A Generic Method for Assembling Software Product Line Components

Daniel Correa

Universidad Nacional de Colombia

Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión

Medellín, Colombia

2020

A Generic Method for Assembling Software Product Line Components

Daniel Correa

Submitted in partial fulfillment of the requirements for the degree of:

Doctor en Ingeniería de Sistemas e Informática

Thesis Directors:

Ph.D. Raúl Mazo

Ph.D. Gloria Lucia Giraldo Gómez

Research Area:

Software Product Lines

Research Group:

Grupo de ingeniería de software

Universidad Nacional de Colombia

Facultad de Minas, Departamento de Ciencias de la Computación y la Decisión

Medellín, Colombia

2020

…To my wife, Juliana

…To my parents, Mercedes and Fernando

…To my sister, Juliana

Acknowledgments

This thesis would not have been possible without the support of many people. First, I would like to thank my thesis directors, Ph.D. Raúl Mazo, professor at the *Université Paris 1 Panthéon – Sorbonne*, and Ph.D. Gloria Lucia Giraldo Gómez, professor at the *Universidad Nacional de Colombia – Sede Medellín*. Many thanks for being available at any moment to solve questions, to review my work, to make critics, to suggest ideas, and many more. Many thanks to Ph.D. Marta Silvia Tabares, Ph.D. Roberto Lopez Herrejon, and Ph.D. Jaime Chavarriaga Lozano for agreeing to be the jury of this thesis. Many thanks to the people of the research group, Juan Carlos, Luisa, Ángela, Esteban, Camilo, León, Yan, and many more. Thanks to the people at the *Centre de Recherche en Informatique* who help me during an internship. Thanks to Tiziana Laudato and Javier Orlando Acosta for the thesis style correction. Thanks to *Universidad Nacional de Colombia* and Colciencias which provided me with the financial means to complete this thesis and made possible to present some chapters of this thesis in other countries. Many thanks to my parents, they support me in all moment and their love and dedication were essential to complete this thesis. Many thanks to my friends Sebastián, Carlos, Johana, Daniel, Andrés, Laura. Thanks to Above & Beyond, Gareth Emery, and BT your music is a wonderful source of inspiration. Thanks to all the professors who taught me many things during my entire career, including, Fernando Arango, Carlos Mario Zapata, Alvaro Monroy, Deiby Salazar, Ángela Jurado, and Diana Mira.

And finally, thanks to my family, my wife's family, and my wife who during the entire thesis were supporting me in everything and gave me love.

Abstract

Software product lines (SPL) facilitate the industrialization of software development. The main goal is to create a set of reusable software components for the rapid production of a software systems family. Many authors propose different approaches to implement and assemble the reusable components of an SPL. However, the construction and assembly of these components continue to be a complex and time-consuming process. This thesis analyzes the advantages and disadvantages of the current approaches to implement and assemble the reusable components of an SPL. Taking advantage of these elements and with the goal of developing a generic method (which can be applied to several software components developed in different software languages), we develop Fragment-oriented programming (FragOP), a framework to design, implement and reuse SPL domain components. FragOP is based on: (i) domain components, (ii) domain files, (iii) fragmentation points, (iv) fragments, (v) customization points, and (vi) customization files. FragOP was implemented in an open-source tool called VariaMos, and we also carried out three evaluations: (i) we created a clothing stores SPL, derived five different products, and discussed the results. (ii) We developed a discussion about the comparison between FragOP and other approaches. And (iii) we designed and executed a usability test of VariaMos to support the FragOP approach. The results show preliminary evidence that the use of FragOP reduces the manual intervention when assembling SPL domain components and it can be used as a generic method for assembling assets and SPL components developed in different software languages.

Keywords: software product lines, fragment-oriented programming, component development, component composition.

Resumen

Las líneas de productos de software (LPS) promueven la industrialización del desarrollo de software mediante la definición y ensamblaje de componentes reutilizables de software. Actualmente existen diferentes propuestas para implementar y ensamblar estos componentes. Sin embargo, su construcción y ensamblaje continúa siendo un proceso complejo y que requiere mucho tiempo. Esta tesis analiza las ventajas y desventajas de las diferentes estrategias actuales para implementación y ensamblaje de componentes de LPS. Con base en esto y con el objetivo de desarrollar un método genérico (el cual se pueda aplicar a múltiples componentes de software desarrollados en diferentes lenguajes), esta tesis desarrolla la programación orientada a fragmentos (FragOP), la cual define un marco de trabajo para diseñar, implementar y reutilizar componentes de dominio de LPS. FragOP se basa en: (i) componentes de dominio, (ii) archivos de dominio, (iii) puntos de fragmentación, (iv) fragmentos, (v) puntos de personalización, y (vi) archivos de personalización. Además, se realizó una implementación de FragOP en una herramienta llamada VariaMos, y se llevaron a cabo tres evaluaciones: (i) se creó una LPS de tiendas de ropa, se derivaron cinco productos y se discutieron los resultados. (ii) Se realizó una discusión acerca de la comparación de FragOP y otras propuestas actuales. Y (iii) se diseñó una prueba de usabilidad acerca del soporte de VariaMos para FragOP. Los resultados muestran evidencia preliminar de que el uso de FragOP reduce la intervención manual cuando se ensamblan componentes, y que FragOP puede usarse como un método genérico para el ensamblaje de componentes.

Palabras clave: líneas de productos de software, programación orientada a fragmentos, desarrollo de componentes, ensamblaje de componentes.

Content

List of figures

List of tables

List of listings

Glossary

Abbreviation	**Term**
ADL	Architectural description language
ANTLR	ANother Tool for Language Recognition
AOP	Aspect-oriented programming
BPEL	Business Process Execution Language
CBSE	Component-based software engineering
CLP	Constraint Logic Programming
COP	Context-oriented programming
COTS	Commercial off-the-shelf
CSS	Cascading Style Sheets
DAO	Data access object
DOP	Delta-oriented programming
DSL	Domain-specific language
DSPL	Dynamic software product lines
FragOP	Fragment-oriented programming
FM	Feature-model
FOL	First order logic
FOP	Software product lines
HTML	Hypertext Markup Language
IDE	Integrated development environment
JDBC	Java Database Connectivity
JS	JavaScript
JSP	JavaServer Pages
MDA	Model-driven architecture
MDD	Model-driven development
MDE	Model-driven engineering
MVC	Model-view-controller
OOP	Object-oriented programming
PL	Product line
PLA	Product line architecture
PLE	Product line engineering
SMS	Systematic mapping study
SOA	Service-oriented architecture
SPL	Software product lines
SPLE	Software product line engineering
UML	Unified Modeling Language
WSDL	Web Service Definition Language

Introduction

Before the advent of mass production in the age of industrialization, manufacturing processes were essentially **handcrafting**. Skilled craftsmen built physical goods, such as machines, furniture, buildings, and clothing, among others, but each product was unique in the sense that it was built from scratch. Thereafter, **mass production** became the leading production philosophy, with the use of assembly lines and standardized parts which were eventually combined to create more complex products. This philosophy improved productivity compared to handcrafting. Nevertheless, individualism was lost (or considered less important) in the sense that a manufacturer no longer incorporated the needs and wishes of individual customers. In the twentieth century, the idea of a product line emerged. A **product line** (PL) is a set of products in a manufacturer's product portfolio that share substantial similarities and that are, ideally, created from a set of reusable parts. This way, manufacturers looked to diversification in order to offer multiple products tailored to individual market segments (Apel *et al.*, 2013).

Software development has a history similar to that of the production of physical goods. Early software was **handcrafted** by a few experts. Thereafter, standardization (**mass production**) appeared alongside products such as Microsoft Word, IBM DB2, and Excel, among others. Nevertheless, these products did not address smaller market segments' needs or individuals' needs (Apel *et al.*, 2013). In order to fulfill these specific needs, the idea of software product lines was developed. A **software product line** (SPL) is a collection of software-intensive systems sharing a common, managed set of characteristics that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way (Clements & Northrop, 2001). The main objective is to avoid developing software systems from scratch, instead of that, those systems should be constructed from reusable parts.

Software product line engineering (SPLE) has gained significant attention over recent years. It has been claimed that SPLE is promising in that it provides a faster, better, and cheaper way to develop a large range of software systems (Chen & Babar, 2011). SPLE comprises two processes: (i) **domain engineering**, which defines the commonalities and the variability of the SPL, and develops the reusable domain artifacts (for instance: components, models and tests); and (ii) **application engineering** process, which derives the software systems from the reusable domain artifacts, based on the particular requirements of the stakeholder.

Proper domain component development and management is crucial to take advantage of SPLE benefits. Currently, the implementation of the reusable domain components and their subsequent assembly (product derivation) continue to be a complex, time-consuming and expensive process (Azanza *et al.*, 2010; Lahiani & Bennouar, 2017).

Why are product lines important?

As discussed above, there are several advantages to the product line production strategy. According to a study by Clements and Northrop (2001) the product line production approach decreases not only the cost per product (by as much as 60%), but also the time to market (by as much as 98%), the labor needs (by as much as 60%) and improves productivity (by as much as 10x), the quality of each derived product (by as much as 10x) and increases the portfolio size and therefore the possibility of capturing new markets. Furthermore, according to Apel *et al.* (2013), companies, such as Boeing, Bosch, Toshiba, Hewlett Packard, and General Motors have presented successful stories with the implementation of software product lines.

Why are domain implementation and product derivation activities important?

Domain implementation (an activity which is part of the domain engineering process) refers to the development of the reusable domain artifacts. Some of the most important reusable domain artifacts are reusable components (Van Ommering & Bosch, 2002). Domain components are the base on which each new application of the product line is produced. In this thesis, a **software component** is defined as a piece of self-contained code with well-defined functionality that can be reused as a unit in various different contexts (Wang & Qian, 2005).

Another important activity is product derivation (which is part of the application engineering process). In SPL application engineering, products are configured and derived (Apel *et al.*, 2013). The derivation process takes the reusable components and assembles them to generate the new software system. The degree of automation of the assembly process depends on how the components were implemented. If the process is not very automatic, the benefits of the SPLE are reduced.

Problem statement

SPL component implementation and assembly have been intensively researched during recent years. Many authors propose several approaches and tools to the design, implementation and assembly of components (Apel *et al.*, 2013), such as feature-oriented programming (FOP; Prehofer, 1997), delta-oriented programming (DOP; Schaefer, *et al.*, 2010), context-oriented programming (COP; Salvaneschi *et al.*, 2012), aspect-oriented programming (AOP; Tizzei *et al.*, 2012), service-oriented architecture (SOA; Alzahmi *et al.*, 2014), Colored Integrated Development Environment (CIDE; Kästner *et al.*, 2008), pure::variants (Beuche, 2008), GenArch (Cirilo *et al.*, 2007), GenArch-P (Aleixo *et al.*, 2013), and agents (Jordan *et al.*, 2012). Commonly, those approaches could be grouped into two main categories: annotative and compositional (Apel *et al.*, 2013). In **annotative approaches** such as pure::variants, CIDE, and GenArch, developers simply introduce markers at the exact positions where a component should be extended (Kästner *et al.*, 2008). This allows **fine-grained** extensions, *i.e.*, changes at lower levels, such as changes in a fixed position inside a class method. In **compositional approaches** such as FOP, DOP, and AOP (Apel *et al.*, 2013), components are implemented in the form of composable units. In FOP, the software assets are developed in terms of “feature modules”, which can be seen as increments of product functionality. For example, in the context of object-oriented programming (OOP), a feature module can introduce new classes or refine existing classes by adding fields and methods, or by overriding existing methods. This allows **coarse-grained** extensions, *i.e.*, changes at top levels, such as changes in the hierarchical structure of an implementation artifact. However, despite the relative success of these approaches, the derivation of individual products from shared software assets continues to be a time-consuming and expensive activity in many organizations (Deelstra *et al.*, 2005; Rabiser *et al.*, 2011; Lahiani & Bennouar, 2017).

We found several unresolved issues that motivate the research presented in this thesis:

- The area of product derivation is still rather immature (Souza *et al.*, 2015) due to the fact that: (i) there is a lack of support for the derivation process; (ii) existing approaches do not present detailed information on the strategies for product customization, resolving variability, and the derivation of additional software assets such as databases, views, images, scripts and configuration files; and (iii) in comparison with the research on the development and modeling of product lines, few approaches and tools are available for product derivation. This is an important issue because the production derivation process includes the assembly of the reusable domain components.

- Most current approaches and tools provide a mechanism for the implementation and assembly of the domain components. However, they are usually attached to very specific software languages or software paradigms. For example, (i) compositional approaches such as FOP, DOP, AOP, and COP have been called as extensions of OOP, and have been widely used to refine, modify or extend classes and objects; and are usually attached to a particular host language (Kästner & Apel, 2008). For instance, compositional tools or extensions, such as AspectJ and DeltaJ are attached to Java, and FeatureC++ is attached to C++. However, software products are not only made up of classes and objects, and are commonly written in multiple software languages. Additionally, (ii) annotative approaches implement components with some form of explicit or implicit annotations, with the prototypical example being the use of *`#ifdef`* and *`#endif`* statements that surround the component code (Kästner & Apel, 2008). Nevertheless, not all software languages provide these statements, and many annotative approaches provide limited support to only a few software languages. Indeed, software products are made up of multiple artifacts developed in multiple languages, such as programming languages (PHP, JAVA, or Python), markup languages (HTML or XML), style sheet languages (CSS), database languages (SQL or SPARQL), scripts, images, and configuration files. According to Mayer and Bauer (2015) who analyzed 1150 open source projects, a mean number of 5 different languages are used in each project. Consequently, developing system modules that can be applied to multiple languages appears to be an important concern (Kästner *et al.*, 2011).

- Annotative approaches have their own limitations in that the domain component files contain all the source code variants, which (i) increases the number of lines of code, (ii) increases the number of relationships between the domain component file and other domain component files, and (iii) tends to make source code complex and therefore difficult to maintain and evolve (Le *et al.*, 2013).

- Compositional approaches present important limitations. According to Kästner *et al.* (2008) “compositional approaches only introduce new code fragments in positions in which the order does not matter. Thus, it is possible to introduce new classes into the program or new methods into a class, but not new statements at a fixed position inside a method”. Additionally, as previously stated, compositional approaches are usually attached to a particular host language.

- The combination of compositional and annotative approaches has important advantages but also some limitations. The few approaches that try to combine annotative and compositional approaches (Kästner & Apel, 2008; Walkingshaw & Erwig, 2012; Behringer & Rothkugel, 2016; Horcas *et al.*, 2018) present similar limitations to those listed previously (limited support for few software languages, use of *if* statements, poorly detailed or no tool support). Consequently, how to combine the compositional and annotative approaches to maximize the advantages and minimize the limitations of each separate approach continues to be an important concern (Kästner & Apel, 2008).

Research questions

This thesis addresses the aforementioned problems by proposing an approach that: (i) allows the implementation and assembly of software product line components independent of their software language, (ii) combines compositional and annotative elements in order to allow fine-grained extensions, but keeps the variations as independent modules, and (iii) supports the product derivation process, trying to automate the entire component assembly process and supporting other SPL activities in which SPL components are involved (such as the customization activity). Consequently, the proposed approach is generic; it allows for the implementation of multiple software components developed in several software languages and assembles them properly. Thus, the main objective of this thesis is to answer the following research question:

Main RQ: How can software product line components be automatically assembled independent of their software language in a generic and reusable way?

The resolution of each of the following four research questions is necessary to solve the main research question of the thesis.

RQ1: How should software product line components be implemented to guarantee a generic assembly and to contain the variation code independent?

To answer this question, this thesis proposes a new paradigm called Fragment-oriented programming (FragOP) which is presented in Chapter 3. This paradigm combines elements from compositional and annotative approaches, and so allows variation points to be represented in most software languages and for them to contain the variation code independently. In this case, SPL components are implemented through the development of domain components, domain files, fragmentation points, fragments, customization points, and customization files (presented in Chapter 4).

RQ2: How should software product line components be assembled to reduce manual intervention as much as possible?

To answer this question, this thesis proposes a new derivation process which moves the domain component files to their final destination and applies the fragment alterations (see Section 5.6). A fragment can inject or modify code developed in several languages.

RQ3: How should software product line components be verified?

This thesis develops a new approach which assembles domain components developed in several languages. Then, it is important to verify that those components are properly assembled. This task is more complex for this approach due to the multiple language support. We take advantage of ANTLR (ANother Tool for Language Recognition), which is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions. We used a series of parsers and lexers for languages, such as PHP, Java, CSS, and MySQL, among others. Based on the derived component file extension, the grammar of each file is analyzed, and alerts are generated if errors are found (see Section 5.8).

<u>RQ4:</u> *What support can be offered to systems engineers for improving software product line component assembly?*

As previously stated, tool support is crucial for managing the SPL, as assembling thousands of components manually voids the benefits of SPL adoption. In this thesis, a software tool called VariaMos (Mazo *et al.*, 2015) is extended to provide support to the FragOP process (see Chapter 5).

Research hypothesis

Based on the aforementioned problems and objectives, the following main research hypothesis was defined:

<u>Main RH:</u> A generic method will allow software product line components to be automatically assembled independent of the software language in which they were implemented.

The main research hypothesis can be better understood by disaggregating and explaining some of the concepts:

What is meant by "generic method"? We define generic as the method's suitability for use in different cases and contexts.

What is meant by "automatic assembly"? A tool or software automates the assembly process, reducing manual intervention as much as possible.

What is meant by "software product line components"? The domain software assets developed in the domain SPL engineering process, which are reused to generate new software systems.

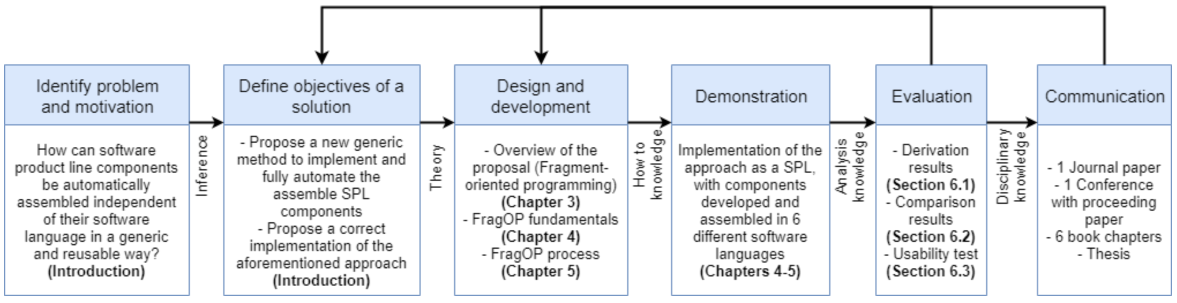
Research method

In order to test the research hypotheses of this thesis, the following research strategy was implemented:

1. We conducted a systematic mapping study on product line engineering. In particular on the techniques, methods, and tools for SPL component implementation and assembly.
2. We identified a set of gaps and drawbacks of the existing approaches with regard to the present research question. Specifically, we examined how solutions could be used together to address the problem tackled by this thesis.
3. We proposed a new generic method to implement and assemble SPL components which was automated.
4. We tested the new method through the development of a running example, and the assembly and derivation of 5 different software systems.
5. We evaluated the correctness of the proposed approach through a laboratory case study which evaluated the proposal's usability.

This research methodology matches the design science process model proposed by Peffers *et al.* (2007) exactly. Figure I-1 presents, in blue, the design science process model for information system research, and the application of this process to the research carried out in this thesis.

Figure I-1: Application of the design science process model for information system research (Peffers *et al.*, 2007) to the research carried out in this thesis


Identify problem and motivation
How can software product line components be automatically assembled independent of their software language in a generic and reusable way? **(Introduction)**
Inference
Define objectives of a solution
- Propose a new generic method to implement and fully automate the assemble SPL components
- Propose a correct implementation of the aforementioned approach **(Introduction)**
Theory
Design and development
- Overview of the proposal (Fragment-oriented programming) **(Chapter 3)**
- FragOP fundamentals **(Chapter 4)**
- FragOP process **(Chapter 5)**
How to knowledge
Demonstration
Implementation of the approach as a SPL, with components developed and assembled in 6 different software languages **(Chapters 4-5)**
Analysis knowledge
Evaluation
- Derivation results **(Section 6.1)**
- Comparison results **(Section 6.2)**
- Usability test **(Section 6.3)**
Disciplinary knowledge
Communication
- 1 Journal paper
- 1 Conference with proceeding paper
- 6 book chapters
- Thesis

Contributions

To overcome the limitations presented in this introduction, this thesis proposes a generic method for defining domain components and using them to create new products. Specifically, the main contributions of this thesis are the following:

1. **A generic approach for implementing software product line components.** In this step, we developed a new approach that is called Fragment-oriented programming (FragOP). FragOP is a framework to design, implement and reuse SPL domain components. It is based on the development of domain components, domain files, fragmentation points, fragments, customization points, and customization files. This approach is also a mix of compositional and annotative approaches (see Chapter 3).
2. **A derivation activity which includes component assembly.** Based on the FragOP elements, we developed a new derivation activity, which includes a series of steps that take information from the component pool and the model information and derive specific software systems. These steps also detail the management of the domain components assembly, allowing the component code to be reused and minimizing manual intervention (see Section 5.6).
3. **A customization activity.** The addition of customization points and customization files improves customization activity. After derivation activity, SPL developers are able to recognize and automate the customization of the derived software products (see Section 5.7).
4. **A verification method for the derived product files.** As previously stated, this approach innovates with component assembly independent of software language. This means that SPL developers are able to create multiple domain components and multiple code variations in languages such as Java, HTML, CSS, and MySQL among others. Consequently, developing a mechanism for early error detection and syntax validation becomes crucial. For this, we used ANTLR and an analysis of the derived product files' grammar (see Section 5.8).
5. **A supporting tool for the proposal.** We extended the capabilities of the VariaMos tool to support the FragOP approach (see Section 3.3).
6. **Three evaluations of the proposal.** (i) We designed a running example to gain insights about if VariaMos (FragOP) was expressive enough to implement a real world, variant-rich multi-language software system (see Section 6.1). (ii) We defined

and discussed the differences between VariaMos (FragOP) and similar SPL implementation mechanisms and tools (see Section 6.2). Finally, (iii) we applied a usability test to the new approach and its implementation from which we found preliminary evidence that VariaMos is a usable tool that properly supports the FragOP approach (see Section 6.3).

Thesis organization

This thesis is organized as follows:

Chapter 1 presents related work through a systematic mapping study (SMS) on SPL implementation. It also analyzes the advancements, gaps, and challenges found in the literature on this topic.

Chapter 2 presents a running example called ClothingStores. ClothingStores is a software product line that consists of the development of an e-commerce store system family to manage and sell clothes. This chapter describes the SPL requirements and SPL software architecture. The running example is used in Chapters 4 and 5 to provide a real scenario to show how the new approach works.

Chapter 3 provides an overview of the generic method presented in this thesis. It presents the FragOP metamodel, the FragOP process and its eight main activities, and introduces VariaMos, a software tool that supports the FragOP approach.

Chapter 4 presents the FragOP fundamentals (the six main elements of the FragOP metamodel), explaining how these elements support the two main FragOP capabilities (assembly and customization).

Chapter 5 presents the FragOP process. It describes and details each of the eight activities, (i) modeling PL requirements, (ii) modeling domain components, (iii) implementing domain components, (iv) binding domain requirements and domain components, (v) configuring products, (vi) deriving products, (vii) customizing products, and (vii) verifying products. Therefore, for each activity, we present how to use VariaMos to support them, and we provide a practical application by using a running example.

Chapter 6 provides an evaluation of the proposal. We discuss the running example results, present a comparison of the FragOP approach and contemporary studies, and develop a usability test to see how VariaMos supports the FragOP approach.

Finally, **Chapter 7** concludes the thesis and proposes future research directions.

1. State of the art

As previously stated, software components are some of the most important SPL artifacts, as they are the base on which each new application of the product line is produced. SPL components are used throughout several SPLE stages such as binding, development, testing, evolution, and derivation, among others. Due to the way SPL components are integrated into several of those stages, we decided to develop a systematic mapping study (SMS) on SPL implementation. This SMS provides an overview of the processes, methods, and tools used to carry out SPL implementation, but also provides details on the role of the SPL components in the entire process.

An SMS is a form of a secondary study aiming to provide a comprehensive overview of a certain research topic, to identify gaps in the research, and to collect evidence in order to guide researchers and practitioners in their current or future work (Wohlin *et al.*, 2013). It allows all available studies in a domain to be analyzed at a high level thereby answering broad research questions regarding the current state of the research on a topic. Another form of secondary study is a systematic literature review (SLR), which aims to identify, evaluate, and interpret all available studies to answer specific research questions (usually in the form: Is technology/method A better than B or not?), and requires more in-depth analysis (Kitchenham & Charters, 2007). We decided to conduct an SMS instead of an SLR because (i) we wanted to provide thematic analysis of the current studies developed in the SPL implementation field; (ii) we wanted to conduct a generic study (related to research trends) with high-level questions of the form: which researchers, how much activity, what type of studies, etc., instead of an specific study (related to outcomes of empirical studies); and (iii) we wanted to conduct a broad study because we were at the beginning of this PhD research project and we wanted to build a big picture of the research project area.

The main goals of this SMS are:

- (G1) to provide basic publication information through a demographic method and to assist researchers in identifying the most appropriate sources of research information on SPL implementation.
- (G2) to assist researchers and practitioners in identifying the range of methods and mechanisms currently available for an SPL implementation (including SPL component implementation).
- (G3) to identify principal research trends in the literature and highlight active research topics that require more future work.

To achieve these objectives, this SMS will answer a selection of the research questions presented in Section 1.2.1.

The remainder of this section is organized as follows: Section 1.1 presents the related work; Section 1.2 describes the systematic mapping process, the definition of the research goals and questions, the search strategy, and the search conduction are presented; Section 1.3 presents the data extraction; Section 1.4 presents the answers to questions about research goal G1 – Publication; Section 1.5 presents the answers to questions about research goal G2 – SPL implementation; Section 1.6 presents the answers to questions about research goal G3 – Topics and trends; Section 1.7 discusses the threats to validity; and finally, Section 1.8 concludes this study and recommends the direction of future work.

Note: It is important to highlight that the references used in the SMS can be found in Appendix B. Therefore, some of these references will be presented in this section with the IEEE format, and they will be preceded by the letter S.

1.1 Related work

During the last decade, many authors have carried out SMS, SLR and surveys on SPL [S11, S12, S13, S33, S54, S55, S71, S77, S82, S87], with these contributions covering many aspects of the field of SPL. However, most of them involve a specific part or even a specific method for the SPL implementation process. As a result, a complete understanding of SPL implementation is not given. Therefore, the research questions posed in this SMS are legitimate and previous studies have not yet answered them. A summary of the currently available SMS, SRL, and survey studies is presented below.

