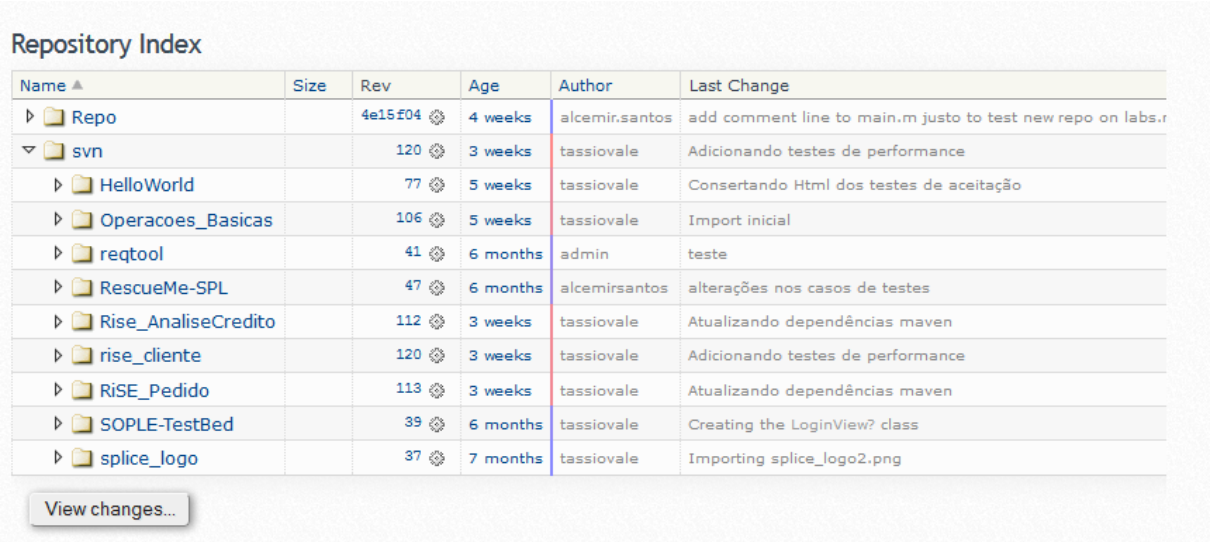


Figure 3.18 SPLICE Repository view

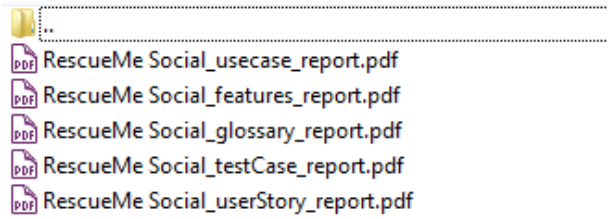
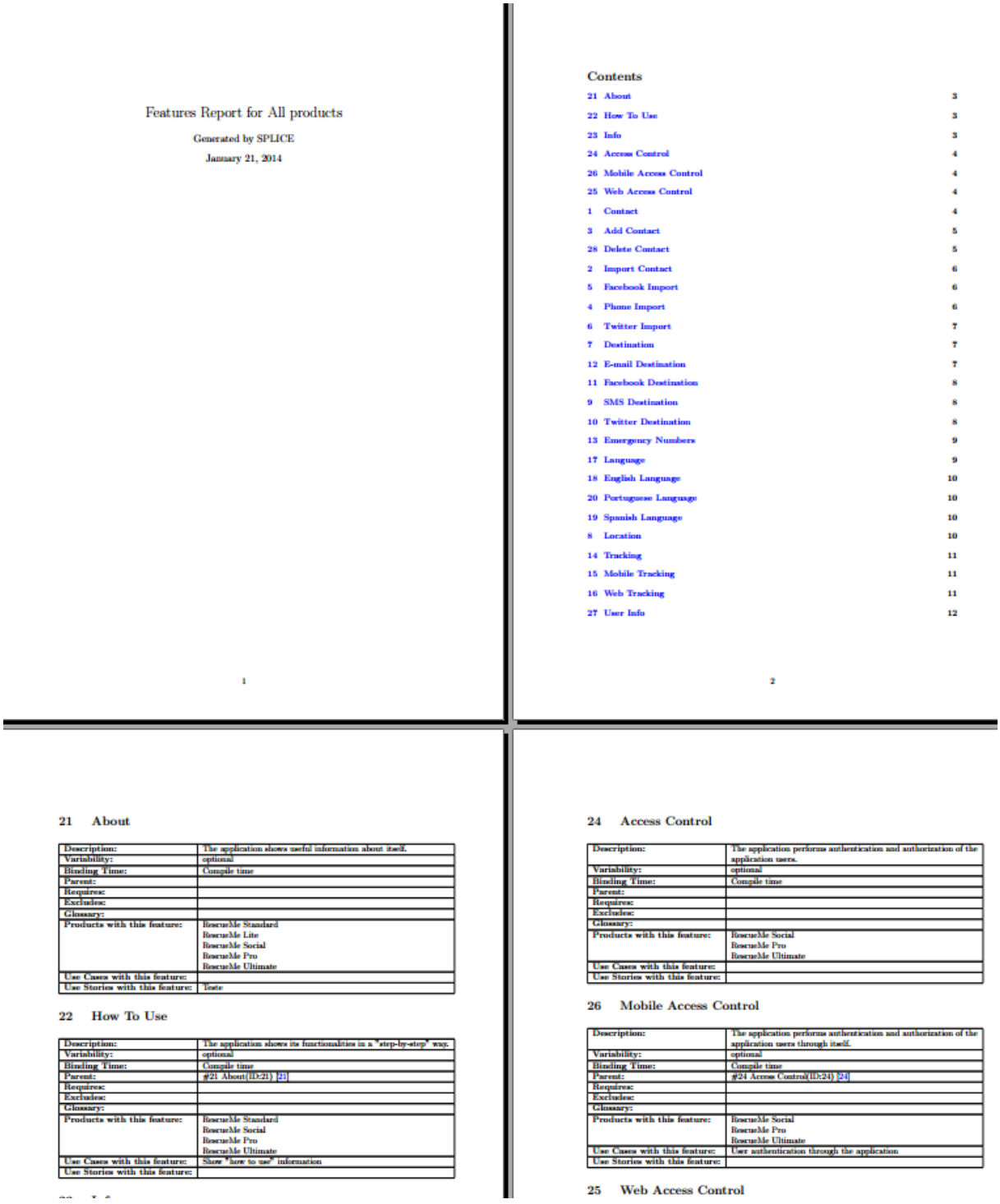


Figure 3.19 SPLICE compressed reports



Features Report for All products

Generated by SPLICE

January 21, 2014

1

Contents

2

21 About

Description:	The application shows useful information about itself.
Variability:	optional
Binding Time:	Compile time
Parent:	
Requires:	
Excludes:	
Glossary:	
Products with this feature:	RescueMe Standard RescueMe Lite RescueMe Social RescueMe Pro RescueMe Ultimate
Use Cases with this feature:	
Use Stories with this feature:	Teste

22 How To Use

Description:	The application shows its functionalities in a "step-by-step" way.
Variability:	optional
Binding Time:	Compile time
Parent:	#21 About(ID:21) [21]
Requires:	
Excludes:	
Glossary:	
Products with this feature:	RescueMe Standard RescueMe Social RescueMe Pro RescueMe Ultimate
Use Cases with this feature:	Show "how to use" information
Use Stories with this feature:	

24 Access Control

Description:	The application performs authentication and authorization of the application users.
Variability:	optional
Binding Time:	Compile time
Parent:	
Requires:	
Excludes:	
Glossary:	
Products with this feature:	RescueMe Social RescueMe Pro RescueMe Ultimate
Use Cases with this feature:	
Use Stories with this feature:	

26 Mobile Access Control

Description:	The application performs authentication and authorization of the application users through itself.
Variability:	optional
Binding Time:	Compile time
Parent:	#24 Access Control(ID:24) [24]
Requires:	
Excludes:	
Glossary:	
Products with this feature:	RescueMe Social RescueMe Pro RescueMe Ultimate
Use Cases with this feature:	User authentication through the application
Use Stories with this feature:	

25 Web Access Control

Figure 3.20 SPLICE PDF report

3.7 Summary

In this chapter, it was presented a web-based tool for SPL lifecycle management, including the set of functional and non-functional requirements, architecture, frameworks and technologies adopted during its construction. Furthermore, it presented the proposal for an lightweight metamodel that represents the interaction among asserts of a SPL during its lifecycle. Next chapter presents a case study performed during the development of a private project.