Lee *et al.* [S11] presented a survey framework that consists of eight SPL specific testing perspectives and compared and analyzed the contributions of selected studies. Neto *et al.* [S12] developed a systematic mapping study on Testing in SPL in which 120 studies were evaluated. They found a huge amount of approaches that handle different and specific aspects in the SPL testing process, however, the quantity of approaches makes comparing studies a difficult task. Through this study, they were able to identify which activities are handled by the existing approaches as well as understanding how researchers are developing work on SPL testing.

Laguna and Crespo [S13] carried out a study which aimed to survey the existing research on the reengineering of legacy systems like SPLs and the refactoring of existing SPLs. Guided by several parameters, 74 papers were selected and classified. The results of the study indicate that the initial works focused on the adaptation of generic reengineering processes to SPL extraction. Several trends were detected in the research: the integrated or guided reengineering of (typically object-oriented) legacy code and requirements; specific aspect-oriented or feature-oriented refactoring of SPLs, and refactoring for the evolution of existing product lines. Most papers included academic or industrial case studies, although only a few were based on quantitative data. Montalvillo and Díaz [S71] conducted a mapping study on SPL evolution that included 107 articles. They developed a classification schema that included four facets: evolution activity, product-derivation approach, research type, and asset type. The results show that regarding the evolution activity, “Implement change” (43%) and “Analyze and plan change” (37%) were the most covered contributions.

Mohabbati *et al.* [S33] developed a systematic mapping study on the combination of service-orientation and SPLE. In this SMS, 81 primary studies were selected. Their research focused on service variability modeling, service identification, service reuse, service configuration and customization, dynamic software product lines, and adaptive systems. The results show that SPLE approaches, especially feature-oriented approaches for variability modeling, have been applied to the design and development of service-oriented systems. Service-orientation is employed in software product line contexts for the realization of product lines to reconcile the flexibility, scalability, and dynamism in product derivations thereby creating dynamic software product lines.

Afzal *et al.* [S54] conducted a literature review of both research and industrial artificial intelligence applications for SPL configuration issues. They found 19 relevant research papers which employ traditional artificial intelligence techniques on small feature sets with no real-life testing or application in industry. Finally, they showed that only 2 standard industrial SPL tools employ artificial intelligence in a limited way to resolve inconsistencies.

Méndez-Acuna *et al.* [S55] developed a literature review in which they reported an attempt to organize the literature on language product line engineering. More precisely, they proposed a definition for the life-cycle of language product lines, and they used it to analyze the capabilities of current approaches (38 studies were included). In addition, they mapped each approach and the technological space supported by it.

Vale *et al.* [S77] conducted a systematic mapping study to investigate the state-of-art of the SPL traceability area, in which 62 primary studies were identified. The results showed that the common strategies for systematizing traceability were metamodeling, different representation structures, model transformations, formal methods, and trace recovery techniques. Most strategies focus on the trace creation activities, with a lack of planning, maintenance, and use of SPL traces.

Dos Santos Rocha and Fantinato [S82] performed an SLR with four research questions formulated to evaluate PL approaches for BPM (63 papers were selected). The results showed that the PL approaches found for BPM only partially cover the BPM lifecycle, not taking into account the last phase which restarts the lifecycle. Therefore, the results indicate that PL approaches for BPM are still at an early stage and are gaining maturity.

Mazo *et al.* [S87] carried out a literature review. Their objective was to identify and analyze the different ways for improving ERP engineering issues with the methods, techniques, and tools provided by PLE. Their literature review analyzed six research papers and found that there is still a lack of interest in addressing ERP engineering issues with the product line strategy.

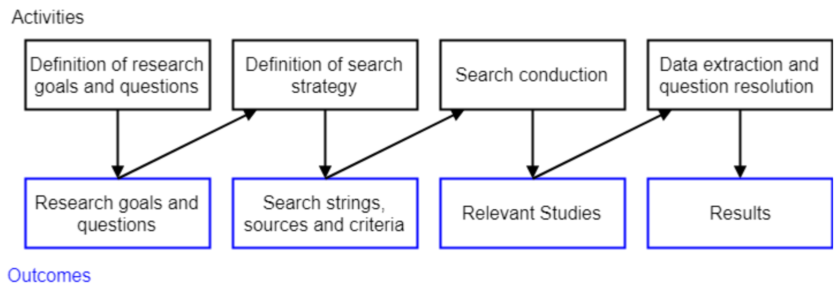
Pereira *et al.*, (2015) developed an SLR which analyzed the available literature on SPL management tools. In this study, 52 papers were included. They identified 41 tools in the literature that provide support to at least one SPL management phase.

Finally, Marimuthu & Chandrasekaran (2017) conducted a systematic mapping study of existing systematic studies of software product lines (tertiary study). They analyzed 60 relevant studies to highlight the SPL research topics, type of published reviews, active researchers and publication forums.

1.2 Systematic mapping process

As previously mentioned, the method used in this research is an SMS. We applied the mapping studies guidelines proposed by Petersen *et al.* (2008), which compares the methods used in mapping studies and SLR. The specific systematic mapping process reported in this paper was performed based on those guidelines and represented in Figure 1-1 as a sequence of activities and their corresponding outcomes. Even if it is not possible to conduct a mapping study or a literature review in a fully objective manner, the guidelines used in our systematic mapping process on SPL implementation renders the study less subjective thanks to pre-defined data types and criteria that narrow the scope for personal interpretation.

Figure 1-1: The systematic mapping process


Activities
Definition of research goals and questions
Definition of search strategy
Search conduction
Data extraction and question resolution
Research goals and questions
Search strings, sources and criteria
Relevant Studies
Results
Outcomes

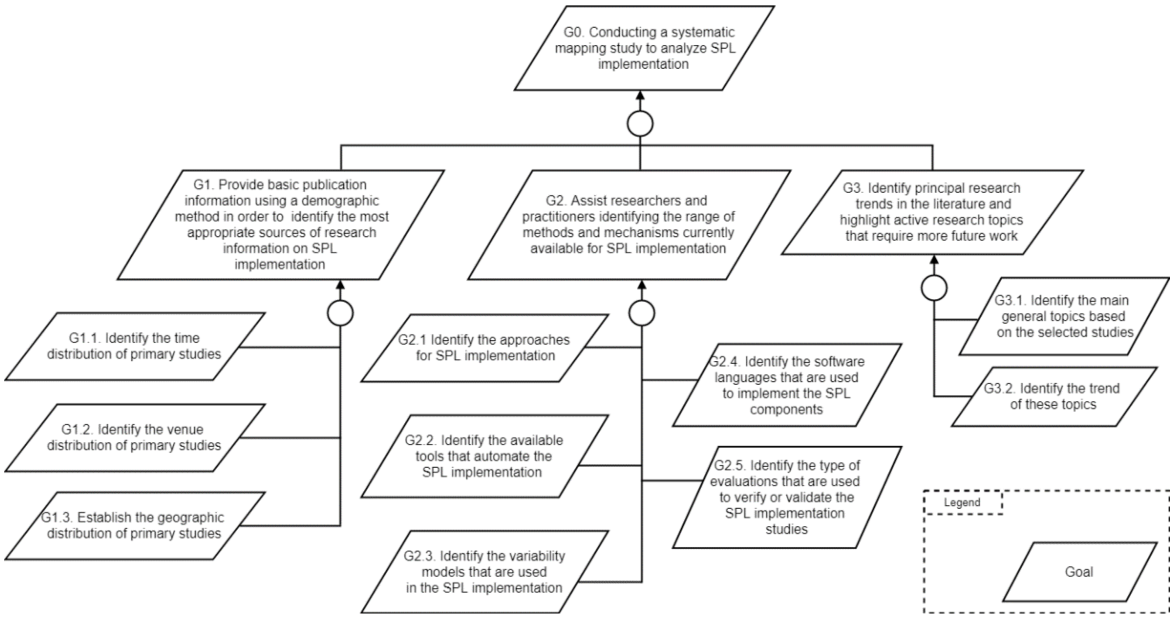
The main activities of this systematic mapping process are (i) definition of research goals and questions, (ii) definition of search strategy, (iii) search conduction, and (iv) data extraction and question resolution. The first three activities are described in the next three subsections Section 1.2.1, Section 1.2.2, and Section 1.2.3; the last activity is described in Section 1.3.

1.2.1 Definition of research goals and questions

Figure 1-2 shows a KAOS diagram (Heaven & Finkelstein, 2004) that details the research goals of this study. The main research goal (G0) refers to the carrying out of the SMS on SPL implementation. G0 is broken down into three primary sub-goals and these primary sub-goals into secondary sub-goals; the secondary sub-goals will serve as a base for the definition of the research questions.

- Primary sub-goals G1 and G3 represent the researchers' perspective. Their secondary sub-goals provide an overview of how different researchers are dealing with SPL implementation in different countries, in different laboratories, and where these contributions are available and what the current trends are.
- Primary sub-goal G2 represents the practitioners, industry and developers' perspectives. Its secondary sub-goals provide an overview of current methodologies for implementing an SPL, the available tools, software languages, and evaluations, among others.

Figure 1-2: Systematic mapping study research goals


G0. Conducting a systematic mapping study to analyze SPL implementation
G1. Provide basic publication information using a demographic method in order to identify the most appropriate sources of research information on SPL implementation
G2. Assist researchers and practitioners identifying the range of methods and mechanisms currently available for SPL implementation
G3. Identify principal research trends in the literature and highlight active research topics that require more future work
G1.1. Identify the time distribution of primary studies
G1.2. Identify the venue distribution of primary studies
G1.3. Establish the geographic distribution of primary studies
G2.1 Identify the approaches for SPL implementation
G2.2. Identify the available tools that automate the SPL implementation
G2.3. Identify the variability models that are used in the SPL implementation
G2.4. Identify the software languages that are used to implement the SPL components
G2.5. Identify the type of evaluations that are used to verify or validate the SPL implementation studies
G3.1. Identify the main general topics based on the selected studies
G3.2. Identify the trend of these topics
Legend
Goal

Based on the previous diagram, we designed 10 answerable and interpretable research questions (SRQ). These research questions will be called SRQ to avoid confusion with the thesis research questions (RQ). Each SRQ is related to a specific secondary sub-goal which is simultaneously related to a primary sub-goal. Table 1-1 shows the 12 research questions and their division.

Table 1-1: Systematic mapping study research questions

Goal	**Sub-goal**	**Research Question**
G1	G1.1	**SRQ1:** What is the time distribution of primary studies? **Rationale:** answering this SRQ will help us to understand if this is a trending field and how it has evolved over the years.
	G1.2	**SRQ2:** What is the venue distribution of primary studies? **Rationale:** it is important to know what the authors preferred venues are in order to know where to find relevant papers about this field.
	G1.3	**SRQ3:** What is the geographic distribution of primary studies? **Rationale:** this SRQ will help us to understand what the lead countries and authors in this field are.
G2	G2.1	**SRQ4:** What approaches for SPL implementation are used? **Rationale:** commonly, the base of SPL implementation is a programming paradigm or mechanism which defines the design, implementation, and assembly of the reusable domain components. By answering this SRQ, we can get an overview of how researchers deal with SPL component implementation.
	G2.2	**SRQ5:** What are the available tools that automate SPL implementation? **Rationale:** SPL implementation is a difficult process that involves a lot of activities. The answer to this question will let us know what the available tools and programs are, what processes they automate, and how these tools are used. Based on this, researchers and companies can learn which tools can be used to satisfy their needs.
	G2.3	**SRQ6:** What variability models are most used in an SPL implementation? **Rationale:** This question specifies that system variability is a common field in SPL implementation. Answering this question will let us know what the most used variability models are.
	G2.4	**SRQ7:** What software languages are most used to implement the SPL components? **Rationale:** Commonly companies and developers have preferred software languages that they have used before to develop software products. Knowing the programming languages that are most used by authors and developers to implement SPL components will serve to discover what the matureness of the technologies and mechanisms using those languages are.
	G2.5	**SRQ8:** What type of evaluations are most used in studies on SPL implementation? **Rationale:** Knowing how the authors evaluate their proposals is useful for future studies.
G3	G3.1	**SRQ9:** What are the main topics of the selected studies? **Rationale:** by answering this SRQ, we can get an overview of what the main topics are in SPL implementation. This information is an

		important starting point for deepening researchers' topics of interest.
	G3.2	**SRQ10:** What trends have these topics followed over the last years? **Rationale:** trending topics are very valuable; they can lead researchers to focus on future research and new developments. Also, this answer helps to understand what topics authors have been developing recently and what challenges they have had.

1.2.2 Definition of search strategy

Keeping in mind the previous research questions, we defined a set of terms. These terms also consider three subjects: (i) the application domain, (ii) the SPL implementation stage, and (iii) the research perspective.

1. **Application domain:** contextualizes the search; in this case, the search only encompasses documents related to software product lines.
2. **SPL implementation stage:** as was mentioned before, the intention is to obtain a complete overview of SPL implementation. This process is comprised of different stages, and so relevant articles need to be found for each stage to obtain the complete overview. For this SMS, we selected 8 principal stages which range from specification to evolution (specification, modeling, binding, personalization, configuration, assembling, validation, and evolution).
3. **Research perspective:** the present SMS perspective is to obtain information about SPL implementation. Then, three terms were added that will help to refine and to obtain results related to this field.

These elements are consolidated in Table 1-2, and Table 1-3 lists the final derived search strings used to conduct the search. The strings are the result of the combination of three parts: (i) all terms in Table 1-2 – Group 1 (separated by "OR"); (ii) specific terms related to each SPL implementation stage, terms of Table 1-2 – Group 2 (separated by "OR"); and (iii) all terms in Table 1-2 – Group 3 (separated by "OR").

Table 1-2: Group of terms

Subject	**Term**	**Group**
Application domain	Product line, product family, SPL	1

SPL implementation stage	Specification: (domain engineering, domain requirements, requirements engineering) – Modeling: (variability language, domain design, variability model) – Binding: (feature binding, variability binding) – Personalization: (product personalization, market personalization, software personalization, component personalization) – Assembling: (product assembling, software assembling, component assembling) – Configuration: (product configuration, software configuration, component configuration, application realization, application implementation) – Validation: (product validation, quality assurance, software validation, product testing, component validation) – Evolution: (product evolution, software evolution, company evolution, component evolution)	2
Research perspective	Component implementation, software implementation, product implementation	3

Table 1-3: Resulting search strings

Search strings	**SPL Implementation Stage**	**No.**
(“product line” OR “product family” OR “SPL”) AND (“domain engineering” OR “domain requirements” OR “requirements engineering”) AND (“component implementation” OR “software implementation” OR “product implementation”)	Specification	DS1
(“product line” OR “product family” OR “SPL”) AND (“variability language” OR “domain design” OR “variability model”) AND (“component implementation” OR “software implementation” OR “product implementation”)	Modeling	DS2
(“product line” OR “product family” OR “SPL”) AND (“feature binding” OR “variability binding”) AND (“component implementation” OR “software implementation” OR “product implementation”)	Binding	DS3
(“product line” OR “product family” OR “SPL”) AND (“product personalization” OR “market personalization” OR “software personalization” OR “component personalization”) AND (“component implementation” OR “software implementation” OR “product implementation”)	Personalization	DS4
(“product line” OR “product family” OR “SPL”) AND (“product assembling” OR “software assembling” OR “component assembling”) AND (“software implementation” OR “product implementation” OR “component implementation”)	Assembling	DS5
(“product line” OR “product family” OR “SPL”) AND (“product configuration” OR “software configuration” OR “component configuration” OR “application realization” OR “application implementation”) AND (“component implementation” OR “software implementation” OR “product implementation”)	Configuration	DS6

(“product line” OR “product family” OR “SPL”) AND (“product validation” OR “quality assurance” OR “software validation” OR “product testing” OR “component validation”) AND (“component implementation” OR “software implementation” OR “product implementation”)	Validation	DS7
(“product line” OR “product family” OR “SPL”) AND (“product evolution” OR “software evolution” OR “company evolution” OR “component evolution”) AND (“component implementation” OR “software implementation” OR “product implementation”)	Evolution	DS8

In addition to the search strings, we established the search sources used to find the primary studies which are shown in Table 1-4. According to Dyba *et al.* (2007), these databases are efficient for conducting systematic studies in the context of software engineering. Furthermore, these databases have also been considered in another SMS (Laguna & Crespo, 2013). After a first consolidation of the results, other databases were considered as part of a second phase (Google Scholar and Citeseerx) to try to find additional results that could offer useful material.

For each database, we applied a “trial search”. This trial search consisted of introducing the first derived search string (DS1) into each database search form, and we checked if the results were as expected. If less than 10 documents were returned or if there were millions of results with inconsistent articles (not related to SPL implementation), then, those databases were discarded (DB6, DB7, and DB8 were discarded). Finally, we introduced each resulting search string into each selected search source and collected the results.

Table 1-4: Selected search sources

#	**URL**	**Source**	**Selected**
DB1	http://dl.acm.org/	ACM DL	Yes
DB2	http://ieeexplore.ieee.org/	IEEE Explore	Yes
DB3	http://www.sciencedirect.com/	ScienceDirect	Yes
DB4	http://www.springer.com/la/	Springer	Yes
DB5	https://scholar.google.com	Google Scholar	Yes
DB6	http://www.scopus.com	Scopus	No (after trial search)
DB7	http://citeseerx.ist.psu.edu	Citeseerx	No (after trial search)
DB8	http://www.isiknowledge.com	Web of Science	No (after trial search)

Inclusion/exclusion criteria

Before conducting the search, the following restrictions and quality criteria for including/excluding publications were defined. These criteria were developed with the intention of finding the most relevant papers to solve the research questions and to exclude papers which do not fit this field and do not allow the research questions to be solved.

- **Restriction R1:** The study only includes papers available in electronic form. Books were analyzed based on information available online and using the hard copy versions.
- **Restriction R2:** Only publications written in English were included.
- **Restriction R3:** Articles related to the topic of this paper published between 1st January 2000 and 31st March 2017 were included.
- **Quality criterion Q1:** Each publication was checked for completeness. Publications containing several unsupported claims or that frequently referred to existing work without providing citations were excluded.
- **Quality criterion Q2:** Works by the same authors with very similar content were included and grouped under the same category (method).

1.2.3 Definition of search conduction

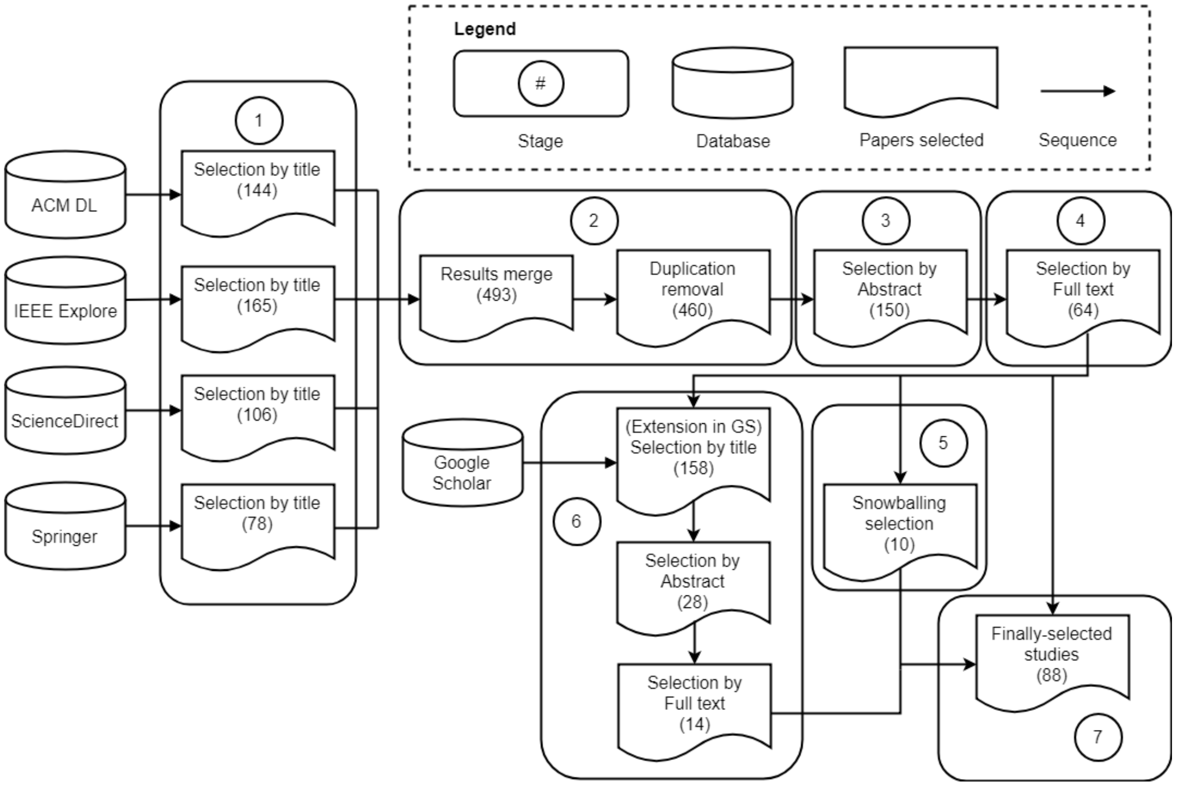
The search conduction was composed of seven stages based on the study by Li *et al.* (2015) work: (i) selection by title, (ii) first results merge, (iii) selection by abstract, (iv) selection by full text, (v) snowballing, (vi) search extension in Google Scholar, and (vii) final results merge. These stages (see Figure 1-3) used the previous search strings and criteria, and they are presented below:

1. **Selection by title:** the search conduction started by using the search strings in four search sources (ACM DL, IEEE Explore, ScienceDirect and Springer), then candidate studies were selected based on the title. Restriction R1, R2, and R3 were applied in this step.
2. **First results merge:** all candidate studies were merged (493 at this point) and duplicated studies were removed (33 studies were removed).
3. **Selection by abstract:** the next stage analyzed the candidate studies' abstracts to guarantee that they were related with the desired topic (SPL implementation); at this point, 150 candidate studies were selected.
4. **Selection by full text:** the previous studies' full texts were analyzed and as a result 64 studies were selected. Quality criteria Q1 and Q2 were applied in this step.
5. **Snowballing:** in order not to miss any potentially relevant studies, we applied the "snowballing" technique to find more connected studies by checking the references

of the selected studies. This process could be iterative as snowballing and could be repeated in the newfound studies. However, only the first iteration was applied, and 10 new studies were found.

6. **Search extension in Google Scholar:** parallel to stage 5, we extended the search by looking in Google Scholar, we used the search strings and made a first scan by title (Restriction R1, R2, and R3 were applied in this step), then, a second scan by abstract, and finally a third scan by full text (Quality criteria Q1 and Q2 were applied in this step). 14 new studies were found.
7. **Final results merge:** at the end, we merged the selected studies from stages 4 and 5 and 6, and 88 relevant studies were selected. These relevant studies are listed in Appendix B.

Figure 1-3: Study selection stages


Legend
#
Stage
Database
Papers selected
Sequence
1
ACM DL
IEEE Explore
ScienceDirect
Springer
Selection by title (144)
Selection by title (165)
Selection by title (106)
Selection by title (78)
2
Results merge (493)
Duplication removal (460)
3
Selection by Abstract (150)
4
Selection by Full text (64)
Google Scholar
6
(Extension in GS) Selection by title (158)
Selection by Abstract (28)
Selection by Full text (14)
5
Snowballing selection (10)
Finally-selected studies (88)
7

1.3 Data extraction

The data extraction process consisted of collecting key information about relevant studies that will be the base to answer the research questions. We created a spreadsheet and for each study we recorded 18 pieces of data (see Table 1-5), the description of each data point is also included in the next table (this description explains in detail the kind of data to be collected). Lastly, relevant research questions were assigned to each data point; this is because some of this data serves to answer one or multiple research questions.

Table 1-5: Studies data extraction

#	Data item name	Description	Relevant SRQ
D1	Article name	The name of the article	None
D2	Article year	The year in which the article was published	SRQ1
D3	Type of publication	The publication type, such as journal, conference, workshop, symposium, or book chapter	SRQ2
D4	Publication name	The name of the journal, conference, workshop, symposium or book chapter where the article was published	SRQ2
D5	Publication venue	The name of the publication venue of the study	SRQ2
D6	Authors	The name of all authors that participate in the study	SRQ3
D7	Country	The main author's affiliation country	SRQ3
D9	Implementation approach	The programming paradigm or approach used to design, implement, and assemble the SPL components	SRQ4
D10	Implementation stage	The SPL implementation stage that was covered in the article	SRQ5
D11	Tools	The name of the tools presented or used in the article	SRQ5
D12	IDE	The integrated developed environments presented or used in the article	SRQ5
D13	Variability models	The variability models presented or used in the article	SRQ6
D14	Provide example with programming code	Yes or no depending on if the article provides an example with programming code	SRQ7
D15	Software languages	The software languages presented or used in the article to implement the SPL components	SRQ7
D16	Evaluation type	The type of evaluation presented or used in the article	SRQ8
D17	Example type	The type of example presented in the article	SRQ8
D18	Topics and Trends	A general topic discussed in the article and/or trends	SRQ9-SRQ10

1.4 Resolving questions about research goal G1 – Publication

The first analysis after the search conduction gives us a global overview of the time distribution and the diversity of the sources in this research field (which provides information to answer SRQ1 and SRQ2). Implementation of SPL has been a relevant topic over the last decade (see Figure 1-4); at the beginning of the twenty-first century there were a few studies in this field, and in 2016 publication peaked with 10 studies published. 2017 shows only 4 publications, but the time-frame between the papers are sent to be published and their final publication should be considered, as the search conduction was carried out between March and April of 2017.

Figure 1-4: Temporal distribution of the sources

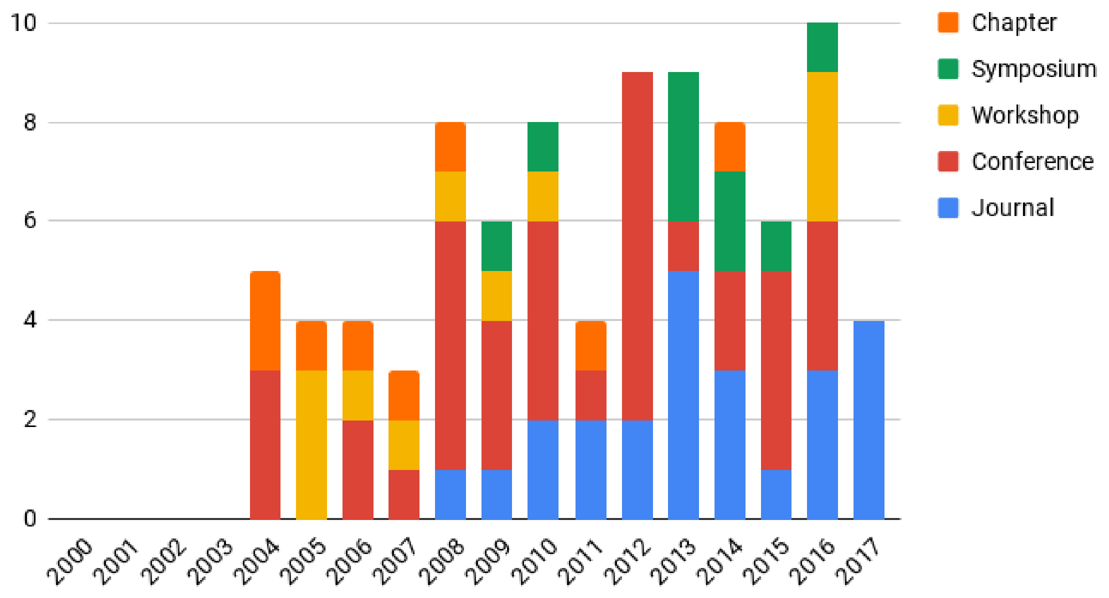
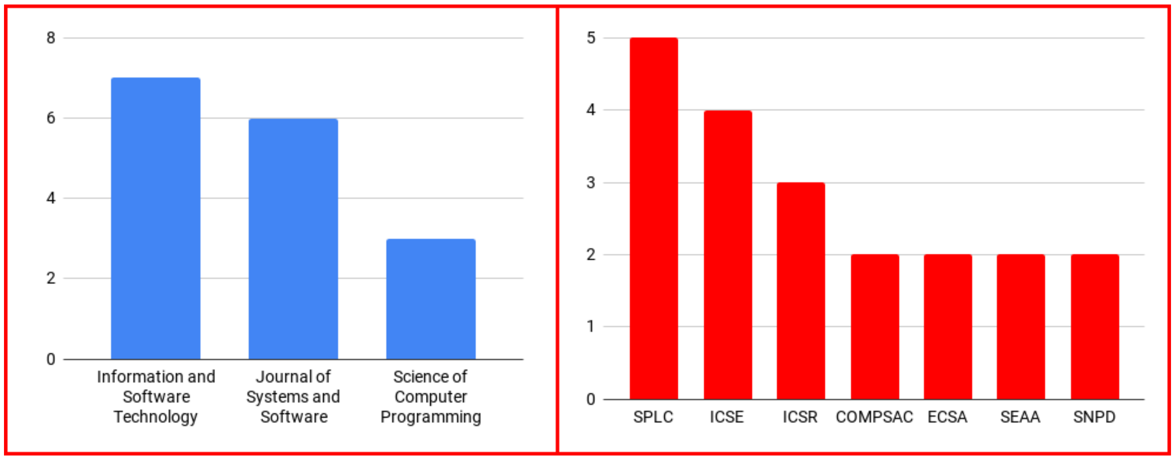


Figure 1-4 shows some interesting information. Most of the publications were presented at conferences or similar (symposiums and workshops) with a total of 56 publications (64%). The second preferred medium is journals with 24 publications (27%) and finally book chapters with 8 publications (9%).

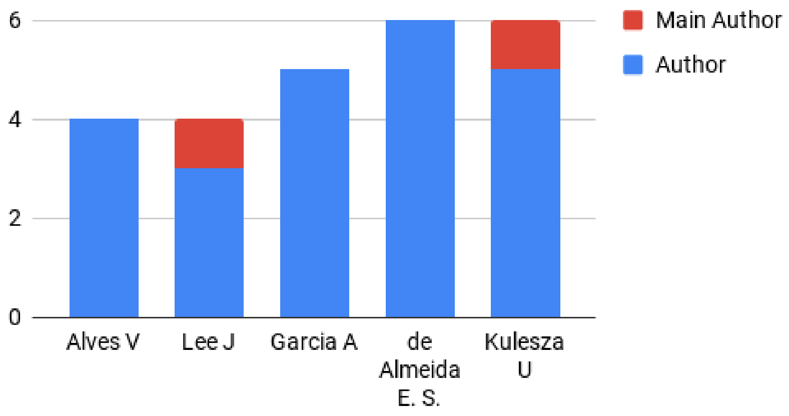
Another important result is related to conferences and journals that have published the most studies (see Figure 1-5). For journals, *Information and Software Technology* with 7 publications was the most used; second, *Journal of Systems and Software* with 6 and third, *Science of Computer Programming* with 3. For conferences, International Software Product Line Conference (SPLC) with 5 publications was the most used; second, International Conference on Software Engineering (ICSE) with 4 and third, International Conference on Software Reuse (ICSR) with 3 publications.

Figure 1-5: Data on most frequent journals and conferences



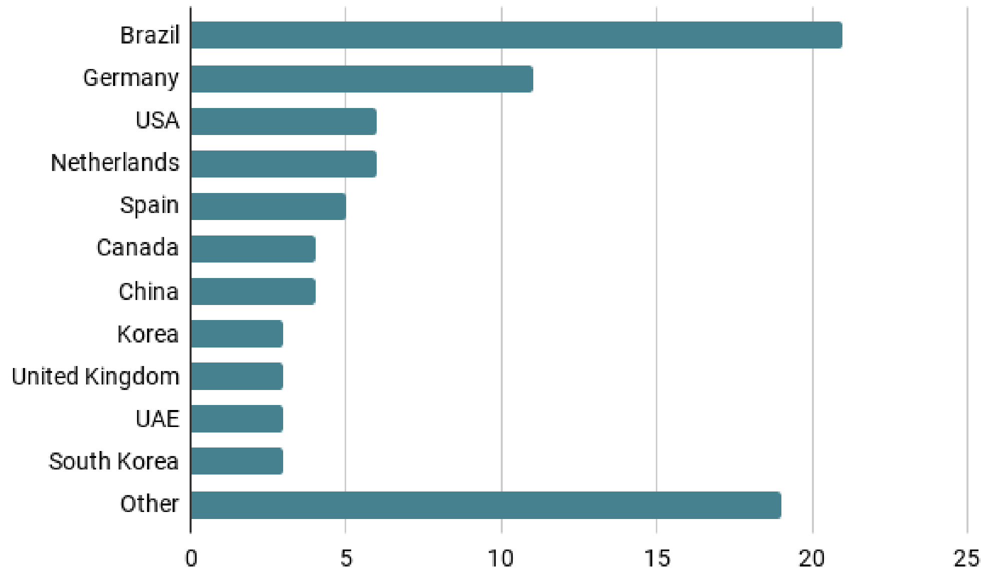
Top authors in the research field are presented in Figure 1-6, showing the number of contributions from each author as a main author or secondary author. Uirá Kulesza (Kuleza U) from Universidad Federal de Río Grande del Norte (Brazil) and Eduardo Santana de Almeida (de Almeida E.S.) from Universidade Federal da Bahia (Brazil) are the top contributors with 6 publications each, followed by Alessandro Garcia (Garcia A) from Pontificia Universidad Católica de Río de Janeiro (Brazil) with 5 publications. Finally, Jaejoon Lee (Lee J) from Lancaster University (UK) and Vander Alves (Alves V) from Universidade de Brasília (Brazil) present 4 publications each (this analysis helps to answer the SRQ3).

Figure 1-6: Data on most frequent authors'



Based on the previous figure, Brazil leads as the country with the most top contributors. This is consistent with the results presented in Figure 1-7. This figure provides a list of the top countries that each publication's main authors are affiliated to (SRQ4 research question). It shows Brazil also leads with 21 publications (24%), followed by Germany with 11 publications (13%), and USA and Netherlands with 6 publications each.

Figure 1-7: Data on each publication's main authors' affiliations



1.5 Resolving questions about research goal G2 – SPL implementation

After the search conduction was carried out, the second analysis focused on the main topic of the SMS (SPL implementation), which provides information for answering SRQ4 to SRQ8. First, we summarize the results of the SPL implementation approaches; second, we present the results in order to answer the research questions.

1.5.1 Summary of SPL implementation approaches

We divided the studies based on their SPL implementation approach (See Table 1-5 – D9). We grouped these studies as follows: (i) aspect-oriented programming (AOP), (ii) service-oriented architecture (SOA), (iii) annotative approaches, (iv) feature-oriented programming (FOP) and delta-oriented programming (DOP), (v) other approaches, (vi) mixed approaches, (vii) not specified. After summarizing each approach, we will show an example of component implementation with some of the approaches listed above. These approaches also describe the way in which the SPL components must be implemented.

Below, the results of the information collected for each of the previous groups are presented. To organize the information, we created a table structure that summarizes the collected information (based on Table 1-5). This structure records:

- **SPLE stage** refers to the software product line engineering stage, that was covered in the study such as modeling, binding, implementation, and evolution. Sometimes a study can cover multiple SPLE stages, in these cases, we recorded the information as “Multiple”.
- **Evaluation** refers to the type of evaluation applied in the study such as running examples, experiments, and comparisons.
- **Specific example** refers to the type of example presented in the study.
- **Tool support** refers to the tools, libraries or software used or mentioned in the study.
- **Software language** refers to the software languages used or mentioned in the study such as Java, PHP, and C++.
- **Variability model** refers to the variability model used or mentioned in the study such as the feature model, OVM, and goals model.

1.5.1.1 Aspect-oriented programming (AOP)

Aspect-oriented programming (AOP) is an approach that aims to modularize the crosscutting concerns of both software product lines and single-systems. These concerns are widely-scoped properties and usually crosscut several modules in the software system. Aspects are the abstractions used to encapsulate otherwise crosscutting concerns. An example of a crosscutting concern is "logging", which is frequently used in distributed applications to aid debugging by tracing method calls [S34]. Therefore, the use of aspects relies on three major mechanisms to modularize and vary crosscutting concerns: (i) join points are the identifiable execution points, for example, method calls or object attribute assignments; (ii) a pointcut is a predicate over dynamic join points, meaning that given a certain dynamic join point, a pointcut can either match this join point or not (at runtime); and (iii) an advice is a new behavior that extends, refines or replaces the computation at selected join points [S35]. AspectJ is an example of an aspect-oriented programming language; in AspectJ, a call to a method, a method execution, or an assignment to a variable are examples of join points. For instance, using AspectJ, it is possible to intercept a call to an interface method in order to check whether any of the parameters are null [S35].

A summary of research focusing on SPL and AOP is presented in Table 1-6. This table details important descriptions about the main papers in this field, the SPLE stage covered, the type of evaluation, the kind of examples presented, and the tools, the software languages and the variability model used.

Table 1-6: Summary of SPL and AOP

Ref	SPLE stage covered	Evaluation	Specific Example	Tool Support	Soft. lang.	Variability model
S15	Architecture	Running example	Public health complaint system	COSMOS∗-VP		Feature model
S18	Evaluation	Running example	Mobile phone company	AspectJ - Ant	Java	
S23	Modeling	Running example	Microwave oven	Uced		Feature model
S24	SPL architecture - Multiple	Running example	Graph algorithms	AspectJ - Extended GenVoca - CoBaCoLa - ReGaL	Java	
S25	Multiple	Running example	Scientific calculator	AspectJ	Java	Feature model
S41	Architecture - Evolution	Running example	Philips TV product line architecture	Koala - INXS - AspectC++ - Aspicere - C4 - WeaveC - AspectC		
S42	Evolution	Running example	BestLap, Mobile Media	AspectJ - CaesarJ	Java	Feature model
S44	Customization - Implementation	Running example	Library management domain	AspectJ - OntoFeature	Java	Feature model
S46	Product derivation			AspectJ - pure::variants - Gears - EMF	Java	Feature model
S59	Architecture	Running example	Library management system	AspectJ		Feature model
S60	Multiple	Running example	Game	AspectJ - Junit - EMF - JET - Gears - Pure::variants - Feature modeling plugin (FMP)	Java	Feature model

S66	Architecture	Running example - Comparison	Terrestrial Digital TV System	AspectualACME - ACME		Feature model
S67	Implementation - Multiple	Running example	Weather station - Home automation system	AspectJ - CaesarJ - OSGi - CAM-DAOP - Ecore - openArchitectureWare	Java	Feature model
S70	Modeling - Implementation	Running example	Microwave system	AspectJ	Java - XML	
S85	Multiple	Running example - Comparison	Bus transportation	AspectJ - Spring framework - Hibernate	Java - XML	Feature model

1.5.1.2 Service-oriented architecture (SOA)

SOA emphasizes building software solution logic in the form of self-contained services that can be reused in multiple systems [S6]. In this approach, the SPL components are implemented in the form of services. An SOA implementation exposes standard interfaces to make services available for authorized service consumers to use in a variety of ways [S16], therefore, services can be replaced or can be reconfigured to adapt to different circumstances [S62]. In addition, an SPL that implements an SOA approach commonly uses Business Process Execution Language (BPEL) or similar languages to support variability. BPEL is an XML-based programming language that can be used to describe the interaction between web services at the message level; in this way, it also describes their composition. One example of a BPEL extension is a language called VxBPEL, which has extra XML elements to support variation points and variants in a BPEL process [S62].

SPL and SOA integration has been widely studied over recent years. A summary of the principal research is presented in Table 1-7. This table contains important descriptions about the main papers in this field, the SPLE stage covered, the type of evaluation, the kind of examples presented, and the tools, the software languages and the variability model used.

Table 1-7: Summary of SPL and SOA

Ref	SPLE stage covered	Evaluation	Specific Example	Tool Support	Software language	Variability model
S4	Multiple	Running example	Mobile learning app		Java - Javascript	Feature model
S5	Implementation	Running example	Office system	JBoss jBPM	Java	Feature model
S6	Multiple	Running example	e-health domain	SoaSPL		Feature model

S8	Architecture - Multiple	Running example - Experiment	Currency exchange	VisualWebC - BPEL4WS	ASP - Java	
S14	Multiple					
S16	Implementation - Evaluation	Comparison	Library	OSGi - Apache Tuscany - JAX-WS	Java	Feature model
S17	Multiple	Running example	e-commerce	Apache ODE - Apache CXF - Eclipse Swordfish - SoaSPLE	Java	Feature model - Multiple View Service Variability Model
S26	Implementation - Product derivation	Industrial report	Aurora – web development enviroment		JSP - HTML - XML - Java	
S62	Modeling - Multiple	Running example	Supply Chain Management System	COVAMOF-VS - BPEL - VxBPEL - ArgoUML	Java - XML	UML profile for architectural variability modeling
S68	Multiple					
S76	Modeling - Implementation	Running example	Online marketplace	BPMN - EPC - YAWL - VxBPEL - COVAMOF - SOMA	Java - .net	Feature model

1.5.1.3 Annotative approaches

Annotative approaches implement SPL components with some form of explicit or implicit annotations, with the prototypical example being the use of *#ifdef* and *#endif* statements to surround the SPL component code. Annotative approaches assemble the variations of all possible configurations within a single artifact, as is the case with the C++ preprocessor and the Java preprocessor Antenna [S48]. During variant derivation, the parts that are not needed within a variant are removed. This is the reason that annotative approaches are well known in their support of fine-grained extensions on statements, parameters, and conditional expressions [S65].

A summary of the principal research on SPL and annotative approaches is presented in Table 1-8. This table contains an important description on the major papers in this field, the SPLE stage covered, the type of evaluation, the kind of examples presented, and the tools, the software languages and the variability model used.

Table 1-8: Summary of SPL and annotative approaches

Ref	SPLE stage covered	Evaluation	Specific Example	Tool Support	Software language	Variability model
S2	Multiple	Running example	Satellite system	pure::variants - Rhapsody - C Compiler	C	Feature model
S20	Architecture - Multiple	Running example	Chat app	xADL - ANTLR	Java	
S36	Multiple	Running example	Mobile games		Java	Feature model
S52	Multiple	Running example	Shopping store	GenArch	Java - XML	Feature model
S61	Configuration - Multiple	Running example	Software configuration management	FeaturePlugin - Fujaba - MODPLFeaturePlugin	Java	Feature model
S72	Requirements - Architecture - Multiple	Industrial report	Automotive	pure::variants - DOORS - Rhapsody		Feature model
S84	Architecture - Multiple	Running example	Chat app	xLineMapper - Eclipse JET - ANTLR - ArchJava - Archface	Java	PLA model

1.5.1.4 Feature-oriented programming (FOP) and delta-oriented programming (DOP)

Feature-oriented programming (FOP) has been used to implement SPLs by composing feature modules. To obtain a product for a feature configuration, feature modules are composed incrementally. In the context of OOP, feature modules can introduce new classes or refine existing ones by adding fields and methods or by overriding existing methods [S37]. Delta-oriented programming (DOP) has been seen as an extension of FOP. In DOP, the implementation of an SPL is divided into a core module and a set of delta modules. The core module comprises a set of classes that implement a complete product for a valid feature configuration. This allows the core module to be developed with well-established single application engineering techniques to ensure its quality. Delta modules specify the changes to be applied to the core module in order to implement other products. A delta module can add or remove classes from product implementation [S37].

A summary of the principal research on SPL, FOP, and DOP is presented in Table 1-9. This table contains an important description of the major papers in this field, the SPLE stage covered, the type of evaluation, the kind of examples presented, and the tools, the software languages and the variability model used. References marked with an asterisk (*) use DOP approaches.

Table 1-9: Summary of SPL, FOP, and DOP

Ref	SPLE stage covered	Evaluation	Specific Example	Tool Support	Software language	Variability model
S27*	Modeling - Multiple	Running example	Bank system	ABS tool - ANTLR - JastAdd - Papyrus	Java - XML	Feature model
S32	Multiple	Running example	Graph spl	FeatureC++	C++	Feature model
S39	Multiple	Running example	Graph spl	rbFeatures - AHEAD - CIDE	Ruby	Feature model
S47	Multiple	Running example	Graph spl – calculator – Expression product line	rbFeatures - FeatureJ	Ruby	Feature model
S56	Multiple	Running example	Map	AHEAD - XAK	JavaScript - XML - SVG	Feature model
S75*	Multiple	Running example	Smart home	SiPL - EMF - SiLift - Simulink		Feature model

1.5.1.5 Other approaches

Peña [S31] discussed the use of agents in SPL. Agent-oriented software engineering is a software engineering paradigm that promises to enable the development of more complex systems than those that can be achieved with current object-oriented approaches using agents and organizations of agents as the main abstractions. A software agent is a piece of software which exhibits the following characteristics: autonomy, reactivity, pro-activity and social ability. The introduction of agents to the industrial world may benefit from the advantages that SPL offers [S31]. Using SPL philosophy, a company will be able to define a core multi-agent system from which concrete products will be derived for each customer.

El-Sharkawy *et al.* [S43] developed a tool called EASy-Producer, an Eclipse extension for efficient software product line development. This tool uses three custom-developed domain-specific languages (DSL): IVML language which is used to define the variability model of the SPL, VIL language which is used to define the relationship between the variabilities and the implementation, and VTL language which supports variability-aware artifact generation.

Another approach that the SMS found is context-oriented programming (COP). COP is an approach that supports the dynamic adaptation of context conditions such as bandwidth availability, presence of WiFi and data connection (Salvaneschi *et al.*, 2012). COP introduces language-level abstractions, such as “layers” that group partial method definitions, to manage the modularization of adaptations and their dynamic activation during the program's execution.

A summary of other SPL implementation approaches is presented in Table 1-10. This table contains an important description of the major papers in this field, the approach used, the SPLE stage covered, the type of evaluation, the kind of examples presented, and the tools, the software languages and the variability model used.

Table 1-10: Summary of other SPL implementation approaches

Ref	Approach	SPLE stage covered	Evaluation	Specific Example	Tool Support	Soft. Lang.	Var. Model
S31	Agents	Multiple	Running example	Security council's procedure to issue resolutions			Feature model - Goals
S43	IVML, VIL, VTL (custom DSLs)	Multiple	Running example	Elevator simulator	EASy-Producer - Dopler tool - FeatureIDE - pure::variant - FaMa framework - REMiDEMMI - FeatureMapper - AspectJ	Java	IVML

1.5.1.6 Mixed approaches

Sometimes, authors develop studies with mixed SPL implementation approaches. Most of these studies compare two or more approaches [S34, S37, S65, S81]. However, in other cases, they try to take advantage of the benefits of some approaches and use them in combination with other approaches. For example, Parra *et al.* [S19] develop an approach for SPL based on SOA services. Their approach mixes the use of SOA services with annotations. The use of annotations serves to manually indicate which parts of the original artifacts can be transformed into services to be used externally. Andrade *et al.* [S22] propose the use of AOP with annotations. In their work, they use AspectJ as the base tool, which is refined with the use of some Java annotations. They implemented this combination to solve some issues presented in previous work which uses AspectJ-based idioms. Santos *et al.* [S73] present RiPLE-HC, a strategy aimed at blending compositional and annotative approaches to implement variability in JavaScript-based systems. In the annotative part, they use the classical *#ifdef* and *#endif* statements to support the fine-grained extensions. Other studies include the blending of compositional and annotative approaches (Kästner & Apel, 2008; Walkingshaw & Erwig, 2012; Behringer & Rothkugel, 2016; Horcas *et al.*, 2018). However, all the previous approaches present important limitations such as limited support for a few software languages, the use of if statements or tree structures, poorly detailed coding, and no tool support.

A summary of SPL with mixed implementation strategies is presented in Table 1-11. It includes studies that deal with comparative methods. The table contains an important description of the major papers in this field, the mixed approaches used, the SPLE stage covered, the type of evaluation, the kind of examples presented, and the tools, the software languages and the variability model used.

Table 1-11: Summary of SPL with mixed approaches

Ref	Approaches	SPLE stage covered	Evaluation	Specific example	Tool support	Soft. Lang.	Var. Model
S19	SOA - annotative	Multiple	Running example	Multiple artifacts	Spoon - FeatureIDE	Java	Feature model
S22	AOP - Annotative	Implementation - Binding	Experiment	Company spl	AspectJ	Java	
S29	Agents - AOP	Multiple	Running example	Personal user services	GenArch - FMP - Jadex	Java - XML	Feature model
S34	AOP - OO	Evolution	Comparison	Mobile media			Feature model
S35	AOP - OO - Component-based	Architecture - Multiple	Running example	Mobile media	AspectJ	Java	Feature model
S37	DOP - FOP	Multiple	Comparison	Expression product line - Graph	DeltaJ - Jak - AHEAD	Java	Feature model
S48	Annotative - FOP - DOP	Multiple	Running example	FeatureAMP – GameOfLife - Violet	EMF - Xtext - DeltaJ - Antenna - C++ preprocessor	Java – C++	Feature model
S53	AOP - FOP	Multiple	Running example - Comparison	Three board games	AspectJ - CaesarJ	Java	Feature model
S63	OO - AOP	Implementation	Running example - Comparison	Smart homes		Java	Feature model
S65	FOP - OO - Annotative	Evolution - Multiple	Running example - Comparison	WebStore - Mobile Media	AHEAD - JAK	Java - JSP	Feature model

S73	Annotative - compositional	Multiple	Running example - Experiment	Learning objects	RiPLE-HC - npm - jam - bower - requireJS - FeatureHouse - FeatureIDE	JavaScript	Feature model
S78	SOA - annotative	Multiple	Running example	Company apps	SPLIT - Spoon - EMF - FeatureIDE	Java - XML	Feature model
S79	SOA - annotative	Multiple	Running example	Smart home	LISA	Java	Feature model
S81	AOP - OO	Modeling	Running example - Comparison	Pacemaker product line	Rhapsody		

1.5.1.7 SPL implementation approach not specified

There were 23 studies in which the SPL implementation approach that was used was not specified. For these studies, the same information shown in the above tables was recorded. This information can be found online (Correa, 2018).

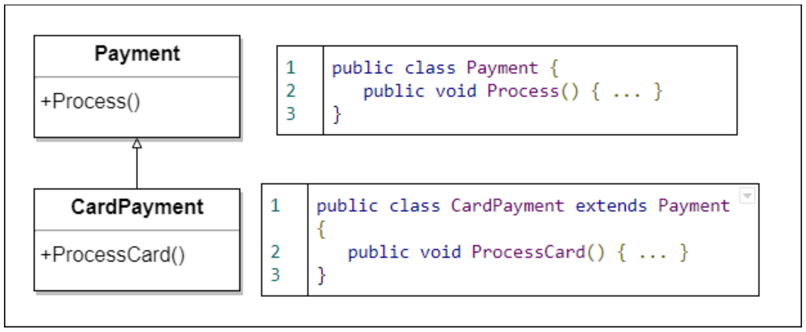
1.5.1.8 Example of component implementation with some current approaches

The previous section explained and summarized how different approaches deal with SPL implementation from a theoretical perspective. In order to illustrate how some of those approaches can be used to design and implement SPL components, a practical example of an appliance stores product line was elaborated. In this example, the appliance stores must have the possibility to manage “Payments” (mandatory feature), and those payments can be extended to support “Card Payments” (optional feature). Based on these features, we decided to implement the Payment and Card Payment using six different SPL implementation approaches (OOP, FOP, DOP, COP, AOP, annotative, and SOA), as shown in Figures 1-8, 1-9, 1-10, 1-11, 1-12, 1-13, and 1-14. The implemented methods were left blank because the intention is to analyze how the components and their code are divided based on each approach, rather than analyzing the method that was implemented.

OOP

Figure 1-8 shows an excerpt of our appliance stores product line represented as a class hierarchy. In OOP, components are implemented with classes. In this example, the “Payment” feature is implemented through the Payment Java class, and the “Card Payment” requirement is implemented through the CardPayment Java class. Extension of components is an important contribution of OOP to improve reuse by inheritance. There are other approaches that include the use of interfaces or patterns, but all of them are based in object-oriented elements, and not all SPL components are object-oriented.