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.

5
Conclusion

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

—WINSTON CHURCHILL (The End of the Beginning)

Software Product Line (SPL) has proven to be the methodology for developing a diversity of software products and software-intensive systems at lower costs, in shorter time, and with higher quality. Many reports document the significant achievements and experience gained by introducing software product lines in the software industry (Pohl *et al.*, 2005). However, the complexity of product lines processes implicate that tool support is inevitable to facilitate smooth performance and to avoid costly errors (Dhungana *et al.*, 2007).

An upcoming concept is the Application Lifecycle Management (ALM), which deals with approaches, methodologies and tools for integrated management of all aspects of software development. Its goal is to making software development and delivery more efficient and predictable by providing a highly integrated platform for managing the various activities of the development lifecycle from inception through deployment (Kääriäinen and teknillinen tutkimuskeskus, 2011). Chapter 2 summarized the basic concepts about software product lines, ALM tools, CASE tools and their aspects. We also presented an informal study of features available in commercial tools.

In this sense, in order to facilitate the process usage and to aid the Software Engineering (SE) during the SPL process, this dissertation presented, in Chapter 3, a lightweight metamodel for SPL using agile methodologies. We also desbribed in details, the Software Product Line Integrated Construction Environment (SPLICE) tool, build 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. We presented the functional and non-functional requirements, its architecture, as well as theinvolved frameworks and technologies.

An evaluation of the SPLICE tool was presented in Chapter 4, with a case study conducted inside the research laboratory. The results demonstrated that the SPLICE address the traceability problem and was considered useful to all experts.

5.1 Research Contribution

The main contributions of this research are described as follows:

- **Informal review of available ALM tools**. Through this study, was conducted an informal search for similar commercial or academic tools. Fourteen characteristics and functionalities that is interesting to be covered in an Application lifecycle tool. A table was created comparing all the tools, that companies can use as a way to identify the best tool according to their needs.

- **SPL Lightweight Metamodel**. Based on a previous metamodel, and the need for applying agile methodologies, we introduced a proposal for a lightweight metamodel that represents the interaction among asserts of a SPL during its lifecycle.

- **SPLICE tool**. To implement the metamodel proposed, it was presented a web-based tool for SPL lifecycle management, including the set of functional and non-functional requirements, architecture, frameworks and technologies adopted during its construction.

- **Case Study**. After the tool development, a case study was performed to evaluate the SPLICE tool and gather opinions and critics about the product, to guide further development.

5.2 Future Work

An initial prototype was developed and evaluated in this work. However, we are aware that some enhancements and features must be implemented, as well as some defects must be fixed. Experts reported some of the enhancements and defects during our survey, and others were left out because they were out of scope of a graduation project. Thus, some important aspects are described as follows:

- **Visual metamodel editing**. One main objective of this work is to present a flexible tool, giving the possibility to adapt the implemented metamodel for specific project needs. Therefore, a visual editor must be implemented during the project setup to be more intuitive to users.

- **Source code variation.** The tool actually do not support source-code variability, since the users are only able to perform assets derivation. With source-code variation support, we could archive a complete derivation process.

- **Risk Management**. According to Sommerville (2011), Risk Management (RM) is important because of the inherent uncertainties that most projects face. We did not implement this model in our metamodel. As consequence, we intend to consider this feature in the next improvements.

- **Further evaluation**. In this work, we presented a case study. A more detailed evaluation is needed by applying the proposed case study protocol in other contexts in order to provide richer findings for the stakeholders.