Figure 1-8: Example of a domain component implementation in OOP



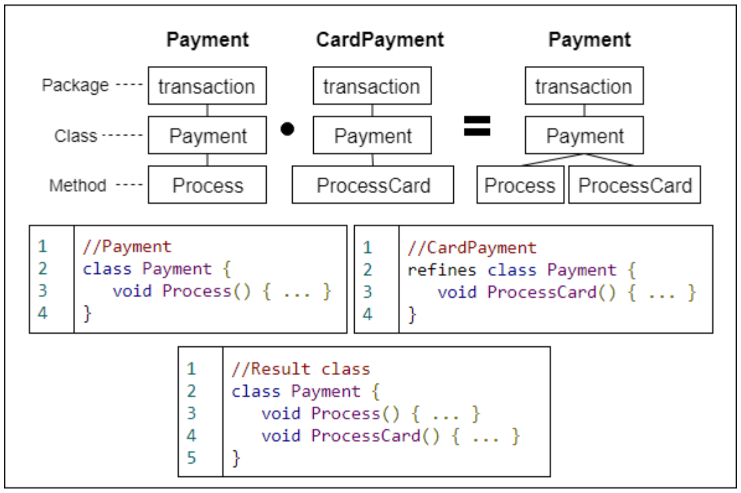
```
public class Payment {
   public void Process() { ... }
}
```

```
public class CardPayment extends Payment
{
   public void ProcessCard() { ... }
}
```

FOP

Figure 1-9 shows two different options for the implementation of components with FOP. The first option is based on “Superimposition”, which is the process of composing software artifacts by merging their corresponding substructures (cf. Fig. 1-9 model). The second option is based on refactoring, which has been proposed as a means for improving the internal structure of a system (*i.e.*, the source code) while preserving the external behavior. In this case, a feature module (component) may also contain class and method refinements (cf. Fig. 1-9 code). The example shows how the CardPayment feature code refines the Payment feature with the addition of the new processCard method. This refinement is carried out in the component integration process. The disadvantage of FOP is that it doesn’t support fine-grained extensions.

Figure 1-9: Example of a domain component implementation in FOP



```
//Payment
class Payment {
   void Process() { ... }
}
```

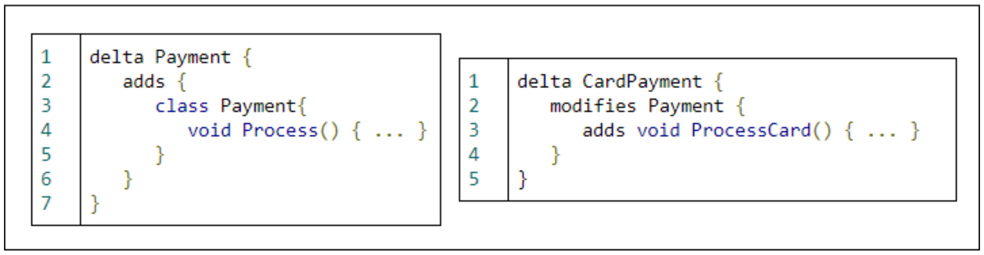
```
//CardPayment
refines class Payment {
   void ProcessCard() { ... }
}
```

```
//Result class
class Payment {
   void Process() { ... }
   void ProcessCard() { ... }
}
```

DOP

Figure 1-10 shows how to implement two components by means of DOP with DeltaJ. Similar to FOP, CardPayment delta modifies the Payment class with the addition of a new method called processCard. Delta also allows removing code which is not allowed in FOP. Finally, similar to FOP, DOP doesn't provide fine-grained extensions.

Figure 1-10: Example of a domain component implementation in DOP



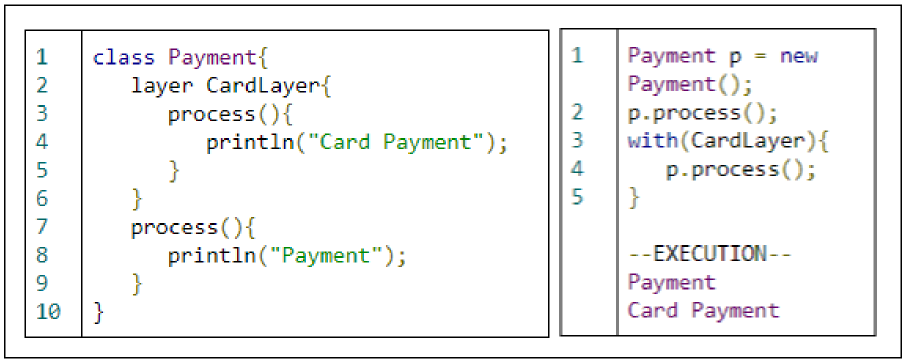
```
delta Payment {
   adds {
      class Payment{
         void Process() { ... }
      }
   }
}
```

```
delta CardPayment {
   modifies Payment {
      adds void ProcessCard() { ... }
   }
}
```

COP

Figure 1-11 shows the component implementation by means of COP. In our example, the Payment class is extended with a CardLayer (layers are used to group partial method definitions), this layer defines a process method, which is only executed when the CardLayer is referred. However, the layer activation mechanism can be quite complex and could tend to make source code complex and difficult to maintain.

Figure 1-11: Example of a domain component implementation in COP



```
class Payment{
   layer CardLayer{
      process(){
         println("Card Payment");
      }
   }
   process(){
      println("Payment");
   }
}
```

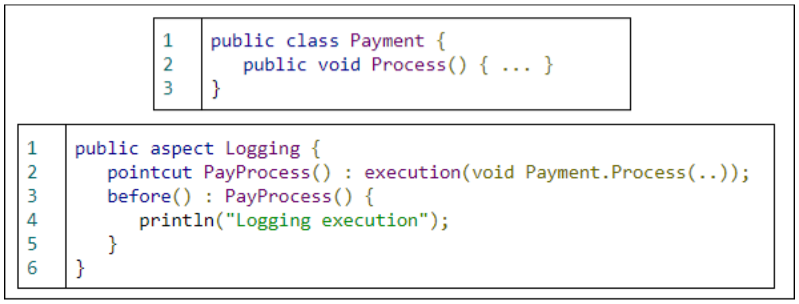
```
Payment p = new
Payment();
p.process();
with(CardLayer){
   p.process();
}

--EXECUTION--
Payment
Card Payment
```

AOP

Figure 1-12 shows the component implementation by means of AOP. AOP is a solution for crosscutting concerns, so instead of implementing the CardPayment feature, we implemented a Logging feature which is a crosscutting concern. There, we defined a Logging aspect with a pointcut and advice. The pointcut was called PayProcess which was linked to the Payment Process method. The advice was linked to the previous pointcut and it was defined to be executed before the pointcut execution. This means, that before the Payment Process method is executed, the Logging aspect will execute its advice. The disadvantage of AOP is that it only supports variability for crosscutting concerns.

Figure 1-12: Example of a domain component implementation in AOP



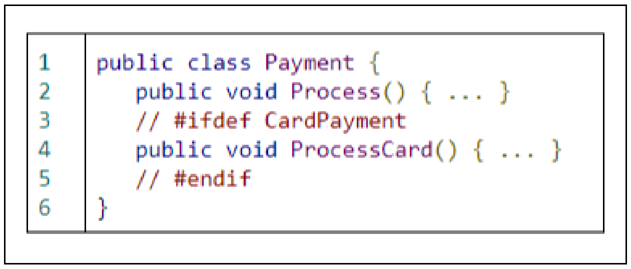
```
public class Payment {
   public void Process() { ... }
}
```

```
public aspect Logging {
   pointcut PayProcess() : execution(void Payment.Process(..));
   before() : PayProcess() {
      println("Logging execution");
   }
}
```

Annotative

Figure 1-13 shows how to implement SPL components by means of Antenna (an annotative approach). In this case, the optional feature CardPayment is surrounded by Java comments. During the product derivation, the parts that are not needed within a file are removed. The disadvantage of annotative approaches is that the SPL components contain all the possible variations inside them, which makes them difficult to maintain and evolve.

Figure 1-13: Example of a domain component implementation in an Annotative approach



```
public class Payment {
   public void Process() { ... }
   // #ifdef CardPayment
   public void ProcessCard() { ... }
   // #endif
}
```

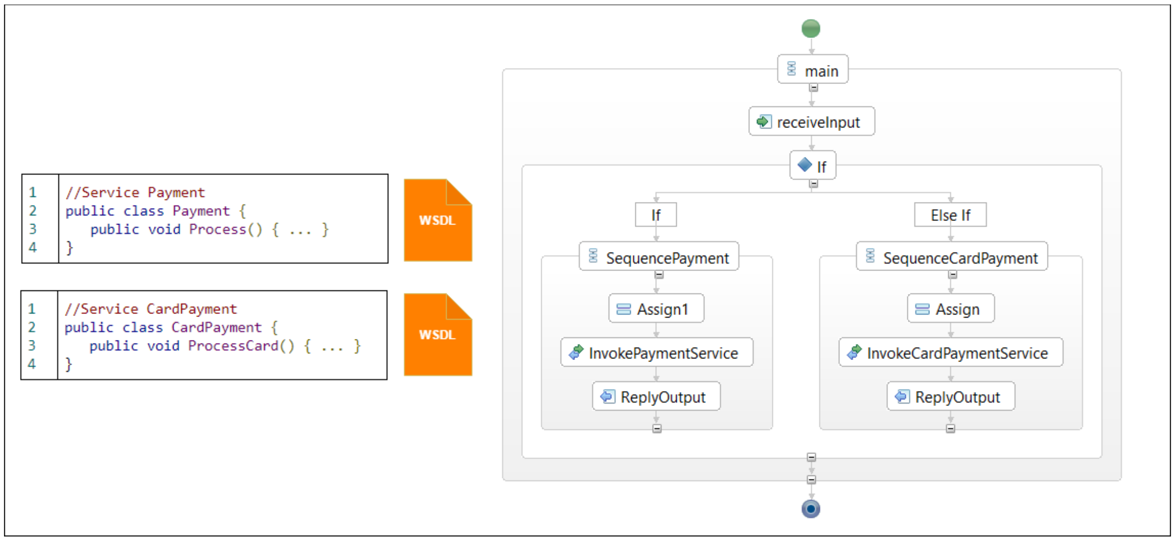
SOA

Figure 1-14 shows how to implement SPL components by means of SOA and with the use of BPEL Designer[1]. In this case, Payment and CardPayment are implemented in form of independent services, with the use of Web Services Description Language (WSDL) files. In addition, a BPEL process file is created to describe the interaction between these services

[1] https://www.eclipse.org/bpel/

(to describe their composition). Depending on the received input, the Payment WSDL or the CardPayment WSDL could be invoked. However, SOA is not recommended for standalone applications, and the GUI has to be designed as an independent project.

Figure 1-14: Example of a domain component implementation in SOA



```
//Service Payment
public class Payment {
   public void Process() { ... }
}
```

```
//Service CardPayment
public class CardPayment {
   public void ProcessCard() { ... }
}
```

1.5.2 Results

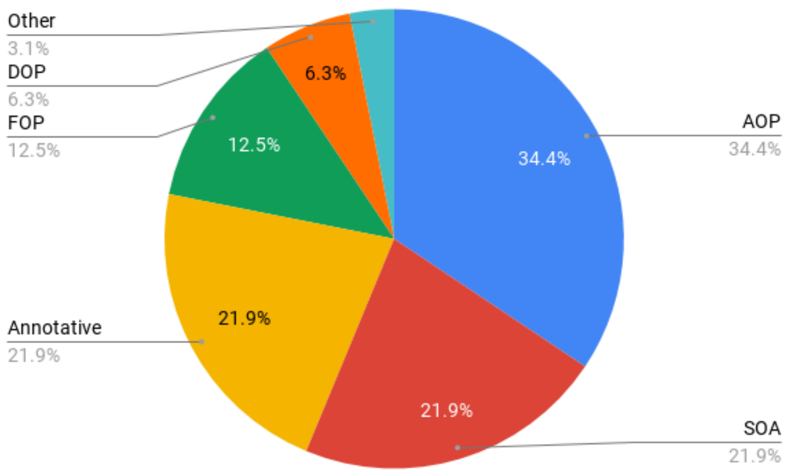
Following, we show the results for answering SRQ4 to SRQ8.

SRQ4: What approaches to implement the SPL components are used?

In the previous section we discussed the studies found about SPL implementation, we also summarized those studies and their contributions. Therefore, we developed the Figures 1-8, 1-9, 1-10, 1-11, 1-12, 1-13, and 1-14 in which we showed how the SPL components are implemented with some of the previous SPL implementations approaches. Figure 1-15 summarizes the SMS findings of SRQ4. Most of the studies that specified SPL implementation approaches focused on AOP (34%), followed by SOA (22%), annotative approaches (22%), FOP (13%), and DOP (6%). In addition, we found some studies in which there was a mix of approaches to support the SPL component implementation, or in which the authors created some comparisons (see Table 1-11).

Nevertheless, we discussed in Introduction the issues with the previous approaches. For example, some of them support only coarse-grained extensions, are attached to a specific software language and include the code variations inside the reusable component files which increases the complexity, among others.

Figure 1-15: Result of SPL component implementation approaches



SRQ5: What are the available tools that automate SPL implementation?

The first analysis of the SPL tools was to focus on the integrated development environments (IDE). SPL projects are commonly developed inside a specific IDE. We found that the most used or discussed IDE was Eclipse (91%), followed by Visual Studio (9%). There were other development environments such as ArchStudio or pure::variants, but they were developed based on the Eclipse platform.

The second analysis was the use of tools for the implementation of an SPL. Most of those tools support a specific SPL implementation approach. Figure 1-16 summarizes the tool results and below we describe the most used or discussed tools based on the SPL implementation approaches:

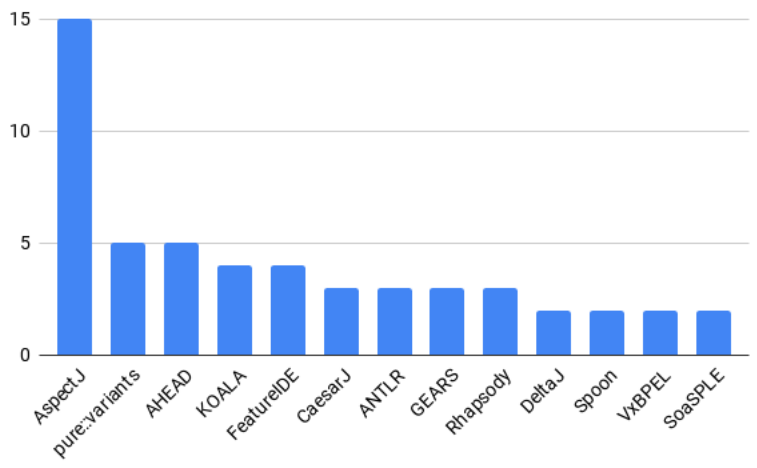
- **AOP:** AspectJ is an AOP extension for the Java programming language. Currently, AspectJ is the most consolidated AOP language [S42]. CaesarJ is an aspect-oriented language which unifies aspects, classes and packages in a single powerful construct that helps to solve a set of different problems of both aspect-oriented and component-oriented programming.
- **SOA:** VxBPEL is proposed as an extension of Business Process Execution Language (BPEL) for to the process description and definition. VxBPEL allows for

run-time variability and variability management in Web service-based systems. Variability information is defined in-line with the process definition [S76]. SoaSPLE is a generic conceptual framework that can be built on top of available Object Management Group (OMG) and standard modeling languages such as UML, BPMN, and SoaML. These languages provide modeling elements that can be used in depicting service views such as SoaMl's Service Interface elements, BPMN's Business Process elements and UML's Interaction Diagram elements [S6].

- **Annotatives:** pure::variants is a tool developed by pure-systems and is used for modeling features, expressing product variants in terms of features, and generating tailored artifacts. It is based on the well-known Eclipse platform and it can be extended by writing plug-ins in the Java programming language [S72]. Rhapsody is a tool developed by IBM Rational, that supports system engineers with modeling static and dynamic aspects of software systems using UML and SysML. Code generation for different languages is supported as well as the simulation of behavioral models such as state charts [S72]. Spoon is a tool that analyses and transforms Java code using processors. Spoon creates an abstract syntax tree of the code being analyzed and offers the API to navigate through the tree and eventually perform modifications [S78]. GEARS provide an all-in-one development environment for establishing, managing and operating your Feature-based PLE Factory. GEARS explicitly supports the integration of existing (i.e., unchanged) software. This implies that GEARS can deal with software over which no control exists [S38].
- **FOP:** AHEAD is an approach to support FOP based on stepwise refinements. The main idea behind AHEAD is that programs are constants and features are added to programs using refinement functions [S65].
- **DOP:** DeltaJ is a programming language which introduces DOP to Java [S37]. DeltaJ is available as an Eclipse plugin and it is based on the Xtext Framework.
- **Other:** FeatureIDE is a set of tools for variability modeling that enables one to create and edit feature diagrams. Furthermore, FeatureIDE provides a configuration tool to create and validate configurations with regard to the constraints defined in the variability model. Using FeatureIDE a developer can create product configurations and validate if such configurations respect the constraints expressed in the variability model. FeatureIDE already supports multiple composer engines and

approaches such as AHEAD, Munge, Antenna, AspectJ, FeatureC++, and FeatureHouse, among others [S78]. Koala [S38] is a component model consisting of an architectural description language (ADL) and tool support. The ADL serves to define interfaces, data types, basic components, and compositions (which are components themselves). The tooling serves to generate products from component compositions. Koala was primarily designed for resource-constrained software and is applied in the consumer electronics domain. ANTLR (Another Tool for Language Recognition) is an open-source project that can automatically generate a code processor from a defined grammar. The generated code processor automatically parses the input source code into a syntax tree, and outputs the code as instructed by the user [S20].

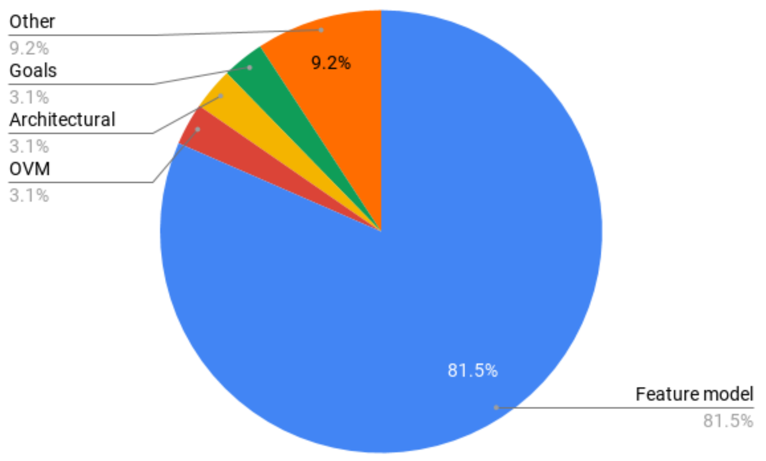
Figure 1-16: Quantity of mentions of some SPL implementation tools



SRQ6: What variability models are most used in an SPL implementation?

Without any doubt, feature models are the most used and discussed variability models in the SPL implementation literature. Some of the SMS studies specified the use of SPL variability models (see Figure 1-17). In this case, the most used or discussed variability model was feature models (82%). There were other studies in which other variability models were discussed, such as, Orthogonal Variability Model (OVM), architectural models, and goals models. However, they represent a small size of the total studies. During these years, feature models have become a popular formalism for describing the commonality and variability of SPLs.

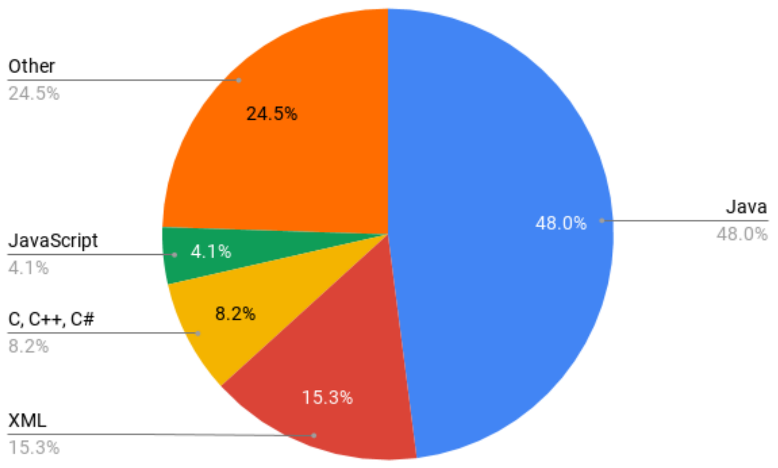
Figure 1-17: Result of variability models



SRQ7: What software languages are most used in the SPL component implementation?

Figure 1-18 summarizes the SMS findings of the software language popularity. In the studies that specify software languages, we found that the most popular was Java (48%). Followed by XML (15%), and C, C++ and C# (8%).

Figure 1-18: Result of software languages

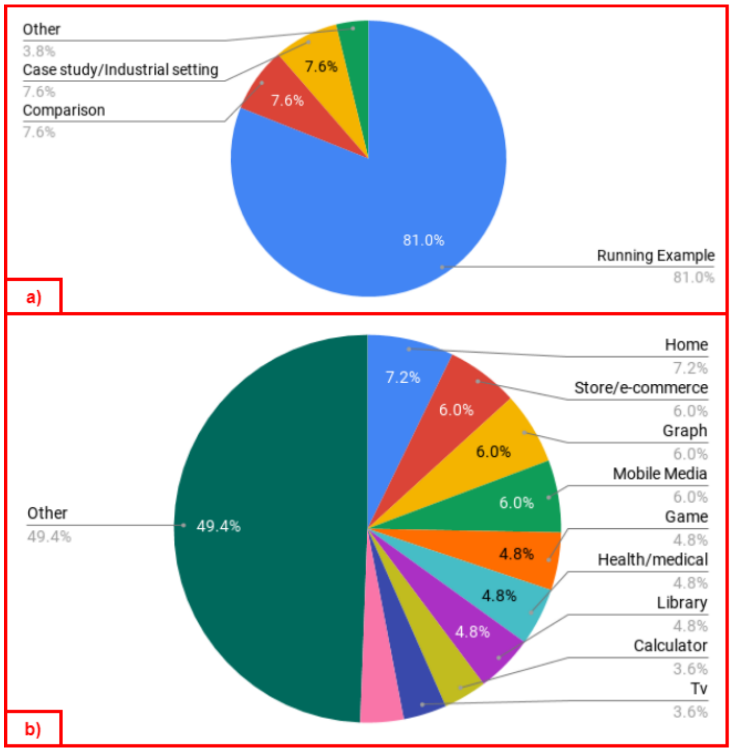


It is not a surprise that Java arises as to the most used software language, this can be due to the fact that the SPL developers are very attached to the use of IDEs such as Eclipse, and many of the authors have proposed and developed tools based on the Eclipse Modeling Framework (EMF). In this framework, developers can specify models with the use of annotated Java, UML, and XML documents, among others. That can be a reason why Java and XML are two of the most used software languages. However, as mentioned in Introduction, many SPL implementation approaches focus on supporting a specific software language or are attached to a specific software language, and software applications use an average of 5 different software languages.

SRQ8: What type of evaluations are most used in studies on SPL implementation?

As we have shown during the development of this chapter, many studies have proposed new mechanisms, concepts, processes, or techniques that improve some stages of the implementation of SPL. Many of those studies also provide a kind of evaluation for their proposals. Figure 1-19-a summarizes SMS evaluations results. Running examples are the most used evaluation technique (81%), which provide a practical way to illustrate the authors' proposals. The use of running examples was followed by comparison studies and case studies both with 8%. It is important to clarify that many studies claimed themselves to use case studies, however, we only recorded case studies for those studies that there were applied in industrial settings. If a study was claimed as a case study but in a non-industrial setting, we recorded them as a running example.

Figure 1-19: Result of evaluations and kind of examples



We also found that most evaluations recorded information such as lines of code (LOC), quantity of classes, quantity of methods, quantity of components, number of product derivations, and number of product configurations, among others.

Finally, we recorded the kind of examples that were presented in the studies' evaluations (see Figure 1-19-b). The results show a wide range of domains in which SPL are applied, which goes from the avionics domain [S50], to games [S36], library management systems [S59], and home automation [S79].

1.6 Resolving questions about research goal G3 – Topics and trends

SRQ9: What are the main topics of the selected studies?

After the SMS development, we have found some main topics in the SPL implementation domain. Some of these topics have been already discussed in Introduction and in Section 1.5; such as SPLE, domain engineering, application engineering, product derivation, variability modeling, compositional and annotative approaches, fine-grained and coarse-grained extensions, component implementation, component assembling, SPL tools, and SPL evaluations. Other main topics are discussed below:

- **MDD – MDE – MDA.** Model-driven development (MDD) and similar areas, such as model-driven engineering (MDE) and model-driven architecture (MDA) improve the way software is developed by capturing key features of a system in models which are developed and refined as the system is created. During the system's lifecycle, models are synchronized, combined and transformed between different levels of abstraction and different viewpoints. In contrast to traditional modeling, models do not only constitute documentation but are processed by automated tools [S67]. Authors and SPL developers have been taken advantage of the MDD characteristics to improve and automatize the implementation of SPL. For example: (i) Alzahmi *et al.* [S6] presented a tool that facilitates the automatic derivation of SOA applications based on MDE as an implementation methodology, (ii) Mefteh *et al.* [S49] developed an approach in which feature models can be built automatically not only from source codes but also from descriptions and uses cases diagrams, and (iii) Mohamed *et al.* [S74] presented a multi-tenant single instance software-as-a-service evolution platform based on Software Product Lines (SPLs) and MDA.

- **PLA – ADL.** Product line architecture (PLA) is an important application of software architecture in the development of a family of software products, or a software product line. It captures architectural commonality and variability among products of the product line [S84]. Architectural description languages (ADLs) can be seemed as an approach for implementing PLA concepts. ADLs typically use architectural styles to define vocabularies of types of components, connectors, properties, and sets of rules that specify how elements of those types may be legally composed in a reusable architectural domain [S66]. PLA and ADL have been also used in conjunction with SPL implementation, Zheng and Cu [S84] presented an approach to implementing product line architecture which combines a code generation and separation pattern with an architecture-based code annotation technique; the Koala tool was designed as a component model consisting of an ADL [S38]; and Barbosa *et al.* [S66] developed PL-AspectualACME which is an extension of the ACME ADL that enriches existing abstractions to express architectural variabilities.
- **DSL.** A domain specific language (DSL) is a formalism for building models which encompasses a meta-model as well as a definition of a concrete syntax that is used to represent the models. The concrete syntax can be textual, graphical or using other means, such as tables, trees or dialogs [S67]. Some authors have developed some DSL to improve the implementation of SPL, such as El-Sharkawy *et al.* [S43] who developed a tool and three custom-made DSLs to support the creation and management of software product line projects, and Pessoa *et al.* [S30] proposed an approach to developing reliable and maintainable DSPLs which uses a DSL to describe reliability goals and adaptability at runtime.
- **DSPL.** Dynamic software product lines (DSPL) have emerged as a promising strategy to develop SPL that incorporate reusable and dynamically reconfigurable artifacts. The central purpose of DSPL is to handle adaptability at runtime through variability management, as well as to maximize the reuse of components [S63].
- **CBSE.** Component-based software engineering (CBSE) focuses on the development and reuse of self-contained software assets in order to achieve better productivity and quality as software systems are composed by previously developed components used (and tested) in other contexts [S28]. CBSE has some similarities with SPL, some SPL developers incorporate CBSE processes and activities inside the SPLE.