Bibliography

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.

Alvaro, A., Almeida, E. S., and Meira, S. L. (2006). A software component quality model: A preliminary evaluation. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06)*, pages 28–37, Washington, DC, USA. IEEE Computer Society.

Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J.-C., Rummler, A., and Sousa, A. (2010). A model-driven traceability framework for software product lines. *Softw. Syst. Model.*, **9**(4), 427–451.

Araújo, J. a., Goulão, M., Moreira, A., Simão, I., Amaral, V., and Baniassad, E. (2013). Advanced modularity for building spl feature models: A model-driven approach. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1246–1253, New York, NY, USA. ACM.

Bayer, J. and Widen, T. (2002). Introducing traceability to product lines. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, PFE '01, pages 409–416, London, UK, UK. Springer-Verlag.

Baysal, O., Holmes, R., and Godfrey, M. W. (2013). Situational awareness: Personalizing issue tracking systems. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1185–1188, Piscataway, NJ, USA. IEEE Press.

Birk, A. and Heller, G. (2007). Challenges for requirements engineering and management in software product line development. In *Proceedings of the 13th International Working Conference on Requirements Engineering: Foundation for Software Quality*, REFSQ'07, pages 300–305, Berlin, Heidelberg. Springer-Verlag.

Brito, K. S. (2007). *LIFT: A Legacy InFormation retrieval Tool*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.

Buhne, S., Lauenroth, K., and Pohl, K. (2005). Modelling requirements variability across product lines. In *Proceedings of the 13th IEEE International Conference*

on Requirements Engineering, RE '05, pages 41–52, Washington, DC, USA. IEEE Computer Society.

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.

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

Cavalcanti, R. d. O., de Almeida, E. S., and Meira, S. R. (2011a). Extending the riple-de process with quality attribute variability realization. In *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, QoSA-ISARCS '11, pages 159–164, New York, NY, USA. ACM.

Cavalcanti, Y. a. C., do Carmo Machado, I., da Mota, P. A., Neto, S., Lobato, L. L., de Almeida, E. S., and de Lemos Meira, S. R. (2011b). Towards metamodel support for variability and traceability in software product lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 49–57, New York, NY, USA. ACM.

Cavalcanti, Y. C., Martins, A. C., Almeida, E. S., and Meira, S. R. L. (2008). Avoiding duplicate cr reports in open source software projects. In *The 9th International Free Software Forum (IFSF'08)*, Porto Alegre, Brazil.

Cavalcanti, Y. C., do Carmo Machado, I., da Mota Silveira Neto, P. A., and Lobato, L. L. (2012). *Software Product Line - Advanced Topic*, chapter Handling Variability and Traceability over SPL Disciplines, pages 3–22. InTech.

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.

da Mota Silveira Neto, P. A. (2010). A regression testing approach for software product lines architectures.

de Oliveira, T. H. B. (2009). Riple-em: A process to manage evolution in software product lines.

de Souza Filho, E. D. (2010). Riple-de: A domain design process for software product lines.

Dhungana, D., Rabiser, R., Grünbacher, P., and Neumayer, T. (2007). Integrated tool support for software product line engineering. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 533–534, New York, NY, USA. ACM.

Dhungana, D., Grünbacher, P., and Rabiser, R. (2011). The dopler meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering*, **18**(1), 77–114.

do Carmo Machado, I. (2010). Riple-te: A software product lines testing process.

Durao, F. A. (2008). *Semantic Layer Applied to a Source Code Search Engine*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.

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

Goeschl, S., Herp, M., and Wais, C. (2010). When agile meets oo testing: A case study. In *Proceedings of the 1st Workshop on Testing Object-Oriented Systems*, ETOOS '10, pages 10:1–10:5, New York, NY, USA. ACM.

Hochmüller, E. (2011). The requirements engineer as a liaison officer in agile software development. In *Proceedings of the 1st Workshop on Agile Requirements Engineering*, AREW '11, pages 2:1–2:4, New York, NY, USA. ACM.

Kääriäinen, J. and teknillinen tutkimuskeskus, V. (2011). *Towards an Application Lifecycle Management Framework*. VTT julkaisuja. VTT.

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

Kitchenham, B. 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.

Lacheiner, H. and Ramler, R. (2011). Application lifecycle management as infrastructure for software process improvement and evolution: Experience and insights from industry. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 286–293.

Lisboa, L. B. (2008). Toolday - a tool for domain analysis.

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

Mascena, J. C. C. P. (2006). *ADMIRE: Asset Development Metric-based Integrated Reuse Environment*. Master's thesis, Federal Uniersity of Pernambuco, Recife, Pernambuco, Brazil.

Melo, Buregio, E. S. A. M. (2008). A reuse repository system: The core system. In *Proc. of ICSR*.

Mendes, R. C. (2008). *Search and Retrieval of Reusable Source Code using Faceted Classification Approach*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.

Moraes, M. B. S. (2010). *A scoping approach for software product lines*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.

Nascimento, L. M. (2008). *Core Assets Development in SPL - Towards a Practical Approach for the Mobile Game Domain*. Master's thesis, Federal University of Pernambuco, Recife, Pernambuco, Brazil.

Neiva, D. F. S. (2009). Riple-re: A requirements engineering process for software product lines.

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.

Ribeiro, H. B. G. (2010). An approach to implement core assets in service-oriented product lines.

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

Santos, E. C. R., ao, F. A. D., Martins, A. C., Mendes, R., Melo, C., Garcia, V. C., Almeida, E. S., and Meira, S. R. L. (2006). Towards an effective context-aware proactive asset search and retrieval tool. In *6th Workshop on Component-Based Development (WDBC'06)*, pages 105–112, Recife, Pernambuco, Brazil.

Schubanz, M., Pleuss, A., Pradhan, L., Botterweck, G., and Thurimella, A. K. (2013). Model-driven planning and monitoring of long-term software product line evolution. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 18:1–18:5, New York, NY, USA. ACM.

Schwaber, C. (2006). The Changing Face Of Application Life-Cycle Management by Carey Schwaber - Forrester Research.

Seidl, C., Heidenreich, F., and Assmann, U. (2012). Co-evolution of models and feature mapping in software product lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 76–85, New York, NY, USA. ACM.

Shaye, S. (2008). Transitioning a team to agile test methods. In *Agile, 2008. AGILE '08. Conference*, pages 470–477.

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

Sommerville, I. (2007). *Software Engineering*. Addison Wesley, 8 edition.

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

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

Uikey, N. and Suman, U. (2012). An empirical study to design an effective agile project management framework. In *Proceedings of the CUBE International Information Technology Conference*, CUBE '12, pages 385–390, New York, NY, USA. ACM.

Vanderlei, T. A., ao, F. A. D., Martins, A. C., Garcia, V. C., Almeida, E. S., and Meira, S. R. L. (2007). A cooperative classification mechanism for search and retrieval software components. In *Proceedings of the 2007 ACM symposium on Applied computing (SAC'07)*, pages 866–871, New York, NY, USA. ACM.

Appendix

A

Case Study Instruments

A.1 Form for Expert Survey

Name:

_____

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

_____

_____

What is your experience with Software Product Lines ?

_____

_____

There was any difficult during the execution of some activity in the tool ?

[ ] Yes. [ ] No.

In case you answered "Yes",detail the difficulty encountered:

_____

_____

_____

_____

How the usage of the tool altered your workflow ?

_____

_____

_____

_____

Did you have any problems creating assets ?

[ ] Yes. [ ] No.

In case you answered "Yes",describe the problems encountered:

_____

Did the tool gave you enough information to identify who did a specific modification in an asset ?

[ ] Yes. [ ] No.

In case you answered "No",describe what was missing:

Do you consider that tool aided the traceability among the assets?

[ ] Yes. [ ] No.

Justify:

Did you expect any traceability link not available in the tool ? If so, which one(s) ?

What is your opinions on navigating among traceability links ?

In your opinion, the reports generated were satisfactory ?

[ ] Yes. [ ] No.

If no , what was missing ?

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

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.