- **COTS.** A commercial off-the-shelf (COTS) product or component is one that is used "as-is". COTS components are designed to be easily installed and to interoperate with existing system components. Lago *et al.* [S83] extended a tool to support traceability in product families which allows accommodating both newly developed and COTS components at code level.

SRQ10: What trends have these topics followed over the last years?

For the question resolution about trends, we analyzed the papers presented between 2015 and 2017. We found five main areas in which authors were developing their studies.

- **Dynamic software product lines.** Software availability has become more and more recognized as a quality issue since business transactions and many customer operations have become computerized. DSPL has become a trending topic to support dynamic product reconfiguration and adaptability at runtime [S30,S45,S63], which improve software availability.
- **SMS and SLR studies.** During the last decade, many have authors have developed several proposals in different SPL areas. Recently, some authors have developed multiple SMS and SLR studies trying to provide an overview of these proposals in different SPL areas [S54,S55,S71,S77].
- **Web and mobile systems.** Due to the internet boom, some authors have proposed some studies to apply SPL techniques to web and mobile systems [S9,S73,S80]. This also means that SPL proposals have to evolve to support many mobile and web software languages and frameworks.
- **MDD.** The use of MDD to support some processes inside the SPLE continue being a trending topic [S49, S75]. Research in MDD will allow to automatize the SPLE processes and reduce the manual intervention.
- **PLA.** The evolution of software architectures requires research in the PLA area [S20, S84]. For example, the use of PLA in microservices is a relevant research area.

1.7 Threats to validity

Threats to the validity of the study can be analyzed from the point of view of construct validity, reliability, and internal validity (Wohlin *et al.*, 2000). First, construct validity reflects the extent to which the phenomenon under study really represents what is being investigated, according to the research questions. The term software product line is well established and hence stable enough to be used as part of the search string. However, for SPL implementation, we consider that this is an ambiguous term and several authors use different names. That is the reason why we divided the SPL implementation term in eight resulting search string, trying to cover as many representative variants as possible, and to reduce the threat of having used the appropriate terms or not (see Table 1-3). Another aspect of the construct validity is the assurance that we find all the papers on the selected topic. We have searched broadly in general publication databases that index the best reputed journals and conference proceedings. The list of different publication media indicates that the width of the search is enough (see Table 1-4). Second, reliability focuses on whether the data are collected, and the analysis is conducted in a way that it can be repeated by other researchers with the same results. We defined the search terms and applied procedures, which may be replicated by others. The non-determinism of some of the databases (Google scholar) is compensated by using more reliable databases (ScienceDirect, Springer, ACM, and IEEE explore). The inclusion/exclusion criteria are related to whether the topic of the field is present in the paper or not. Finally, in order to address the internal validity concerns, a review protocol was created beforehand and evaluated by two researchers, which took on roles of quality assurance as well. The internal validity was also enhanced by following the systematic mapping guidelines proposed by Petersen *et al.* (2008).

1.8 Conclusions

This chapter presented the results of a systematic mapping study on SPL implementation. Including an overview of the processes, methods, and tools used to carry out SPL implementation; and details on the role of the SPL components in the entire process. In total, 88 studies were included in this mapping study from 2000 to March 2017. The SMS included the definition of 10 research questions which were defined and answered. These questions were divided into three categories publication, SPL implementation, and topics and trends. A summary of each category is presented below:

- **Publication.** SPL implementation remains as an interesting field in which many authors from many different countries have been proposing many contributions during the last decade. The most preferred journal to publish this type of articles is Information and Software Technology, and the most preferred conference is the

International Software Product Line Conference (SPLC). Brazil leads as the country which has more quantity of publications in this field and has the majority of the top contributions.

- **SPL implementation.** There are several approaches to implement SPL, the most discussed are AOP, SOA, annotative approaches, FOP, and DOP. There are different software tools that support specific approaches or some processes of the SPLE. The most mentioned include AspectJ, pure::variants, AHEAD, KOALA, and FeatureIDE. About variability models and software languages, feature model appears as the most preferred variability model, and Java and XML are the most used software languages. Finally, there are different kinds of evaluations in which the most discussed are running examples, case studies and comparisons.
- **Topics and trends.** Some of the general topics in the SPL implementation domain include SPLE, domain engineering, application engineering, product derivation, variability modeling, compositional and annotative approaches, MDD, PLA, DSL, DSPL, CBSE, and COTS. Therefore, current trends include DSPL, SMS and SLR studies, web and mobile systems, MDD, and PLA.

2. Running example

The use of running examples is very useful in software engineering. They have been used to provide a practical way to illustrate the concepts of a methodology, process, and technique, among others (Wileden & Kaplan 1999; Mens, 2004; Epifani *et al.*, 2009). We also found that many SPL studies used running examples as a way to describe and show their concepts (see Figure 1-15). In this thesis, we present FragOP which consists of several concepts, processes, activities, and tooling support, that should be understood and used to design and implement an SPL. Based on that fact, we defined an SPL running example that will be explained as this chapter develops and will be used to illustrate the FragOP elements and provide a realistic scenario of how to implement an SPL with the use of FragOP. The running example is consistently referred to throughout Chapters 4 and 5.

We called the running example **ClothingStores**. ClothingStores is a software product line, which consists of the development of an e-commerce store system family to manage and sell clothes. The main idea is to provide a set of capabilities, such as product management, user management, shop system, cart system, web management, sharing system, login system, database management, offline payment, and comment system, among others. The implementation of these features will allow developing several customized clothing store products.

The ClothingStores SPL was designed as a real-world example, covering most of the problems SPL developers face when implementing an SPL. These problems include:

- **Crosscutting concerns.** ClothingStores will contain a `Login` component, which in case of being part of a final product, must be integrated transversally over multiple other product files.
- **Fine-grained extensions.** ClothingStores will present multiple fine-grained extensions that must be applied for most of the derived products. For example, to

modify the header menu, to modify specific parts of the product views, to modify product class methods, SQL files, among others.

- **Coarse-grained extensions.** There are many cases in which a ClothingStores product will require coarse-grained extensions such as replacing a validation method for the admin classes and including DAO methods, among others.
- **Product customization.** Database config vars, product name, and some default texts inside the product views must be customized.
- **Managing multiple language files.** The ClothingStores SPL will be designed as a real web application which includes domain files types, such as SQL, images (.jpg and .png), JavaScript, HTML, JSP, Java, and CSS.

In the next two subsections we describe the ClothingStores requirements, and then go on to describe the ClothingStores software architecture, including its project folder structure.

2.1 Requirements

The SPL requirements define the possible capabilities of the derived software products. In an SPLE, the requirements shared by all members of the product line (**mandatory**) and requirements which are specific for one or several special products (**optional**) must be defined.

The SPL requirements are usually represented with visual languages, such as **Feature Models** (FMs; Kang *et al.*, 1990). In an FM, a **feature** can be defined as a quality or a characteristic of a (software) system (Apel *et al.*, 2013). Features have a Boolean nature even though it is well known that quality attributes, as opposed to the functional requirements, are non-Boolean. There are some other limitations of the feature modeling language that can be consulted in Mazo (2014). For the ClothingStores SPL example, we defined a total of 25 features which are described in Table 2-1 and graphically represented in Figure 2-1. Mandatory features contain an asterisk (*) at the end of the “Feature Identifier”.

Figure 2-1: ClothingStores feature model

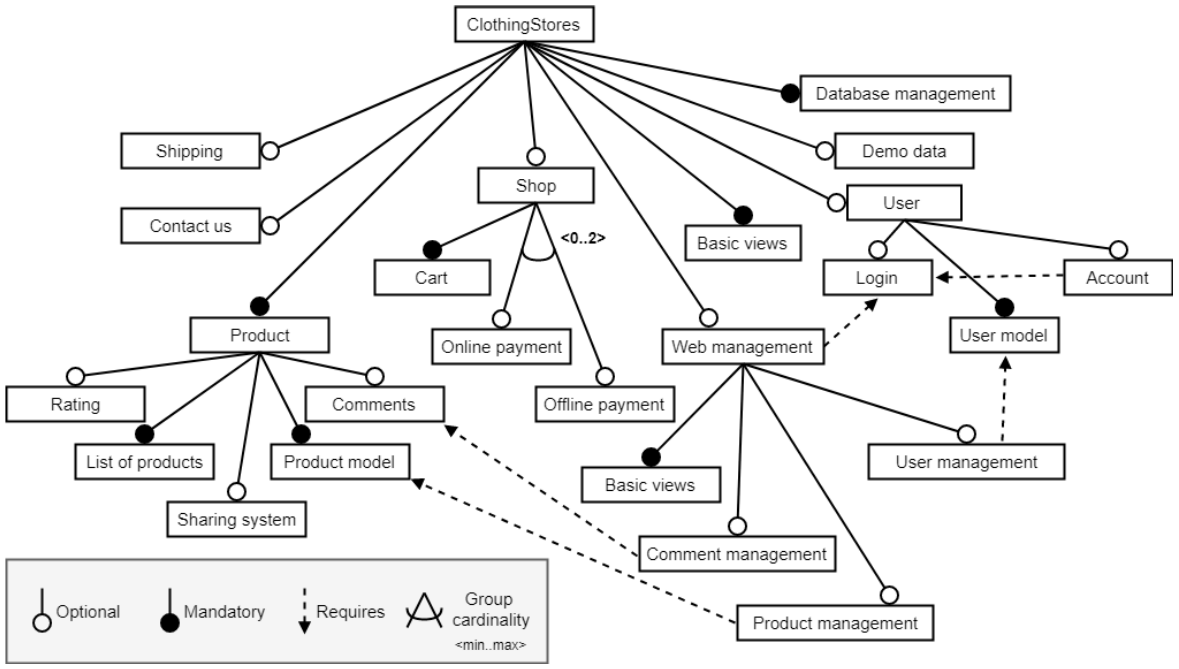


Table 2-1: ClothingStores list of features

ID	Feature Identifier	Description	Parent
F01	ClothingStores*	The root or name of the PL	
F02	Basic views*	Refers to the basic views that any ClothingStores product must contain (*e.g.*, headers, footers, home section, and CSS styles)	F01
F03	Contact us	A website section that contains the store contact information (*e.g.*, phone number, address, and email)	F01
F04	Shipping	A website section that contains the store shipping information	F01
F05	Database management*	Manages the communication with the database (in this case MySQL)	F01
F06	Demo data	Provides sample SQL data (*e.g.*, products, users, and comments)	F01
F07	Product*	Groups the product functionalities	F01
F08	Product model*	Provides a service to store product information, attributes, and its operations	F07
F09	List of products*	Represents a display service of all products in the store	F07
F10	Comments	Provides a mechanism to comment on products	F07
F11	Sharing system	Provides a mechanism to share products on Facebook and Twitter	F07
F12	Rating	Provides a mechanism to rate products	F07
F13	User	Groups the user functionalities	F01
F14	User model*	Provides a service to store user information, attributes, and its operations	F13
F15	Account	Represents a display service of the user information (*e.g.*, user name, user type, user identifier)	F13

F16	Login	Provides a mechanism to connect and disconnect from the application	F13
F17	Shop	Groups the shop functionalities	F01
F18	Cart*	Provides a mechanism to add products to the user cart, and display and remove the products added to the cart	F17
F19	Online payment	Allows payment through PayPal	F17
F20	Offline payment	Allows offline payment by providing bank account information	F17
F21	Web management	Groups the web management functionalities	F01
F22	Basic views*	Refers to the basic views that the web management module contains (*e.g.*, header and home section)	F21
F23	Product management	Allows products to be managed, such as create products, edit products, list products, and delete products.	F21
F24	User management	Allows users to be managed, such as create users, edit users, list users, and delete users.	F21
F25	Comment management	Allows comments to be managed, such as create comments, edit comments, list comments, and delete comments.	F21

In addition to the concept of feature, there are some concepts that organize these features into a feature model as presented and exemplified in the following paragraphs:

- **Mandatory:** Given two features `F1` and `F2`, `F1` father of `F2`, a mandatory relationship between `F1` and `F2` means that if the `F1` is selected, then `F2` must be selected too and vice versa. For instance, in Figure 2-1, features `Shop` and `Cart` are related by a mandatory relationship.
- **Optional:** Given two features `F1` and `F2`, `F1` father of `F2`, an optional relationship between `F1` and `F2` means that if `F1` is selected then `F2` can be selected or not. However, if `F2` is selected, then `F1` must also be selected. For instance, in Figure 2-1, features `Product` and `Rating` are related by a mandatory relationship.
- **Requires:** Given two features `F1` and `F2`, `F1` requires `F2` means that if `F1` is selected in the product, then `F2` has to be selected too. Additionally, it means that `F2` can be selected even when `F1` is not. For instance, `Web management` requires `Login` (cf. Figure 2-1).
- **Group cardinality:** A group cardinality is an interval denoted `<n..m>`, with n as lower bound and m as upper bound limiting the number of child features that can be part of a product when its parent feature is selected. If one of the child features is selected, then the father feature must be selected too. For instance in Figure 2-1, `Online payment` and `Offline payment` are related in a `<1..2>` group cardinality.

- **Exclusion:** Given two features `F1` and `F2`, `F1` excludes `F2` means that if `F1` is selected then `F2` cannot be selected in the same product. This relationship is bi-directional: if `F2` is selected, then `F1` cannot be selected in the same product.
- **Feature cardinality:** Is represented as a sequence of intervals `[Min..Max]`, with `Min` as lower bound and `Max` as upper bound limiting the number of instances of a particular feature that can be part of a product. Each instance is called a *clone*.

The previous requirements allow different kinds of software products to be defined, from very basic clothing stores products that contain some of the previous requirements, to very complete ones that contain almost all the previous requirements.

2.2 Software architecture

The previous SPL requirements provide relevant information that is useful for defining the SPL software architecture. The software architecture provides a general framework to develop different products from an SPL, this also means, the domain components' source code must be consistent with the software architecture (Zheng & Cu, 2016). The software architecture includes an architectural pattern, high-level decisions and an effective way to manage the product variability. Finally, the definition of the SPL software architecture provides relevant information for modeling and developing the domain components.

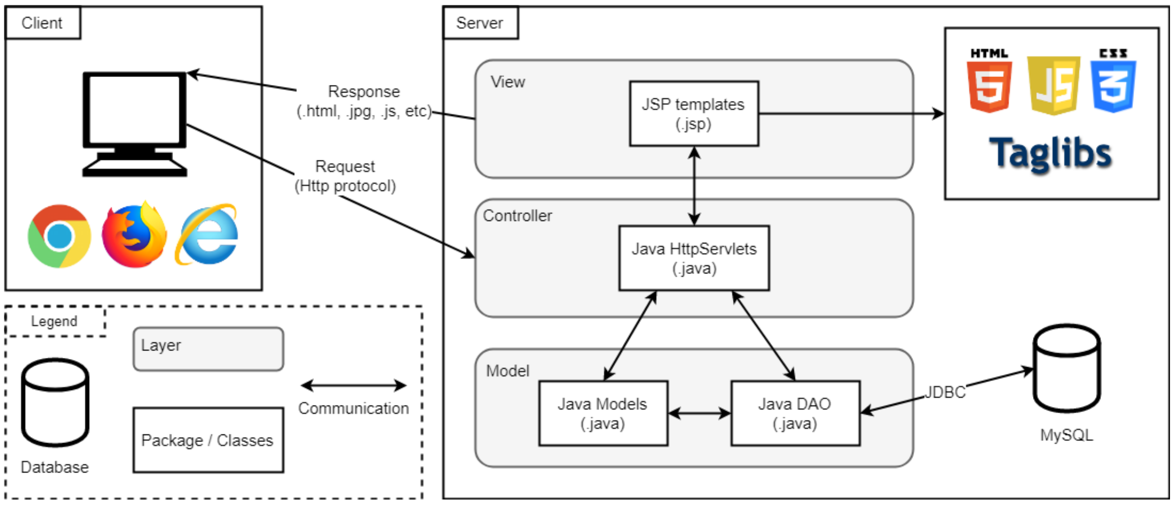
Keeping in mind the multiple challenges that the development of an SPL involves, which includes the use of several software languages, we decided to define a software architecture that used different software languages that communicated between each other. For this example, we decided:

- To implement the ClothingStores SPL by following a client-server architecture with three layers: model-view-controller (MVC).
- To develop the assets with the use of software languages, such as Java, Cascading Style Sheets (CSS), Hypertext Markup Language (HTML), and JavaServer Pages (JSP), among others.
- To select MySQL[1] as the database engine to store the application information.

[1] https://www.mysql.com/

Figure 2-2 shows the SPL reference software architecture and the relationship between the different elements. The architecture is divided into (i) **clients** who are the users that request information from the application. They access the application through the HTTP protocol with the help of browsers, such as Firefox or Google Chrome; (ii) **server** which stores the application information and responses to the client's requests. This reference software architecture will be used later as a base to design and construct the domain components.

Figure 2-2: ClothingStores reference software architecture

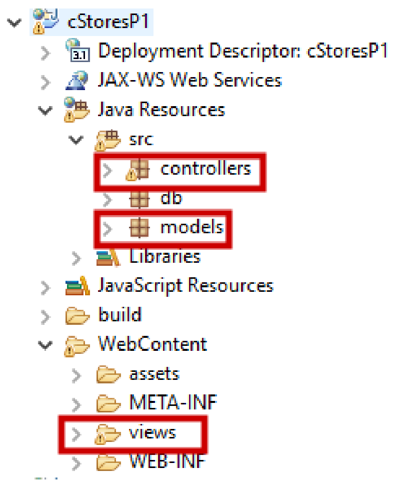


The server is divided into three layers:

- **View:** contains the graphical representation of the application. Views are developed in JavaServer Pages (JSP). A **JSP page** is a text document that contains two types of text: static data, which can be expressed in any text-based format (such as HTML), and JSP elements, which construct dynamic content. Views also contain CSS, JS, images, and taglibs files (which are a set of useful JSP custom tags).
- **Controller:** is designed as an HttpServlet. Controllers allow the client to request information to be collected and communication with the other layers. Commonly, at the end of the controller code, the request is forwarded to a JSP, and then, a response is sent to the client in the form of a .html, .css, .jpg or another client-side format file.
- **Model:** contains the application information. Java models contain application classes such as user, product, and comments. Java data access objects (DAO) provide a mechanism to communicate with the database (MySQL), in this case, the communication is made through the Java Database Connectivity (JDBC) interface.

Additional to the SPL software architecture, we also defined the project reference folder structure. This is the folder structure that any new SPL software product will follow. This structure is based on the folder structure provided by the Eclipse Enterprise Edition[1] when a new “Web Project” is created (see Figure 2-3). This structure shows where to store the controllers, models, and views.

Figure 2-3: ClothingStores project reference folder structure



```
cStoresP1
  Deployment Descriptor: cStoresP1
  JAX-WS Web Services
  Java Resources
    src
      controllers
      db
      models
    Libraries
  JavaScript Resources
  build
  WebContent
    assets
    META-INF
    views
    WEB-INF
```

2.3 Summary

This chapter introduced the ClothingStores running example. ClothingStores is defined as an SPL of an e-commerce store system family to manage and sell clothes. First, the ClothingStores requirements were defined, from very simple requirements like a contact us section to more complex like a cart system. Then, we defined ClothingStores reference software architecture as a client-server system. It included a separation of three layers (model-view-controller) and a MySQL database. We discussed the relationship between the different architectural elements, and we presented the basic project folder structure (the structure that contains any new software product by this SPL).

This running example was designed as a real-world example, covering most of the problems SPL developers face when implementing an SPL; Including crosscutting concerns, fine-grained extensions, coarse-grained extensions, product customization, and

[1] https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/photonr

managing multiple language files. Finally, we will use this running example in the next chapters, to demonstrate the capabilities of the new SPL implementation approach in a practical way.

3. Overview of the proposal

In the Introduction and in Chapter 1 we found several issues that current SPL implementation approaches present. Because of these issues, we decided to propose a new approach that combines some of the advantages of existing work. This new approach allows the component assembling to be automated, supports customization activity and the final product derivation. We named this new approach **Fragment-oriented programming (FragOP)**. In this chapter, we present an overview of this approach, including (i) a metamodel that describes the approach at an abstract level, (ii) its process with its main activities, and (iii) a tool that supports it.

3.1 FragOP metamodel

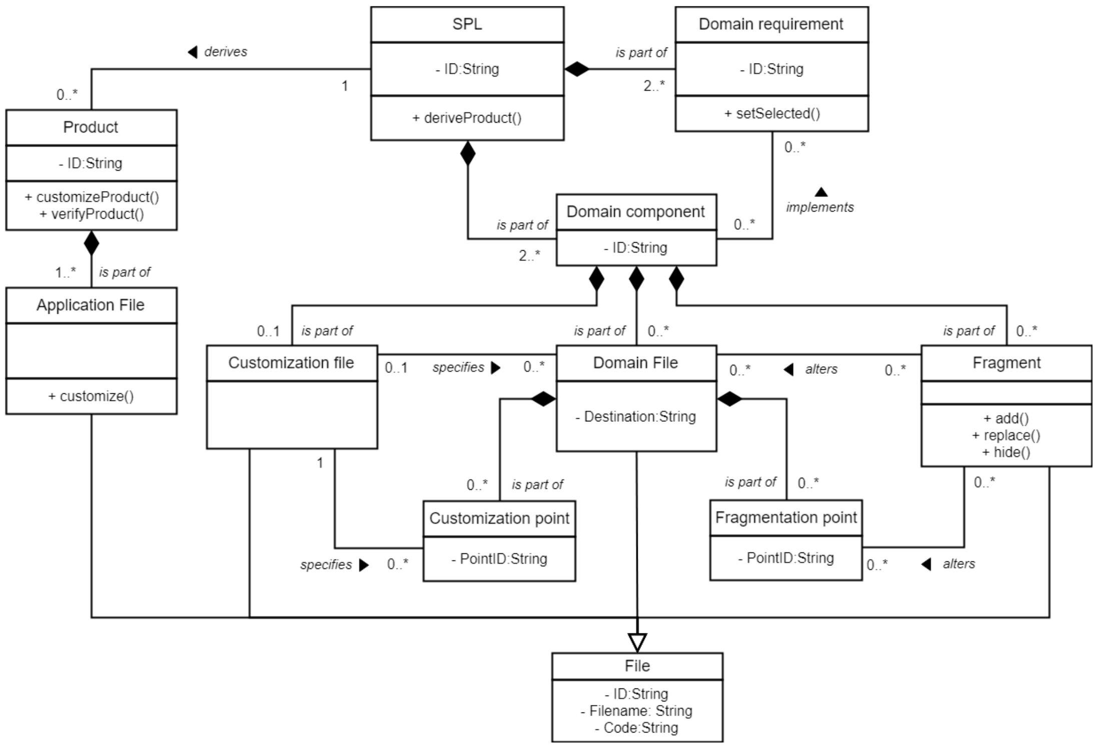
FragOP is a framework used to design, implement and reuse domain components in the context of an SPL. This framework is a mix of compositional and annotative approaches, which is based on the definition of six fundamental elements: (i) domain components, (ii) domain files, (iii) fragmentation points, (iv) fragments, (v) customization points, and (vi) customization files. The fragments act as composable units (compositional approach) and the fragmentation points act as annotations (annotative approach).

All the fundamental elements play an important role in the implementation of an SPL based on FragOP. For instance, implementing an SPL with OOP is a very different task than implementing an SPL with FOP or SOA because each approach has its own rules, structures, paradigm, and elements that support it. The role of each FragOP element, their relationships, their make-up, and the information they store can be seen in the **FragOP metamodel** (see Figure 3-1). Here, we present an overview of the FragOP metamodel elements:

- **SPL** represents the software product line and contains an ID that represents the name of the corresponding SPL.

- **Domain requirements** represent SPL domain requirements.
- **Domain components** represent SPL reusable domain components and contain an ID that represents a folder in which the component is stored.
- **File** is an abstract class used for inheritance purposes, contains an ID and the file code.
- A **domain file** is a basic element which most software components are made up of; for instance, HTML, CSS, JavaScript, Java, and JSP files.
- A **fragment** is a special type of file which alters the application code.
- A **fragmentation point** is an annotation (a very simple mark) that specifies a "point" in which a domain file can be altered.
- A **customization file** is a file which specifies the domain files (for the current domain component) that should be customized.
- **Customization points** are annotations (very simple marks) that specify the "points" in which a domain file should be customized.
- **Product** represents a folder in which a new SPL product is derived.
- **Application files** are copies of domain files which are generated when a new product is derived. These files can be also modified by the fragments.

Figure 3-1: FragOP metamodel (UML class diagram)

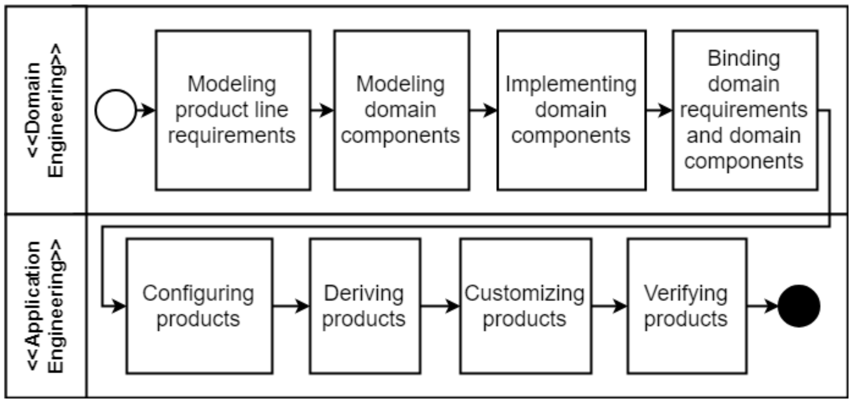


In the metamodel depicted in Figure 3-1, FragOP elements are modeled by meta-classes, and relationships between these elements are modeled by meta-associations. We formalized the meta-classes and meta-associations, with the definition of a certain number of predicates, formulas and functions; this formalization was carried out in first-order logic (FOL) according to Bradley & Manna (2007). We also developed a SWI-Prolog file with the definition of the meta-model elements, to be able to test it and reason over it. The formalization document and the SWI-Prolog file can be found in an online repository (Correa, 2018). Finally, each FragOP element is described in detail in Chapters 4 and 5.

3.2 FragOP process

The FragOP metamodel presents the main elements that must be used and understood in an SPL that implements a FragOP approach. However, it does not describe the process for the implementation of the entire SPL. That is the **FragOP process** objective. There are eight main activities that constitute this process (cf. Figure 3-2): (i) modeling PL requirements, (ii) modeling domain components, (iii) implementing domain components, (iv) binding domain requirements and domain components, (v) configuring products, (vi) deriving products, (vii) customizing products, and (vii) verifying products. These eight activities could also be grouped into two main processes: domain engineering, and application engineering.

Figure 3-2: FragOP process (UML activity diagram)


<<Domain Engineering>>
Modeling product line requirements
Modeling domain components
Implementing domain components
Binding domain requirements and domain components
<<Application Engineering>>
Configuring products
Deriving products
Customizing products
Verifying products

Below, we introduce the FragOP activities related to the domain engineering process, and the FragOP activities related to the application engineering process.

3.2.1 Domain engineering

In SPLE, the domain engineering process defines the commonalities and the variability of the SPL, culminating with the development of the domain artifacts (Metzger & Pohl, 2014). FragOP defines four activities related to this process which are summarized as follows: (i) modeling PL requirements, (ii) modeling domain components, (iii) implementing domain components, and (iv) binding domain requirements and domain components.

- **Modeling PL requirements** is the activity in which all the PL requirement are elicited. Elicitation implies the definition of mandatory and optional requirements, and the relationships or dependencies between the said requirements. Variability models (such as feature models) are commonly used to represent the PL requirements (Soltani *et al.*, 2012).
- **Modeling domain components.** Commonly, PL requirements are realized through the development of software components or pieces of code. In this activity, the PL domain components, their domain files (including fragments and customization files) and the relationship between these elements, are defined using a component model.
- **Implementing domain components** is the activity in which the components and files are developed based on the component model. The main idea is to develop reusable domain components that could be used for different PL software products. These components and files should be designed to be as generic as possible with respect to the corresponding domain (Correa & Mazo, 2018). This activity implies (i) developing the domain components with their domain files' code, (ii) including the fragmentation points, (iii) codifying the fragments, (iv) including the customization points, and (v) codifying the customization files. Here, a PL developer could use a preferred IDE that supports the codification process. The result of this activity is the development of a domain component pool that includes the reusable assets of the PL.
- **Binding domain requirements and domain components** is the activity in which a binding model between the component model and the variability model is created. The binding is an activity that links components and requirements; it specifies which domain requirements are realized by which domain components. FragOP allows a domain component to be linked with a domain requirement (one-to-one

relationship). Later, this information is used in the configuration and derivation activities.

3.2.2 Application engineering

In SPLE, the application engineering process derives the applications of the SPL from the domain artifacts and based on customer needs (Metzger & Pohl, 2014). FragOP defines four activities related to this process which are summarized below: (i) configuring products, (ii) deriving products, (iii) customizing products, and (vi) verifying products.

- **Configuring products** consists of selecting the specific features that a specific product will contain based on the stakeholder requirements (Soltani *et al.*, 2012). The result is a configured variability model.
- **Deriving products** consists of generating specific software products based on the configured variability model. The selected features and the variability model are taken as an input. Then, the binding is resolved to show what components should be assembled based on the selected features. Then, the components are assembled in a product folder (the output). In this activity, the fragments' codes are injected over the product's file codes, which allows the product derivation activity to be automated.
- **Customizing products.** Even when PL software products are derived based on the customer's needs, it is very common for these products to require customization (Montalvillo *et al.*, 2017), for example, to parameterize configuration files or variables, to modify dummy texts, and to include specific customer requirements, among others. FragOP takes advantage of the customization files and customization points and facilitates the customization activity. It shows which product's files should be customized and at what specific points. This activity is automated to permit the easy customization of the software products.
- **Verifying products.** The last activity in the application engineering process is product syntax verification. Due to the fact that FragOP allows component file codes to be injected and modified (through the use of fragments), it becomes relevant to verify the resulting products. FragOP suggests including the use of lexers and parsers to verify the syntax of each resulting file code. With the use of VariaMos this activity is automated which improves the software product quality.

The entire FragOP process is described in detail in Chapter 5.

3.3 FragOP implementation

FragOp was implemented as part of the VariaMos tool. **VariaMos**[1] is a modeling tool that incorporates a language to represent and simulate families of systems and (self) adaptive systems (Mazo *et al.*, 2015). VariaMos was initially developed at the Computer Science Research Center (CRI) of Université Paris 1 Panthéon-Sorbonne in Paris, France. Subsequently, different research groups in Colombia and France have been improving this tool. During recent years, this tool has been used in several SPL projects and approaches (Sawyer *et al.*, 2012; Mazo *et al.*, 2015; Correa *et al.*, 2018, Correa *et al.*, 2019).

Currently, VariaMos offers some capabilities such as product line requirements modeling and product simulation which are useful for designing, reasoning, and implementing SPLs. We took advantage of these capabilities and we extended VariaMos with new capabilities to support the FragOP process: (i) modeling domain components, (ii) binding (or weaving) the product line requirements model and the domain component model, (iii) configuring new products from the domain models, (iv) deriving the configured products, (v) customizing the derived products, and (vi) verifying the domain models and the derived products. Only one FragOP activity ("implementing domain components") is not supported by VariaMos and must be carried out with external software. In this case, we recommend using an integrated development environment (IDE), such as Sublime[2], IntelliJ[3], NetBeans[4], or Eclipse[5].

Instructions about how to use VariaMos and the IDEs to carry out each of the previous activities are provided in Chapter 5.

3.4 Summary

This chapter introduced an overview of the thesis proposal. It presented FragOP as a framework used to design, implement and reuse domain components in the context of an

[1] https://variamos.com/home/
[2] https://www.sublimetext.com/
[3] https://www.jetbrains.com/idea/
[4] https://netbeans.org/projects/www/
[5] https://www.eclipse.org/

SPL. FragOP is defined as a mix of compositional and annotative approaches and is based on the definition of six fundamental elements: (i) domain components (i) domain files, (iii) fragmentation points, (iv) fragments, (v) customization points, and (vi) customization files.

This chapter also presents the FragOP metamodel which describes each FragOP element, their relationships, how are made up, and the information they store. Therefore, it presents the FragOP process which is divided into eight main activities.

Finally, this chapter introduces VariaMos, which is a software modeling tool used to represent and analyze variability-based systems. VariaMos already contains relevant functionalities that support most of the FragOP process activities, in fact, VariaMos has been enhanced to support seven of the eight FragOP process activities.

The next two chapters describe in detail each FragOP metamodel element and each FragOP process activity.

4. FragOP fundamentals

In the previous chapter, we mentioned that there are six FragOP fundamental elements. In this chapter, we will describe these elements and use the running example to exemplify and demonstrate the use of these elements. In this way, we will explain how these elements support the two FragOP main capabilities (assembling and customization).

Each SPL implementation approach has its advantages and disadvantages, and all of them have different capabilities. For example, AOP is a good candidate to implement crosscutting concerns, annotative is a good candidate to support the implementation and assembling of multiple software languages and both fine-grained and coarse-grained extensions, and SOA supports web services and BPM processes. In this case, FragOP's main capabilities are both supporting generic assembling and customization (for different kinds of components developed in several software languages).

Below, we discuss two FragOP fundamental elements (domain components and domain files). These FragOP fundamental elements are transversal elements that are used in the two FragOP main capabilities. We will also discuss the FragOP assembling and customization capabilities with their own fundamental elements. These fundamental elements have been presented and described in detail in two articles (Correa *et al.*, 2018; Correa *et al.*, 2019).

Finally, the definition of these FragOP fundamental elements will allows us to answer the RQ1 because these elements specify the way in which the SPL components should be implemented.

4.1 Domain component

Related work has documented several approaches for the implementation of domain components (Thüm *et al.*, 2014), such as:

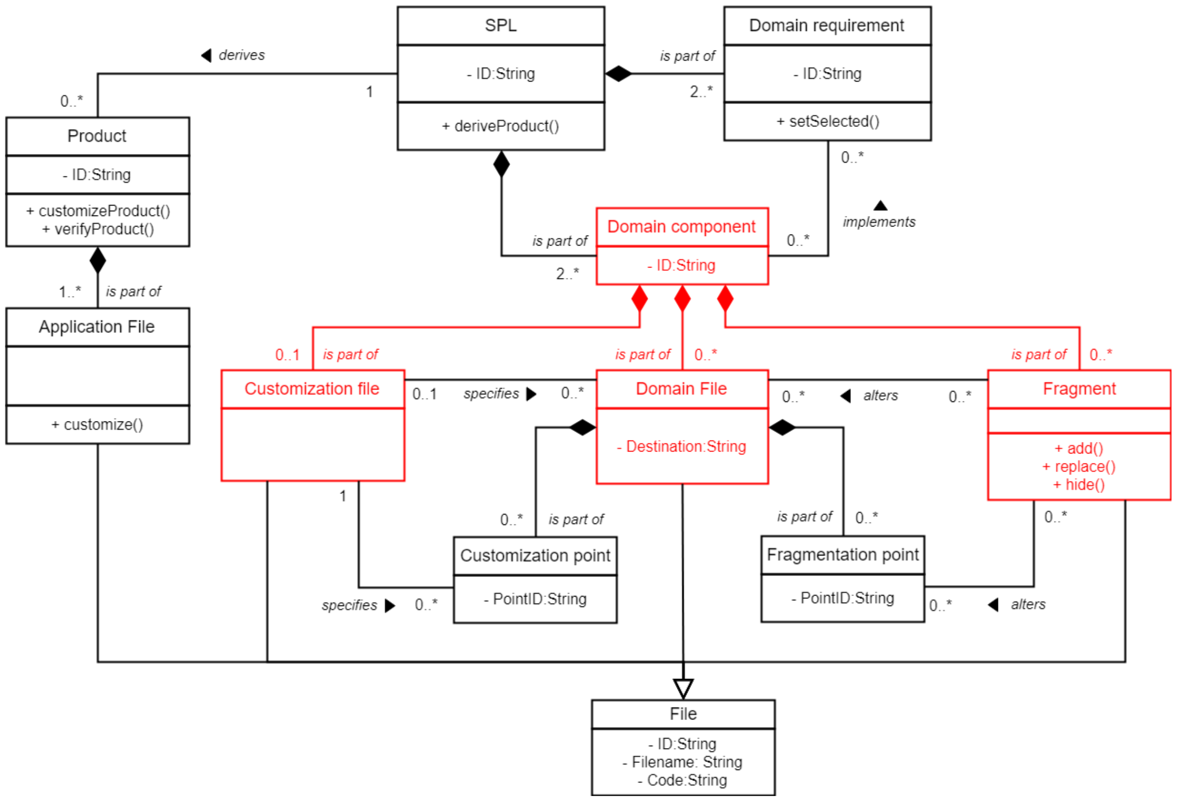
- Feature-oriented programming with AHEAD (Java 1.4), FeatureC++ (C++), and FeatureHouse (C, Java 1.5, JML, Haskell, XML, JavaCC).
- Aspect-oriented programming with AspectJ (Java).
- Delta-oriented modeling and programming with DeltaEcore and DeltaJ (Java).
- Annotation-based implementation with CIDE (multi-language), preprocessor Antenna, C preprocessor CPP by Colligens, and preprocessor Munge.

Each of these specifies its own manner of implementing the domain components. For example:

- In OOP, components are implemented with classes.
- In AOP, components are implemented with aspects.
- In DOP, components are implemented with core modules and delta modules. The core module comprises a set of classes that implements a complete product. Delta modules specify changes to be applied to the core module.
- In FOP, components are implemented with feature modules, which can be seen as increments of product functionality.
- In SOA, components are implemented with web services, which are independent functionalities that can be reused in multiple software systems.
- In annotative approaches, such as CIDE, developers simply annotate code fragments inside the original code and use tool support to view and navigate through the annotations.

In FragOP, a **domain component** is a grouping of domain files, fragments and customization files `DC = {DF, FR, CF}`. In this approach, each domain component is stored in an independent folder that contains its respective domain files, fragments and customization files. Figure 4-1 shows how the FragOP metamodel relates domain components with their files. It also shows that a domain component contains an ID. This ID represents the domain component folder name. Additionally, a domain component must contain at least one domain file or fragment.

Figure 4-1: FragOP metamodel highlighting the domain component, domain file, fragment, and customization file relationship



The property of each domain component stored in its own folder has been used in approaches such as Feature IDE with AHEAD (FOP) or with FeatureHouse (FOP). However, other approaches such as DeltaJ (DOP), AspectJ (AOP) and CIDE (annotative approach) do not store a domain component in a separate folder, rather they create a base project in which everything could be stored at the same level. Storing a domain component in its own dedicated folder supports the SPL maintainability and evolution because an SPL developer can easily find the specific files that are related to a specific domain component.

It is also important to highlight that FragOP and most of the approaches that store domain components in their own folder do not allow a hierarchy to be specified among the domain components. This means that the storing of a domain component inside another domain component is not allowed. We think that restricting the hierarchy of components improves the reusability because it keeps the domain components as independent as possible.

4.2 Domain file

In FragOP, domain components are made up of **domain files** that represent HTML, CSS, JavaScript, Java, and JSP files, among others. Any file that could be reused for the development of multiple SPL products can be considered a domain file. This means that in the FragOP approach, a domain file could be as complex as a software class that allows communication with a database, or as simple as a text file that contains configuration variables. The FragOP metamodel (see Figure 4-1) indicates that a domain file contains (i) an ID, (ii) a filename, (iii) the file code, and (iv) a destination which represents the final location in which the domain file must be assembled. This final location must be consistent with the SPL basic project structure (see Figure 2-3).

Supporting different domain files developed in several software languages is a key characteristic of FragOP. As we mentioned in the Introduction: (i) according to Mayer and Bauer (2015) who analyzed 1150 open source projects, a mean number of 5 different languages are used in each project; (ii) compositional approaches are usually attached to a particular host language (Kästner & Apel, 2008); and (iii) annotative approaches usually use *`#ifdef`* and *`#endif`* statements to surround the component code, although not all software languages provide these statements, and many other annotative approaches provide limited support to few software languages.

For instance, if an SPL adopts a DOP (DeltaJ) approach, the Java assets can be easily managed with the DeltaJ tool. However, other assets such as images, HTML files, CSS files, and XML files must be manually managed by the SPL developer.

Below, we present some examples of real domain files. Listing 4-1 shows the source code of three ClothingStores domain files: (i) `BasicViewsGeneral-Header` (header.jsp), (ii) `UserManagement-ManageUsers` (ManageUsers.java), and (iii) `DatabaseManagement-Config` (Config.java).

- **ID: BasicViewsGeneral-Header – Filename: header.jsp – Destination: WebContent/views/header.jsp** is a file which is written in JSP and HTML and represents the header of the application. This code contains a menu (the highlighted code), which corresponds to an unordered list with only one element (*i.e.*, `Home`) that is linked to the home section of the application. This domain file belongs to the `BasicViewsGeneral` domain component.

- **ID: UserManagement-ManageUsers – Filename: ManageUsers.java – Destination: src/controllers/admin/ManageUsers.java** is a file which is written in Java and represents a controller for managing the user information. It contains three functions: (i) `doGet` which is used to display the users and to remove users, (ii) `doPost` which is used to create new users, and (iii) `validation` (the highlighted code) which was created with the intention of executing some validations before the `doGet` and `doPost` execution. This domain file belongs to the `UserManagement` domain component.
- **ID: DatabaseManagement-Config – Filename: Config.java – Destination: src/db/Config.java** is a file which is written in Java and represents a database configuration file. It defines four variables (the highlighted code) which allow communication with the database engine. As a domain file, these variables present sample values, however, the value of each variable must be changed (customized) for the final product. This domain file belongs to the `DatabaseManagement` domain component.

Listing 4-1: BasicViewsGeneral-Header (header.jsp), UserManagement-ManageUsers (ManageUsers.java), and DatabaseManagement-Config (Config.java) component file source codes

BasicViewsGeneral-Header (header.jsp)

```
<%@ page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix = "fn" %>
<html>
  <head>
    <title>${title}</title>
    <link rel="stylesheet" type="text/css" href="<c:url value =
"/assets/css/bootstrap.min.css"/>" />
    <link rel="stylesheet" type="text/css" href="<c:url value =
"/assets/css/style.css"/>" />
  </head>

  <body>
    <nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark">
      <button class="navbar-toggler" type="button" data-toggle="collapse"
data-target="#navbarsExampleDefault" aria-controls="navbarsExampleDefault"
aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>

      <div class="collapse navbar-collapse" id="navbarsExampleDefault">
```

```
        <ul class="navbar-nav mr-auto">
          <li class="nav-item active">
            <a class="nav-link" href="<c:url value='Home'/>">Home <span
class="sr-only">(current)</span></a>
          </li>
        </ul>
      </div>
    </nav>
   <div>
```

UserManagement-ManageUsers (ManageUsers.java)

```
package controllers.admin;

import java.io.IOException; import javax.servlet.RequestDispatcher; import
javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet; import
javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest; import
javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession; import models.User; import
models.UserDAO;

@WebServlet(urlPatterns = {"/Admin/Users"})
public class ManageUsers  extends HttpServlet {

      protected boolean validation(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException{
       return true;
      }

      @Override
      protected void doGet(HttpServletRequest request, HttpServletResponse
response)
      throws ServletException, IOException {

             if(this.validation(request, response)){
                  String remove = request.getParameter("remove");
                  if(remove != null){
                    UserDAO.remove(Integer.parseInt(remove));
                  }

                  request.setAttribute("users",UserDAO.getUsers());
                  request.setAttribute("title", "Admin Panel - Users");
                         RequestDispatcher view =
request.getRequestDispatcher("../views/admin/users.jsp");
                         view.forward(request, response);
             }
       }

       protected void doPost(HttpServletRequest request, HttpServletResponse
                    response) throws ServletException, IOException {

             if(this.validation(request, response)){
                    String user = request.getParameter("user");
                    String pass = request.getParameter("pass");
                    String name = request.getParameter("name");
                    String type = request.getParameter("type");

                    User u = new User(name,type,user,pass); UserDAO.insert(u);
                    response.sendRedirect("Users");
             }
       }
}
```

DatabaseManagement-Config (Config.java)

```
package db;

public class Config {
      public static final String db_driver = "com.mysql.jdbc.Driver";
      public static final String db_url = "URL";
      public static final String db_user = "USER";
      public static final String db_pass = "PASS";
}
```

4.3 FragOP assembling capability

To implement software product lines efficiently, the domain component code has to be **variable**. Variability is defined as the ability to derive different products from a common set of artifacts (Apel *et al.*, 2013). This means the approach, tool, paradigm or methodology used to implement the SPL domain components should support the code variability.

For a better understanding of the variability concept, we present the following scenario. Suppose that an SPL contains two domain components, login and user management. If a customer wants an application that includes the previous two components, then, it is very common that these components must be assembled as part of the product derivation activity (to include some functionalities of the login component inside the user management component). The variability scenario can be also applied to the files presented in Listing 4-1. For example, other domain components could require the modification of the `BasicViewsGeneral-Header` file, specifically to add new elements in the header menu.

To conclude, if the domain component code supports variability, the assembly activity could be automated. However, if the domain component code does not support variability, the assembly activity must be carried out manually, which affects the SPL efficiency.

The FragOP approach supports the domain component assembly through the use of three FragOP fundamental elements, domain files, fragmentation points and fragments (Correa *et al.*, 2018; see Figure 4-2). Figure 4-3 shows an example of the connection between these FragOP fundamental elements involved in a realistic assembly scenario. It also shows how FragOP supports variability at the code level. In this example, a domain file (header.jsp) supports the code variability through the inclusion of a fragmentation point (menu-modificator). Additionally, a fragment (alterHeader.frag) specifies a code alteration in the previous fragmentation point of the previous domain file. The fragmentation point, fragment, and the example are fully explained in the next two subsections.

Figure 4-2: FragOP metamodel highlighting the domain file, fragment, and fragmentation point relationship

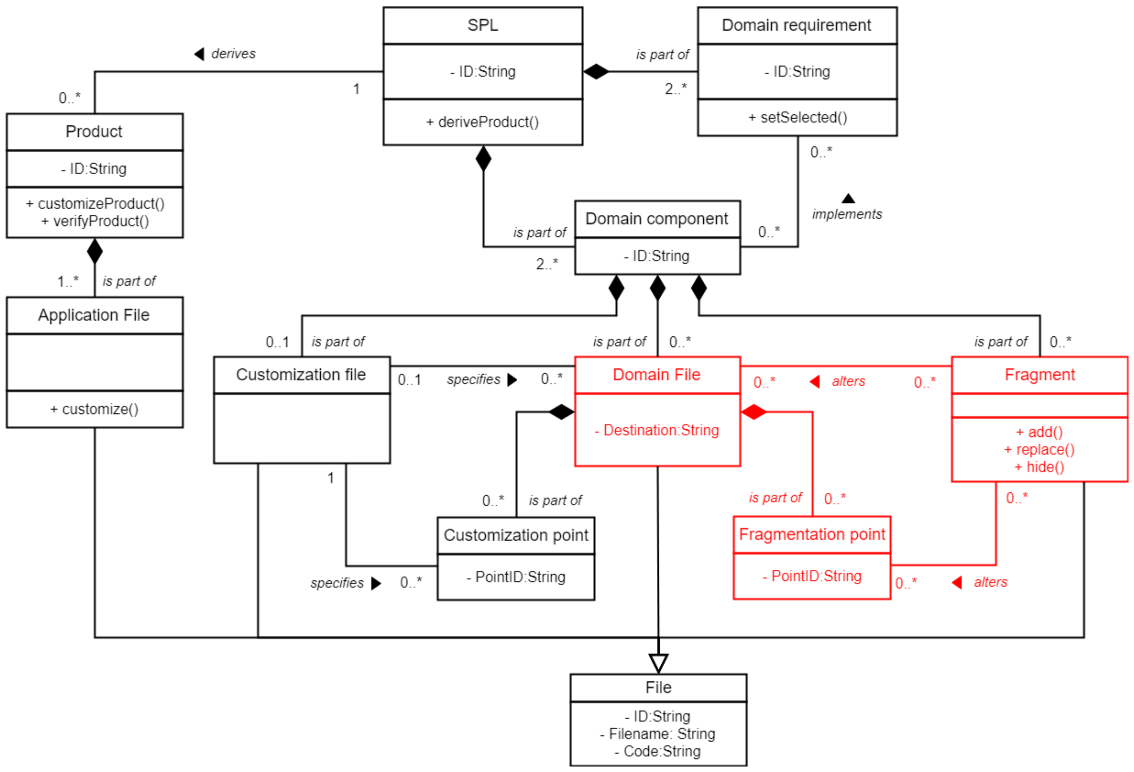
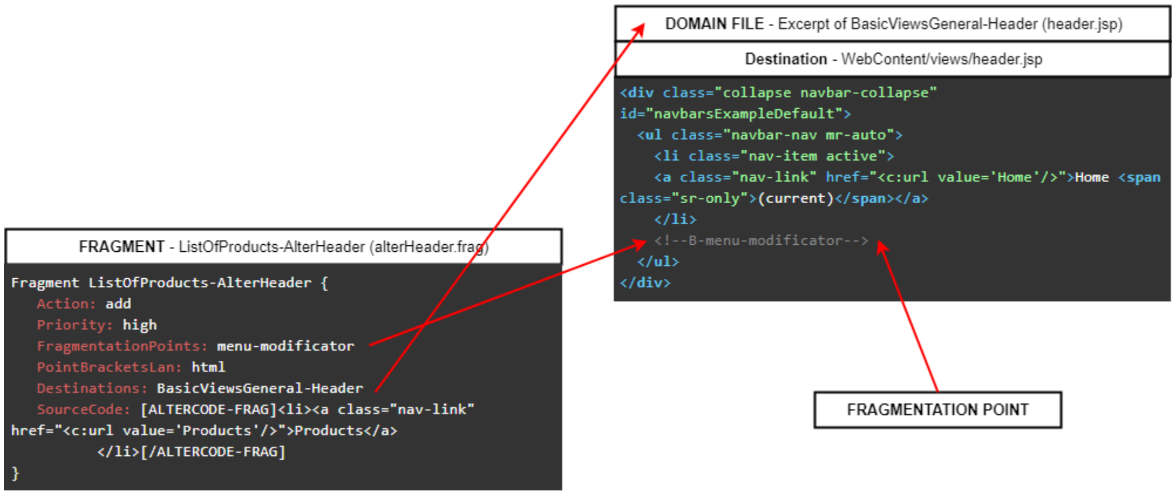


Figure 4-3: An example of the connection between a domain file, a fragment, and a fragmentation point


DOMAIN FILE - Excerpt of BasicViewsGeneral-Header (header.jsp)

Destination - WebContent/views/header.jsp

```
<div class="collapse navbar-collapse"
id="navbarsExampleDefault">
  <ul class="navbar-nav mr-auto">
    <li class="nav-item active">
    <a class="nav-link" href="<c:url value='Home'/>">Home <span
class="sr-only">(current)</span></a>
    </li>
    <!--B-menu-modificator-->
  </ul>
</div>
```

FRAGMENT - ListOfProducts-AlterHeader (alterHeader.frag)

```
Fragment ListOfProducts-AlterHeader {
   Action: add
   Priority: high
   FragmentationPoints: menu-modificator
   PointBracketsLan: html
   Destinations: BasicViewsGeneral-Header
   SourceCode: [ALTERCODE-FRAG]<li><a class="nav-link"
href="<c:url value='Products'/>">Products</a>
          </li>[/ALTERCODE-FRAG]
}
```

4.3.1 Fragmentation point

In FragOP, we use annotations to support code variability. In this approach, we call them fragmentation points. A **fragmentation point** is an annotation (a very simple mark) that specifies a “point” at which a domain file can be altered. This is a key FragOP element because it allows developers to define very specific code locations in which a domain file could be extended or refined (could vary).

Listing 4-2 shows the fragmentation point shape. FragOP suggests creating fragmentation points by starting with a comment block `LanguageCommentBlock` based on the current file language type. For example, for a file written in Java, the fragmentation point should start with `/*` and should end with `*/`. For a file written in HTML, the fragmentation point should start with `<!--` and should end with `-->`. This way, the source code of a file is not altered by the addition of the fragmentation points, ensuring code consistency and code maintainability. If a specific file code does not provide a comment block (like txt files), then, we suggest creating a regular expression, like `[FragAnnot][/FragAnnot]`.

Listing 4-2: Fragmentation point shape

```
LanguageCommentBlock<B|E>-<PointID>LanguageCommentBlock
```

After the `LanguageCommentBlock` opening section, the fragmentation point continues with `<B|E>-<PointID>`. `<B|E>` corresponds to a fragmentation point begin section (`B`) or end section (`E`). At the first occurrence of a fragmentation point, it should contain the letter `B`. The end section is optional because it is used to delimitate where a fragmentation point ends, which is only required to replace and hide actions that we will describe in the next section. The fragmentation point continues with a minus (-) symbol and a `PointID`, which is a custom text that is used to identify the fragmentation point. Finally, the `LanguageCommentBlock` closing section should be added. Listing 4-3 shows a fragmentation point example. Listing 4-3 shows a fragmentation point example.

Listing 4-3: Fragmentation point shape example

```
<!--B-menu-modificator-->
```

We decided to use annotations because it supports fine-grained extensions (changes at lower levels, such as changes in a fixed position inside a class method). This is also very useful for specifying changes to non-object-oriented software assets, such as HTML or CSS. However, fragmentation points have some important differences from the common annotations used by other approaches:

- **Fragmentation points do not use if, else statements.** Some annotative approaches use these statements to specify when a code variation should be executed or not. However, as we mentioned, not all software languages provide if, else statements.
- **Fragmentation points do not include the variant code.** For example, in the Munge approach, code variations are annotated by feature directives using IF and END inside comments. This means that the domain files include the base code, and also all the possible variant codes (this also applies for other approaches such as CIDE and Antenna). This affects the domain files readability, maintainability, and evolution because in an SPL there can be thousands of code variants. In FragOP, the variable code is located inside a new type of file called fragment which will be discussed in the next section.
- **Fragmentation points in the form of comment blocks.** In the Munge approach, the conditional tags are contained in Java comments (so they do not interfere with development environments such as Eclipse). In FragOP, fragmentation points are also defined with language comments.

Finally, Listing 4-4 shows the source code of the new `BasicViewsGeneral-Header` (header.jsp) and `UserManagement-ManageUsers` (ManageUsers.java) files. They were refined with the inclusion of two fragmentation points.

- **`menu-modificator`** is a fragmentation point which was included inside the header.jsp file (inside the menu navigation bar). The main idea is that other domain components could require the modification of the header.jsp file, specifically to add new elements to the header menu.
- **`validation-function`** is a fragmentation point which was included inside the ManageUsers.java file (surrounding the `validation` function). The main idea is that a component such as `Login` could require the modification of the ManageUsers.java file. If `Login` is present in the derived product, the ManageUsers.java `validation` function should be replaced with a new one that includes a call to the login class or to the login elements. This way, the new

`validation` function is able to check that only permitted users (for instance admins) are using the ManageUsers.java class.

Listing 4-4: Refined BasicViewsGeneral-Header (header.jsp) and UserManagement-ManageUsers (ManageUsers.java) component files

BasicViewsGeneral-Header (header.jsp)

```
<%@ page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix = "fn" %>
<html>
  <head>
    <title>${title}</title>
    <link rel="stylesheet" type="text/css" href="<c:url value =
"/assets/css/bootstrap.min.css"/>" />
    <link rel="stylesheet" type="text/css" href="<c:url value =
"/assets/css/style.css"/>" />
  </head>

  <body>
    <nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark">
      <button class="navbar-toggler" type="button" data-toggle="collapse"
data-target="#navbarsExampleDefault" aria-controls="navbarsExampleDefault"
aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>

      <div class="collapse navbar-collapse" id="navbarsExampleDefault">
        <ul class="navbar-nav mr-auto">
          <li class="nav-item active">
            <a class="nav-link" href="<c:url value='Home'/>">Home <span
class="sr-only">(current)</span></a>
          </li>
          <!--B-menu-modificator-->

        </ul>
      </div>
    </nav>
   <div>
```

UserManagement-ManageUsers (ManageUsers.java)

```
package controllers.admin;
import java.io.IOException; import javax.servlet.RequestDispatcher; import
javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet; import
javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest; import
javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession; import models.User; import
models.UserDAO;

@WebServlet(urlPatterns = {"/Admin/Users"})
public class ManageUsers  extends HttpServlet {

       /*B-validation-function*/
       protected boolean validation(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException{
       return true;
       }
       /*E-validation-function*/

            @Override
            protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {

             if(this.validation(request, response)){
                  String remove = request.getParameter("remove");
                  if(remove != null){
                   UserDAO.remove(Integer.parseInt(remove));
                  }

                  request.setAttribute("users",UserDAO.getUsers());
                  request.setAttribute("title", "Admin Panel - Users");
                         RequestDispatcher view =
request.getRequestDispatcher("../views/admin/users.jsp");
                         view.forward(request, response);
             }
             }

             protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {

             if(this.validation(request, response)){
                   String user = request.getParameter("user");
                   String pass = request.getParameter("pass");
                   String name = request.getParameter("name");
                   String type = request.getParameter("type");

                   User u = new User(name,type,user,pass); UserDAO.insert(u);
                   response.sendRedirect("Users");
             }
       }
}
```

4.3.2 Fragment

Fragmentation points define specific points in which a file can be altered, but how to alter those files? That is the objective of fragments. A **fragment** is a special type of file in which the SPL developers specify code alterations to the domain files. It is worth noting that these alterations are designed at the domain level to be used at the application level when components are being assembled for the derivation of new products (described in Section 5.6), which guarantees the reusability of the domain components. In general, a fragment respects the shape presented in Listing 4-5 and is explained thereafter.

Listing 4-5: Fragment shape

```
Fragment <ID> {
 Action: <add || replace || hide>
 Priority: <high || medium || low || priority_number>
 PointBracketsLan: <language>
 FragmentationPoints: <pointID1, pointID2, ...>
 Destinations: <fileID1, fileID2, ... || path1, path2, ...>
 SourceFile: <filename>
 SourceCode: [ALTERCODE-FRAG]<code>[/ALTERCODE-FRAG]
}
```

Fragment <*ID*>. `ID` serves as an identifier for the fragment. The `ID` is used when the components are assembled, allowing the developers to find the fragment that has been responsible for any alteration, which is useful for code traceability.

Action: <*add* || *replace* || *hide*>. Specifies the type of the alteration.

• *add* allows a piece of code to be injected at specific `PointIDs`.

• *replace* allows a piece of code to be replaced at specific `PointIDs` or (ii) allows a file to be replaced on specific destination paths.

• *hide* allows a piece of code to be hidden at specific fragmentation `PointIDs` (the pieces of code are placed inside a comment block).

Priority: <*high* || *medium* || *low* || *priority_number*>. Specifies the fragment priority (`high`, `medium` or `low`). Fragments with `high` priority are assembled before fragments with `medium` or `low` priority. Therefore, it is possible to specify a `priority_number` (INT number). By default, `high` priority takes a value of 10, `medium` priority takes a value of 500, and `low` priority takes a value of 1000. This feature could be useful in a case where two or more different fragments inject code at the same fragmentation point. For example, two different fragments could inject code into the header menu (in order to include new menu options). Depending on each fragment priority, one code will be injected first and the other will be injected second (which allows a code integration order to be defined).

PointBracketsLan: <*language*> (Optional). `Language` specifies the comment bracket language in which the fragmentation points are defined. For example: PHP, HTML or Java.

FragmentationPoints: <*pointID1*, *pointID2*, ...> (Optional). `PointIDs` are unique texts which serve to identify fragmentation points. The user is able to define multiple fragmentation points and destinations, which means that the fragment source code or source file will be injected in several places.

Destinations: <*fileID1*, *fileID2*, ... || *path1*, *path2*, ...>.

• *FileIDs* represent the domain files to be altered.

• *Paths* represent the locations where a file should be replaced.

SourceFile: <*filename*> (Optional). `Filename` represents the new file to be added.
SourceCode: <*code*> (Optional). `Code` contains the source code that will be injected.

We took advantage of different characteristics of several composable units, such as aspects, deltas modules and feature modules to design the fragment shape. Below, we discuss some of the main characteristics.

- **Fragments as composable units.** The advantage of having the code variants as independent composable units is that domain files do not contain all the possible code variants, as is common in most annotative approaches.
- **Fragments linked to the fragmentation points.** Commonly, compositional approaches present two different types of component elements. In FOP, there are classes and class refinements. In DOP, there are base modules and a set of delta modules. In AOP, there are program files and aspects. One of the previous two elements of each approach modifies the other element (for instance, delta modules modify the base modules code). These modifications are commonly linked to an object-oriented specific element (such as a class, class method or class attribute) or to a tree element (some approaches such as FeatureHouse model the domain components by tree structures). The problem is that many domain files are not object-oriented, such as HTML files or XML files; and other domain files do not provide tree structures, such as SQL files, or TXT files. Therefore, the compositional approaches do not allow fine-grained extensions. In the present case, the fragment element establishes a connection to the domain file element through a fragmentation point, which allows fine-grained extensions (see Figure 4-3).
- **Fragments work for multiple domain file languages.** Compositional approaches that rely on object-oriented elements to execute code variants are usually attached to host languages. For example, DeltaJ (DOP) only works with Java files and FeatureC++ (FOP) only works with C++ files. Compositional approaches that rely on tree structures, such as FeatureHouse (FOP), only work with languages that provide tree structures (such as Java, C, and XML). FragOP fragments do not rely on object-oriented elements or tree structures, solely relying on fragmentation points that can be added to many different domain file languages. This makes fragments a good candidate to support variability over multiple domain file languages.

- **Fragments that replace entire files.** Another key characteristic of this approach is that fragments are able to replace an entire file. This could be useful for domain files that cannot be modified with the inclusion of fragmentation points, such as images or PDF files. For example, suppose that an SPL contains a “general views” component that includes a “default logo”. The SPL also contains a “premium version” component with a “premium logo”. In this case, it is recommended to create a fragment that will replace the “default logo” with the “premium logo” when the “premium version” component is assembled.
- **Fragments specify the alteration order.** Compositional approaches use composable units to generate the application code. However, in most of these, it is not possible to specify the order or the code line in which composable units are included. In annotative approaches, this problem does not exist because the domain files contain all possible code variations, this way the SPL developer can specify the order and the code line of the code variations. Fragments allow to specify the alteration order through the fragments’ priority; here the SPL developer can define if a fragment code should be injected before other fragment codes with lower priority; and with fragmentation points, the code variations can be easily located in specific domain file code lines.
- **Fragments allow a piece of code to be injected at multiple locations.** Approaches such as FeatureHouse (FOP), FeatureC++ (FOP), and DeltaJ (DOP) only allow a specific piece of code to be injected (method and attribute, among others) into a specific file or class. For example, if an SPL developer wants to include the same class attribute into two different classes, he/she has to create two delta modules to include the same attribute in the two classes. In annotative approaches, the same code variation (new attribute) must be specified twice (once for each class). On the other hand, AspectJ (AOP) allows an aspect (a code variation) to be included in several places, but it is limited to object-oriented elements. In FragOP, a single piece of code can be easily injected at multiple locations, through the definition of multiple fragment destinations.

For a better understanding of how fragments work, consider the following case based on the ClothingStores example. Listing 4-6 shows (i) the `ListOfProducts-AlterHeader` (alterHeader.frag) code which specifies that the `BasicViewsGeneral-Header` file

(Destinations) will be altered in the `menu-modificator` (FragmentationPoints) with a `high` priority. In this case, the fragment will `add` (Action) a new menu element (SourceCode) inside the file. This is consistent with the example presented in Figure 4-3. And (ii) the `Login-AlterAdmin` (alterAdmin.frag) code specifies that the `UserManagement-ManageUsers` file (Destinations) will be altered in the `validation-zone` (FragmentationPoints) with a `high` priority. In this case, the fragment will `replace` (Action) the `UserManagement-ManageUsers` validation function with a new validation function (SourceCode).

Listing 4-6: ListOfProducts-AlterHeader (alterHeader.frag) and Login-AlterAdmin (alterAdmin.frag) fragment source codes.

ListOfProducts-AlterHeader (alterHeader.frag)

```
Fragment ListOfProducts-AlterHeader {
   Action: add
   Priority: high
   FragmentationPoints: menu-modificator
   PointBracketsLan: html
   Destinations: BasicViewsGeneral-Header
   SourceCode: [ALTERCODE-FRAG]<li>
            <a class="nav-link" href="<c:url value='Products'/>">Products</a>
           </li>[/ALTERCODE-FRAG]
}
```

Login-AlterAdmin (alterAdmin.frag)

```
Fragment Login-AlterAdmin {
   Action: replace
   Priority: high
   FragmentationPoints: validation-function
   PointBracketsLan: java
   Destinations: UserManagement-ManageUsers
   SourceCode: [ALTERCODE-FRAG]protected boolean validation(HttpServletRequest
request, HttpServletResponse response) throws ServletException, IOException{
             HttpSession session = request.getSession();
       User u = (User) session.getAttribute("datauser");
       if(u == null) { response.sendRedirect("../Home"); return false;   }
       else if(!u.getType().equals("admin")){
             response.sendRedirect("../Home"); return false;
       }
       return true;
       }[/ALTERCODE-FRAG]
}
```

It is important to highlight that the previous fragments are designed to inject their code only when a software product is derived. This is carried out later as part of the FragOP derivation activity, which is described in Section 5.6 (Listing 5-1).

4.4 FragOP customization capability

Product customization is a critical task of SPLE. The domain components hardly ever fully satisfy the requirements of a specific software product. Thus, a customization process is required in almost all PL (Cobaleda *et al.*, 2018).

The FragOP approach supports the product customization through the use of three FragOP fundamental elements, domain files, customization points and customization files (Correa *et al.*, 2019; see Figure 4-4). Figure 4-5 shows an example of the connection between these FragOP fundamental elements involved in a realistic customization scenario. It also shows how FragOP supports product customization. In this case, a domain file (Config.java) supports product customization through the inclusion of a customization point (`vars`). Additionally, a customization file (customization.json) specifies the customization points of the domain files of the current domain component `DatabaseManagement`. The customization point, customization file, and the example are fully explained in the following two subsections.

Figure 4-4: FragOP metamodel highlighting the domain file, customization file, and customization point relationship

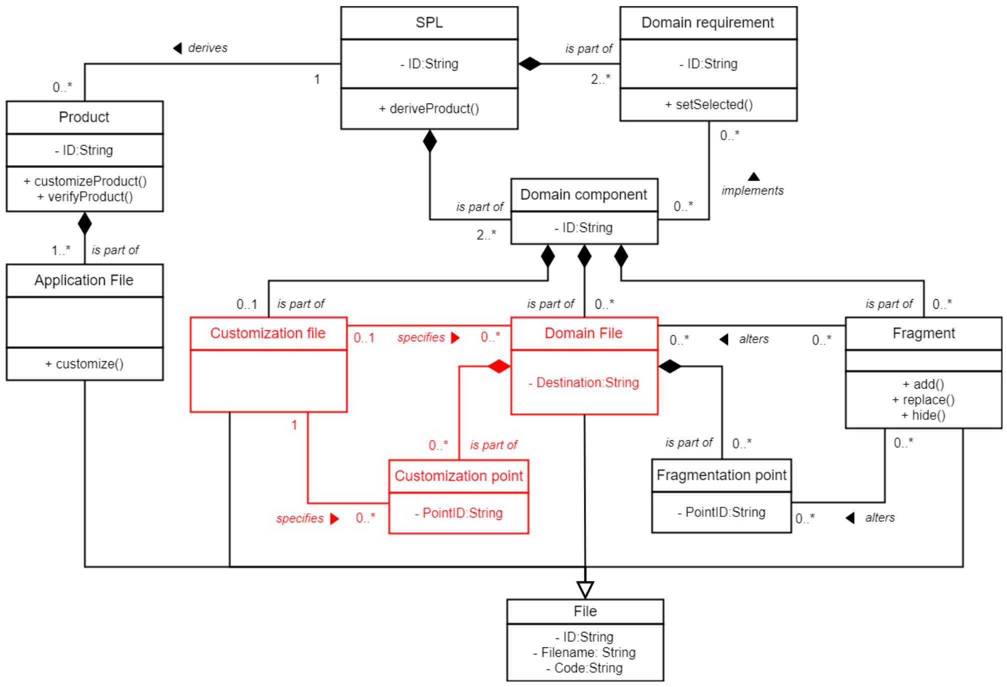
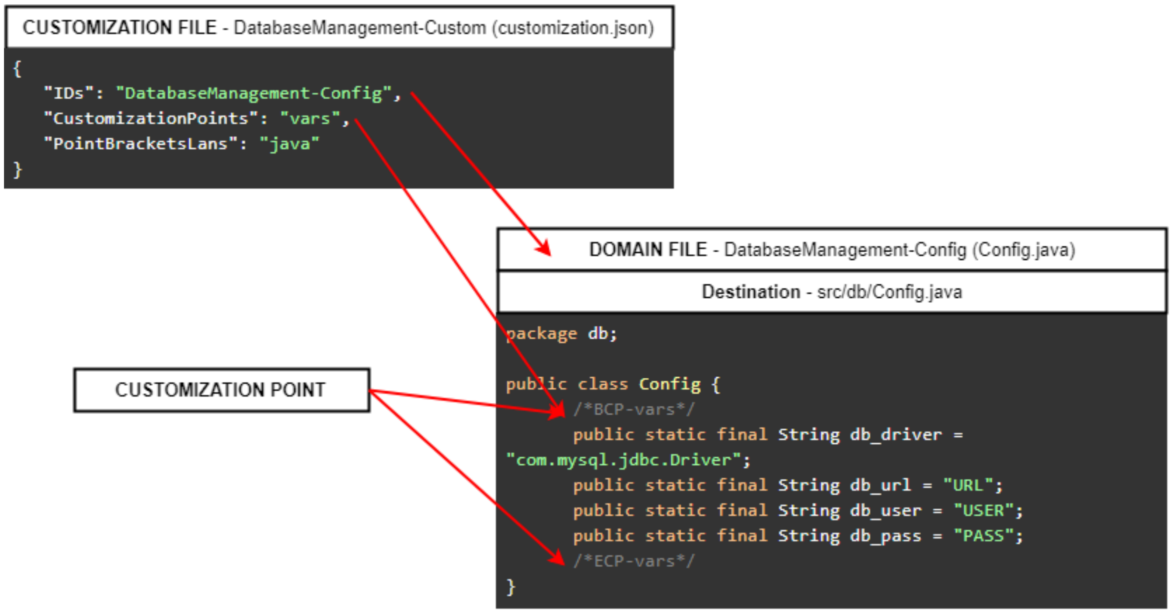


Figure 4-5: An example of the connection between a domain file, a customization file, and a customization point


CUSTOMIZATION FILE - DatabaseManagement-Custom (customization.json)
DOMAIN FILE - DatabaseManagement-Config (Config.java)
Destination - src/db/Config.java
CUSTOMIZATION POINT

```
{
   "IDs": "DatabaseManagement-Config",
   "CustomizationPoints": "vars",
   "PointBracketsLans": "java"
}
```

```
package db;

public class Config {
        /*BCP-vars*/
        public static final String db_driver =
"com.mysql.jdbc.Driver";
        public static final String db_url = "URL";
        public static final String db_user = "USER";
        public static final String db_pass = "PASS";
        /*ECP-vars*/
}
```

4.4.1 Customization point

In FragOP, we use annotations to support product customization. In this approach, we call them customization points. A **customization point** is an annotation (a very simple mark) that specifies a "point" in which a domain file should be customized.

Customization point shape is very similar to fragmentation point shape, the main difference is that a customization point shape should contain a begin part (`BCP`) and an end part (`ECP`). Listing 4-7 shows the customization point shape. The code to be customized at the application level should be placed in the middle of both `BCP` and `ECP` parts.

Listing 4-7: Customization point shape

```
LanguageCommentBlock<BCP>-<PointID>LanguageCommentBlock
LanguageCommentBlock<ECP>-<PointID>LanguageCommentBlock
```

We decided to use annotations once again because it allows developers to define very specific customization locations, and it can be used for many kinds of software assets (including non-object-oriented assets). The customization points have the following characteristics:

- **Customization points are different from fragmentation points.** Customization points and fragmentation points are very similar, however, we decided to distinguish between the two elements. This is due to the fact that: (i) fragmentation points are used to specify points in which a domain file code can vary, and they are connected with fragments that modify the domain file code. The possible code variations are pre-defined inside the fragments' code (which are stored in the domain component pool). And these variations are executed in the product derivation activity (see Section 5.6). (ii) Customization points are used to specify points in which a domain file code should be customized. The code customizations are not pre-defined, because each product customization is unique, and it is customer-dependent. And these customizations must be manually applied by the SPL developer after the product derivation (see Section 5.7).
- **Customization points guide SPL developers in the customization activity.** Most of the SPL implementation approaches do not provide a product customization capability (such as CIDE, DeltaJ, Munge, Antenna, AspectJ, and AHEAD, among others). Nevertheless, the literature presents different customization strategies. Kim *et al.* (2005) propose three strategies: selection, plug-in, and external profile technique. However, these strategies only work with interface classes and are not applied in SPL scenarios. Rabiser *et al.* (2009) suggest a decision-oriented software product line approach to support the end-user personalization of a system based on their needs. However, the personalization is limited to the elements that the decision model supports. Pleuss *et al.* (2012) propose the use of abstract UI models to bridge the gap between automated, traceable product derivation and customized high-quality user interfaces. However, it requires to create abstract UI models with all possible scenarios, and this is only applied to user interfaces. Other strategies include inheritance, overloading, dynamic class loading, but again not all assets are object-oriented. Finally, in FragOP we decided to avoid the use models, decisions and object elements to support the product customization. This is because we wanted to support the customization of most kinds of files (generic customization), and we know that most product customizations are unique. However, even the most complete model will not fulfill all customer customizations. In this case, we decided to use customization points to indicate sections inside the domain files that should be customized. Later, the SPL developer should manually customize these

sections. Although the customization is manual, we provide a way to guide the developer in the customization activity (see Section 5.7).

In order to present an example of customization points in action, the ClothingStores `DatabaseManagement-Config` (Config.java) file is detailed in full. Listing 4-1 shows the Config.java code which is a configuration file that contains four variables which allow communication with the database engine. As a domain file, these variables present sample values, however, for a final product the value of each variable must be changed. As a consequence, we refined the `DatabaseManagement-Config` (Config.java) file with the inclusion of a customization point (see Listing 4-8). Later, the SPL developer will be able to customize this file in order to establish the real values for each variable (described in Section 5.7).

Listing 4-8: Refined DatabaseManagement-Config (Config.java) file source code

DatabaseManagement-Config (Config.java)

```
package db;

public class Config {
       /*BCP-vars*/
       public static final String db_driver = "com.mysql.jdbc.Driver";
       public static final String db_url = "URL";
       public static final String db_user = "USER";
       public static final String db_pass = "PASS";
       /*ECP-vars*/
}
```

4.4.2 Customization file

A **customization file** is a file which specifies the domain files (for the current domain component) that should be customized. Only one customization file is allowed per domain component, its filename must be customization.json, and it must respect the shape presented in Listing 4-9 and explained below.

Listing 4-9: Customization file shape

```
{
   "IDs": ["FileID1", "FileID2", "..."],
   "CustomizationPoints": ["PointID1", "PointID2", "..."],
   "PointBracketsLangs": ["language1", "language2", "..."]
}
```

IDs: <*FileID1*, *FileID2, …*>. This represents the domain files to be customized.

CustomizationPoints: <*pointID1*, *pointID2, …*> (Optional). `PointIDs` are unique texts which serve to identify customization points.

PointBracketsLangs: <*language1*, *language2, …*> (Optional). This specifies the comment bracket `languages` in which the customization points are defined. For example, PHP, HTML, and Java.

The customization points and the point brackets languages are optional, this way a customization file is able to specify entire domain files that must be customized (replaced) or specific customization points to be customized. Customizing an entire domain file is useful when it is not possible to include customization points. For example, when there is a domain file such as a default logo, that must be customized with the real client company logo.

As shown in the Listing 4-8, the `DatabaseManagement` component contains the `DatabaseManagement-Config` (Config.java) file which was refined with a customization point. It means that the SPL developer must create a customization file inside the `DatabaseManagement` component to specify the customization points for the current component. Listing 4-10 shows the `DatabaseManagement-Custom` (customization.json) customization file source code. This code indicates that for the current component (`DatabaseManagement`) one customization point (`vars`) that belongs to the `DatabaseManagement-Config` file has been defined. The complete relationship between the domain file, customization point, and customization file can be found in Figure 4-5.

Listing 4-10: DatabaseManagement-Custom (customization.json) file source code

DatabaseManagement-Custom (customization.json)

```
{
   "CustomizationPoints": "vars",
   "PointBracketsLangs": "java",
   "IDs": "DatabaseManagement-Config"
}
```

The component file customization will be executed and applied later in the product customization activity which is described in Section 5.7.

Finally, it is important to highlight that customization files and customization points are very useful for simple customizations, such as parametrizing variables, changing a default text, or replacing an image file, nevertheless, complex customization like the creation of a new component must be applied manually by the SPL developer.

4.5 Summary

This chapter presented the FragOP fundamental elements: (i) domain components (i) domain files, (iii) fragmentation points, (iv) fragments, (v) customization points, and (vi) customization files. The six FragOP fundamental elements are summarized below:

- Domain components are folders that store domain files, fragments and customization files.
- Domain files represent the files that could be reused for the development of multiple SPL products (such as HTML, Python, JavaScript, Java, and JSP files).
- Fragmentation points are annotations (simple marks) that specify “points” in which a domain file can be altered.
- Fragments are a special type of file in which the SPL developers specify code alterations to the domain files.
- Customization points are annotations (simple mark) that specify “points” in which a domain file should be customized.
- A customization file specifies the domain files to be customized and the customization points of the domain files of the current domain component.

We explained in this chapter the decisions to create each of the previous FragOP’s fundamental elements, the functionalities from the literature that were taken into account to improve the FragOP fundamental elements, and the main characteristics of each FragOP fundamental element.

This chapter also presented the two FragOP main capabilities, assembling and customization; and we used the running example to provide a practical way of understanding both the FragOP fundamental elements and the two FragOP main capabilities. The two FragOP main capabilities are summarized below:

- The assembling capability provides an effective and generic way to support software variability. The use of domain files, fragmentation points, and fragments, allow specifying variation points inside most software language files (because it only requires the use of language comment blocks or regular expressions). And even, if it is not possible to modify the file source code to include the variation points (*i.e.*, an image or PDF file), the FragOP fragments allow replacing an entire file.

- The customization capability provides an effective and generic way to support component customization. The use of domain files, customization points, and customization files, allow specifing customization points inside most software language files (because it only requires the use of language comment blocks, or regular expressions). And even, if it is not possible to modify the file source code to include the customization points (*i.e.*, an image or PDF file), the FragOP customization files allow customizing an entire file.

To conclude, the definition of these six elements allows us to answer RQ1 because these elements specify the way in which the SPL components should be implemented. The next chapter will describe each activity in the FragOP process and will provide a practical way to implement an SPL with the FragOP approach.

5. FragOP process

The FragOP process provides a course of action for implementing an SPL using the FragOP approach. This process was designed following the common SPLE structure. The FragOP process contains eight main activities (i) modeling PL requirements, (ii) modeling domain components, (iii) implementing domain components, (iv) binding domain requirements and domain components, (v) configuring products, (vi) deriving products, (vii) customizing products, and (viii) verifying products. These activities describe from very early SPLE processes such as variability modeling to later SPLE processes such as product derivation.

In this chapter, we present each of the previous activities in detail. For each, (i) we summarize the theory from the SPLE literature and discuss how it should be applied using the FragOP approach, (ii) we show how VariaMos supports it, and (iii) we carried out a demonstration in the running example and so exemplify the use of this approach in a realistic scenario.

5.1 Modeling product line requirements

The definitions of a requirement according to the Institute of Electrical and Electronics Engineers (IEEE, 1990) are:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
3. A documented representation of a condition or capability as in (1) or (2).

Requirements are often classified into functional requirements and non-functional requirements. **Functional requirements** specify a function that a system must be able to perform. **Non-functional requirements** specify quality attributes (such as usability, reliability, performance, and supportability) and constraints on a system (such as time, budget, hardware, and material).

Establishing the product line requirements is a difficult task that involves the participation of different stakeholders and should be done through a requirement engineering process. This process consists of three steps:

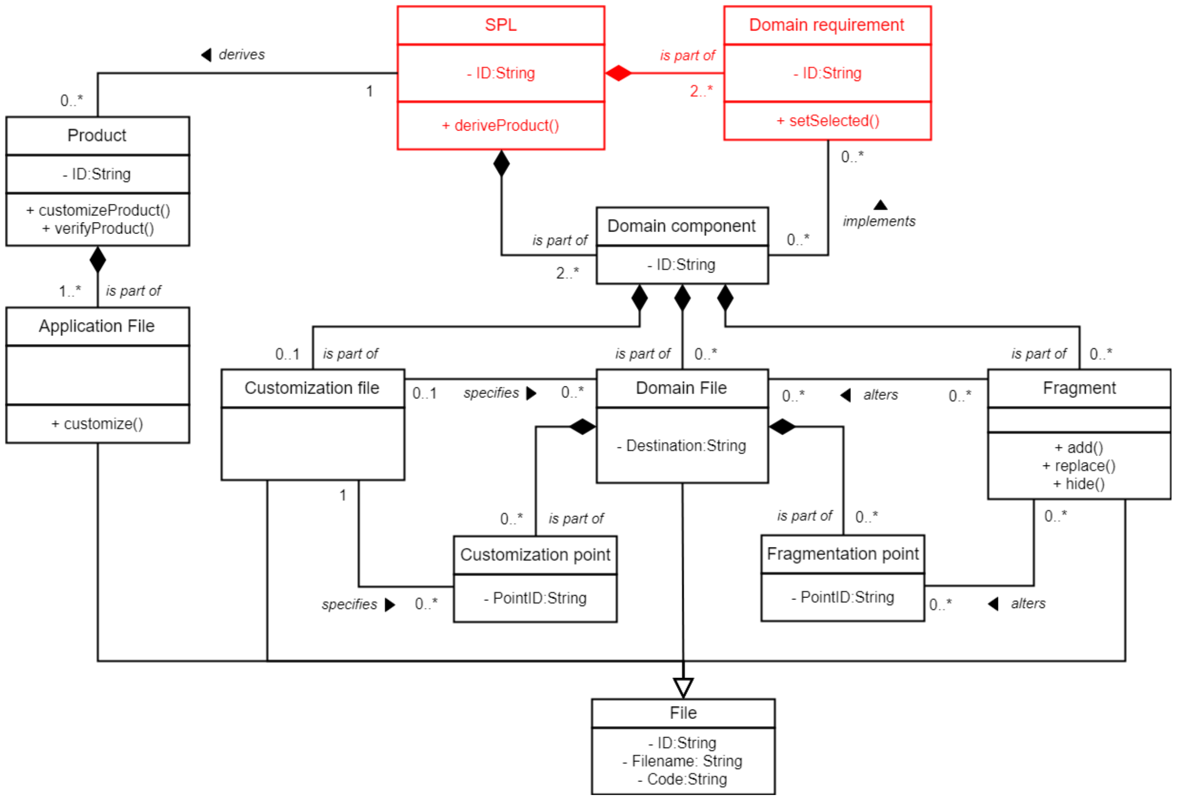
- **Requirements elicitation** consists of finding and identifying the relevant PL requirements. First, it is important to identify the different stakeholders (such as final users, employees, software developers, and executives) that could be interested in a product (relevant to the PL domain). Second, the analyst should identify the information sources (such as stakeholders, documents, laws, and social networks) that are relevant for defining the PL requirements. Third, the analyst starts with the requirement elicitation, which consists of finding and identifying the PL requirement based on the previous information sources. There are different strategies for the PL requirement elicitation, one of them consists of analyzing the market segment relevant to the PL domain. Then, the existing products should be analyzed in order to discover PL requirements and finally define the common and variable requirements. And fourth, the requirements are written with the use of a template or specification.
- **Requirements analysis** consists of a formal documentation of the PL requirements. Commonly, PL analysts use variability models, such as feature models, goal models, and the Orthogonal Variability Model (OVM), to represent the PL requirements. Each model has its own advantages and disadvantages. Feature models are the most used; they represent the PL variability through a tree structure in which the features form the nodes of the tree, and the arcs and groupings of features represent feature variability (Beuche & Dalgarno, 2007). Feature models are commonly used to represent functional requirements, however, they are not well adapted to represent non-functional requirements and their particular relationships with the functional requirements. Goal models are graphs where a goal node is refined into several subgoal nodes (Yu *et al.*, 2008). In this approach, functional requirements are modeled by *goals*, and quality attributes (non-functional

requirements) are modeled as *softgoals*. The *softgoals* have a multi-valued label to indicate the degree of its satisfaction: fully satisfied (FS), partially satisfied (PS), fully denied (FD) or partially denied (PD). Thus, for instance, a security requirement will not have a true or false value but a certain degree of satisfaction.

- **Requirements management** consists of documenting, planning and coordinating the PL requirements. Here the requirements are stored, and traceability is defined.

The FragOP approach requires the domain requirements to be elicited and then formally specified within a modeling language. Figure 5-1 shows that in FragOP an SPL contains two or more domain requirements. There should be a minimum of two domain requirements because it guarantees at least two configurations of two different products. It is also important to highlight that this approach does not restrict the way in which the domain requirements are modeled. It means that the SPL developer can use feature models, OVM, and goal models, among others.

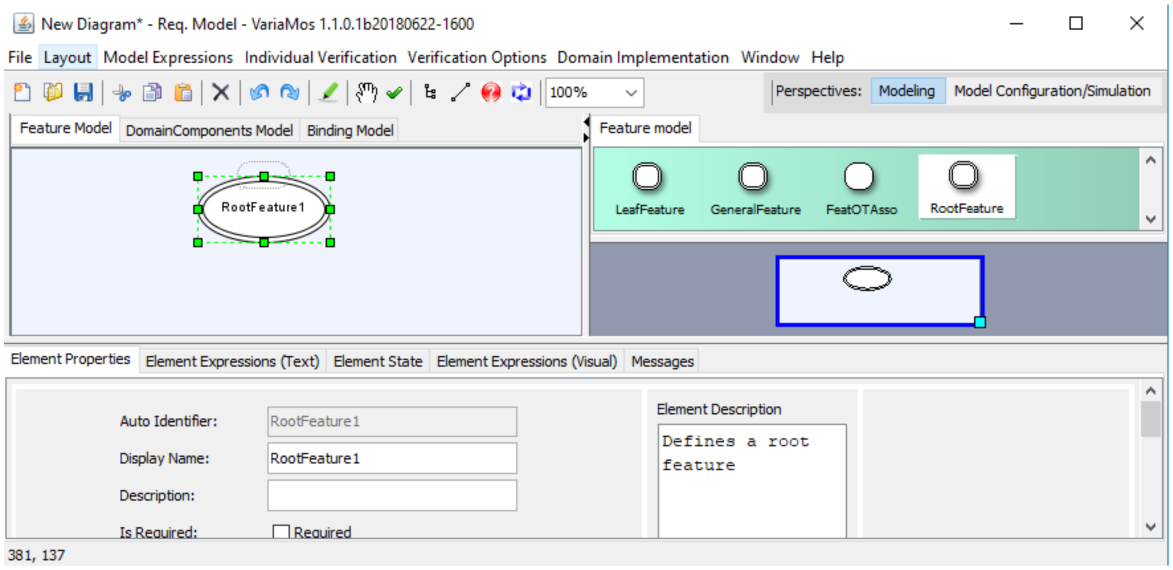
Figure 5-1: FragOP metamodel highlighting the concepts of SPL and domain requirement



5.1.1 VariaMos support

Modeling product line requirements is an activity that is fully supported by VariaMos. VariaMos allows for domain requirements to be specified in the form of a “Feature model”. After downloading and running VariaMos, the user should select a “Component-based project”. This kind of project allows navigating between three different views: (i) feature model, (ii) domain component model, and (iii) binding model. Figure 5-2 shows the feature model view. Here the SPL developer is able to graphically represent the domain requirements through the feature model. VariaMos also provides some verification options, such as (i) more than one root element, (ii) child elements without parents, (iii) dead elements, and (iv) false optional elements. A complete tutorial on how to use VariaMos to represent feature models, configure and verify them can be found online (Correa, 2018).

Figure 5-2: VariaMos “feature model” view main elements



Auto Identifier: RootFeature1
Display Name: RootFeature1
Description:
Is Required: ☐ Required

```
Defines a root
feature
```

Currently, the “Component-based project” only supports feature models, however, VariaMos offers the possibility of creating new models, such as goals models or OVM, and even offers the possibility of creating user custom models.

Other current approaches also support the domain requirements specification, for example:

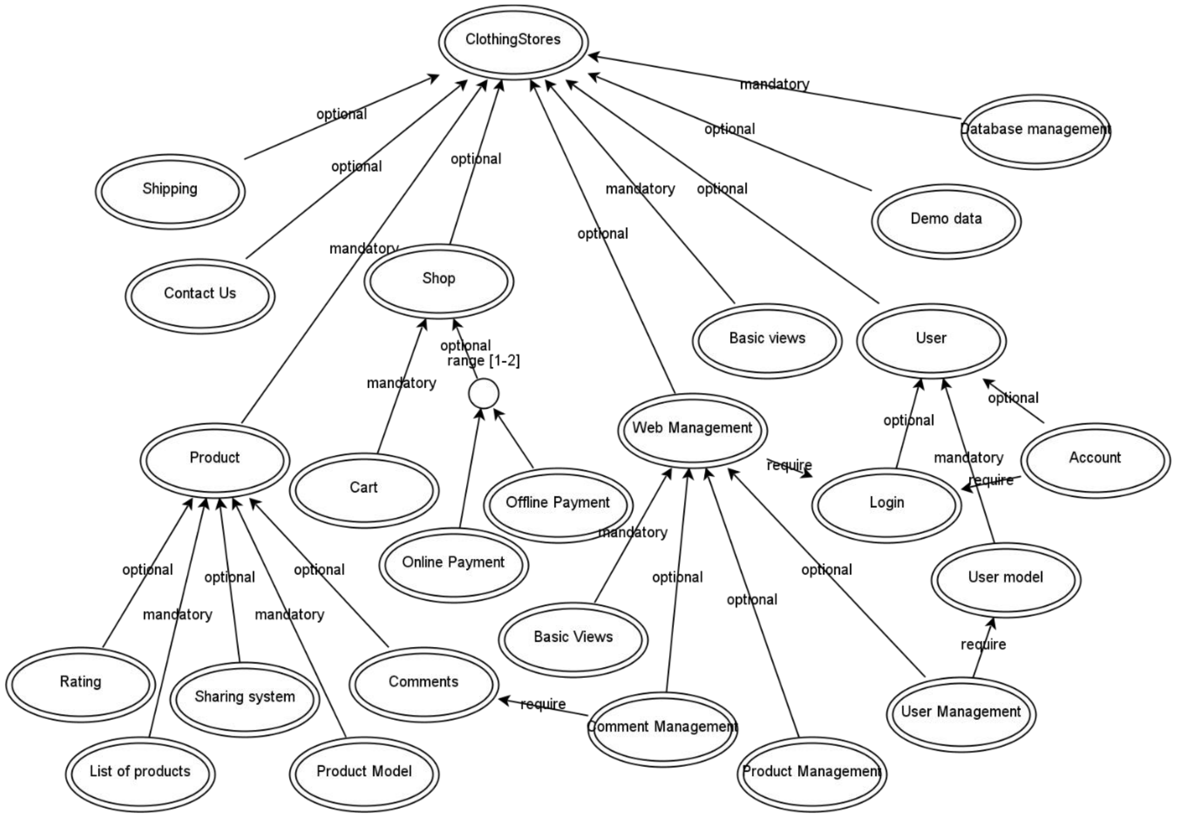
- **FeatureIDE with AHEAD (FOP) or FeatureHouse (FOP) or AspectJ (AOP).** After the creation of a FeatureIDE project, a model.xml file is automatically created. When the model.xml file is opened, a graphical editor is presented. There the SPL developer can edit the SPL feature model.

- **DeltaJ (DOP).** This approach provides a .spl file, in this file, there is a variable called “Features” where the SPL developer types the feature names (there is no graphical support).
- **CIDE (annotative).** This approach provides three ways to execute feature modeling: (i) the definition of a list of features, which are all optional and unrelated, (ii) a graphical feature model editor from FeatureIDE, and (iii) a connector to the pure::variants tool, where the SPL developer is able to use pure::variants feature models.

ClothingStores SPL

We used the VariaMos feature model view to define the 25 features of the ClothingStores example (see Figure 2-1). A developer can create those features manually or load a pre-developed model (Correa, 2018) which contains the complete feature model as shown in Figure 5-3.

Figure 5-3: Feature model of the complete ClothingStores running example (VariaMos)

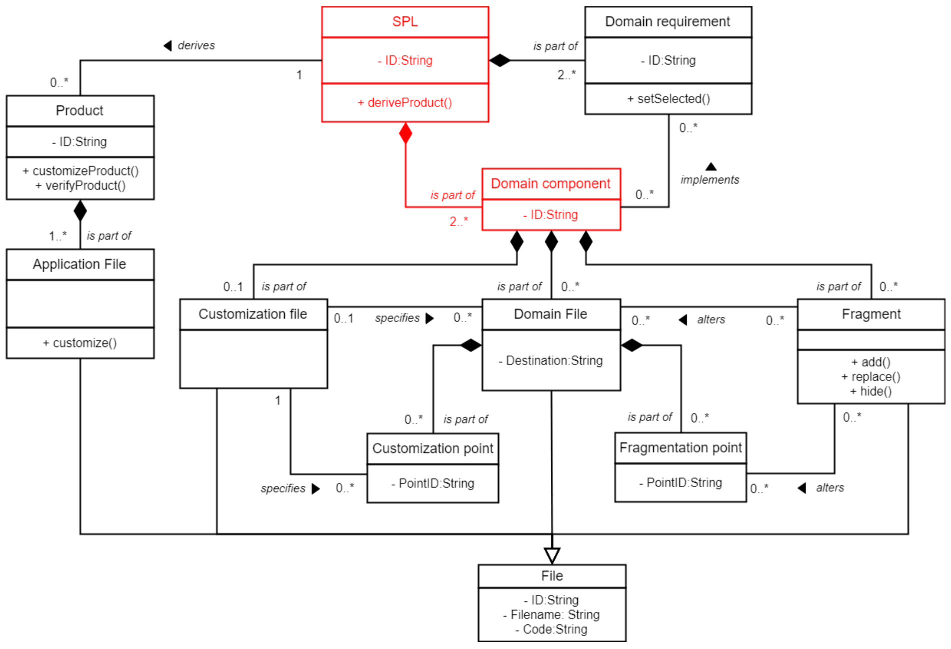


5.2 Modeling domain components

Modeling domain components is not a typical activity in most of the current SPL implementation approaches. Many of the current approaches have a direct link between the features (defined in the feature model) and the component files that operationalize them, thereby, there is no need for a component model. On the other hand, a domain component model allows the software components, their files, and their relationships to be represented. The SPL domain component model provides a general insight for software architects and software developers into the SPL domain system. Therefore, it allows having a complete separation between the domain requirements (problem space) and the domain components (solution space).

In FragOP, an SPL contains two or more domain components (see Figure 5-4), which guarantee the derivation of at least two different products. The FragOP domain component model also serves to define: (i) the links between domain components and their files, (ii) the domain file destinations, (iii) the domain component IDs and file IDs, and (iv) a future connection between the domain requirements and the domain components (see Section 5.4). Later, each domain component file must be operationalized with its code implementation (see Section 5.3).

Figure 5-4: FragOP metamodel highlighting the concepts of SPL and domain component



5.2.1 VariaMos support

Before starting the domain component modeling activity, the SPL developer has to define the domain components and files that the SPL will contain. In this case, we used the ClothingStores requirements (see Table 2-1), and we defined a list of 20 domain reusable components. Each domain component is also connected to its own list of files which operationalize it as shown in Table 5-1. The first element in the table is the domain component identifier, which is a simple text with the domain component name. This text is going to represent a real folder, so the PL developer should avoid spaces or strange symbols. Table 5-1 also records the following information for each file: (i) `File ID`: we suggest that it be created as a combination of the domain component identifier plus a minus symbol ("-"), plus the file identifier; (ii) `Filename`: the real filename including its extension; and (iii) `Destination`: the destination in which the file is going to be derived at the end of the FragOP process, based on the SPL basic project folder structure (see Figure 2-3).

Table 5-1: ClothingStores list of domain components and their files

#	Component ID	File ID	Filename	Destination
C01	BasicViewsGeneral	BasicViewsGeneral-Index	index.jsp	WebContent/views/index.jsp
		BasicViewsGeneral-Banner	banner.jpg	WebContent/assets/img/banner.jpg
		BasicViewsGeneral-Bootstrap	bootstrap.min.css	WebContent/assets/css/bootstrap.min.css
		BasicViewsGeneral-Bootstrap2	bootstrap.min.js	WebContent/assets/js/bootstrap.min.js
		BasicViewsGeneral-Header	header.jsp	WebContent/views/header.jsp
		BasicViewsGeneral-Footer	footer.jsp	WebContent/views/footer.jsp
		BasicViewsGeneral-Home	Home.java	src/controllers/Home.java
		BasicViewsGeneral-Popper	popper.js	WebContent/assets/js/popper.js
		BasicViewsGeneral-Style	style.css	WebContent/assets/css/style.css
		BasicViewsGeneral-JQuery	jquery-3.2.1.min.js	WebContent/assets/js/jquery-3.2.1.min.js
		BasicViewsGeneral-Custom	customization.json	
C02	ContactUs	ContactUs-Contact	Contact.java	src/controllers/Contact.java
		ContactUs-View	contact.jsp	WebContent/views/contact.jsp
		ContactUs-AlterHeader	alterHeader.frag	
		ContactUs-Custom	customization.json	
C03	Shipping	Shipping-Shipping	Shipping.java	src/controllers/Shipping.java

		Shipping-View	shipping.jsp	WebContent/views/shipping.jsp
		Shipping-AlterHeader	alterHeader.frag	
		Shipping-Custom	customization.json	
C04	DatabaseManagement	DatabaseManagement-DB	DB.java	src/db/DB.java
		DatabaseManagement-Config	Config.java	src/db/Config.java
		DatabaseManagement-MainSQL	main.sql	main.sql
		DatabaseManagement-Custom	customization.json	
C05	DemoData	DemoData-DemoSQL	demo.sql	demo.sql
C06	ProductModel	ProductModel-Product	Product.java	src/models/Product.java
		ProductModel-ProductDAO	ProductDAO.java	src/models/ProductDAO.java
		ProductModel-AlterMainSQL	alterMainSQL.frag	
C07	ListOfProducts	ListOfProducts-ListProducts	ListProducts.java	src/controllers/ListProducts.java
		ListOfProducts-View	listproducts.jsp	WebContent/views/listproducts.jsp
		ListOfProducts-OneProduct	oneproduct.jsp	WebContent/views/oneproduct.jsp
		ListOfProducts-AlterStyle	alterStyle.frag	
		ListOfProducts-AlterHeader	alterHeader.frag	
C08	Comments	Comments-Comment	Comment.java	src/models/Comment.java
		Comments-CommentDAO	CommentDAO.java	src/models/CommentDAO.java
		Comments-AddComment	AddComment.java	src/controllers/AddComment.java
		Comments-AlterDemoSQL	alterDemoSQL.frag	
		Comments-AlterMainSQL	alterMainSQL.frag	
		Comments-AlterListProducts	alterListProducts.frag	
		Comments-AlterOneProduct	alterOneProduct.frag	
C09	SharingSystem	SharingSystem-Fb	fb.png	WebContent/assets/img/fb.png
		SharingSystem-Twitter	twitter.png	WebContent/assets/img/twitter.png
		SharingSystem-AlterOneProduct	alterOneProduct.frag	
C10	Rating	Rating-AlterListProducts	alterListProducts.frag	
		Rating-AlterMainSQL	alterMainSQL.frag	
		Rating-AlterManageProducts	alterManageProducts.frag	
		Rating-AlterOneProduct	alterOneProduct.frag	
		Rating-AlterProduct	alterProduct.frag	
		Rating-AlterProductDAO	alterProductDAO.frag	
		Rating-AlterStyle.frag	alterStyle.frag	
C11	UserModel	UserModel-User	User.java	src/models/User.java
		UserModel-UserDAO	UserDAO.java	src/models/UserDAO.java
		UserModel-AlterDemoSQL	alterDemoSQL.frag	
		UserModel-AlterMainSQL	alterMainSQL.frag	
C12	Account	Account-Account	Account.java	src/controllers/Account.java
		Account-AccountView	account.jsp	WebContent/views/account.jsp

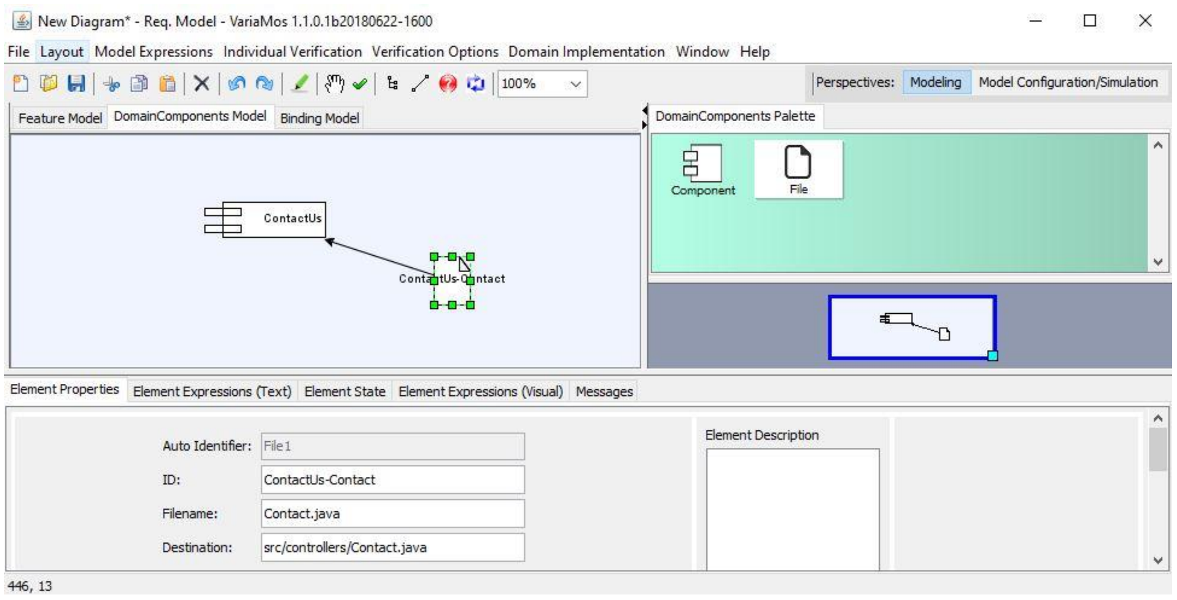
		Account-Img	user.png	WebContent/assets/img/user.png
		Account-AlterHeader	alterHeader.frag	
C13	Login	Login-Login	Login.java	src/controllers/Login.java
		Login-LoginForm	login_form.jsp	WebContent/views/login_form.jsp
		Login-AlterAccount	alterAccount.frag	
		Login-AlterAdmin	alterAdmin.frag	
		Login-AlterHeader	alterHeader.frag	
C14	Cart	Cart-Cart	Cart.java	src/controllers/Cart.java
		Cart-CartView	cart.jsp	WebContent/views/cart.jsp
		Cart-AlterProductDAO	alterProductDAO.frag	
		Cart-AlterOneProduct	alterOneProduct.frag	
		Cart-AlterHeader.frag	alterHeader.frag	
C15	OnlinePayment	OnlinePayment-AlterCart	alterCart.frag	
C16	OfflinePayment	OfflinePayment-AlterCart	alterCart.frag	
C17	BasicViewsAdmin	BasicViewsAdmin-Home	Home.java	src/controllers/admin/Home.java
		BasicViewsAdmin-Index	index.jsp	WebContent/views/admin/index.jsp
		BasicViewsAdmin-Header	header.jsp	WebContent/views/admin/header.jsp
		BasicViewsAdmin-Custom	customization.json	
C18	ProductManagement	ProductManagement-ManageProducts	ManageProducts.java	src/controllers/admin/ManageProducts.java
		ProductManagement-View	products.jsp	WebContent/views/admin/products.jsp
		ProductManagement-AlterAdminHeader	alterAdminHeader.frag	
		ProductManagement-AlterProductDAO	alterProductDAO.frag	
C19	UserManagement	UserManagement-ManageUsers	ManageUsers.java	src/controllers/admin/ManageUsers.java
		UserManagement-View	users.jsp	WebContent/views/admin/users.jsp
		UserManagement-AlterAdminHeader	alterAdminHeader.frag	
		UserManagement-AlterUserDAO	alterUserDAO.frag	
C20	CommentManagement	CommentManagement-ManageComment	ManageComment.java	src/controllers/admin/ManageComment.java
		CommentManagement-View	comments.jsp	WebContent/views/admin/comments.jsp
		CommentManagement-AlterCommentDAO	alterCommentDAO.frag	
		CommentManagement-AlterAdminHeader	alterAdminHeader.frag	

Table 5-1 also highlights three different types of files: (i) **blue background** refers to domain files (see Section 4.2), (ii) **white background** refers to fragments (see Section 4.3.2), and

(iii) **red background** refers to customization files (see Section 4.4.2). Destination information must not be provided for fragments and customization files, because those files will not be included as part of the final derived products.

After developing the list of domain components and their files, the SPL developer should use VariaMos to create the domain component model. This activity requires the SPL developer to navigate to the “Domain component model” view (see Figure 5-5). The process consists of graphically representing the domain components and their files based on the previous list (see Table 5-1).

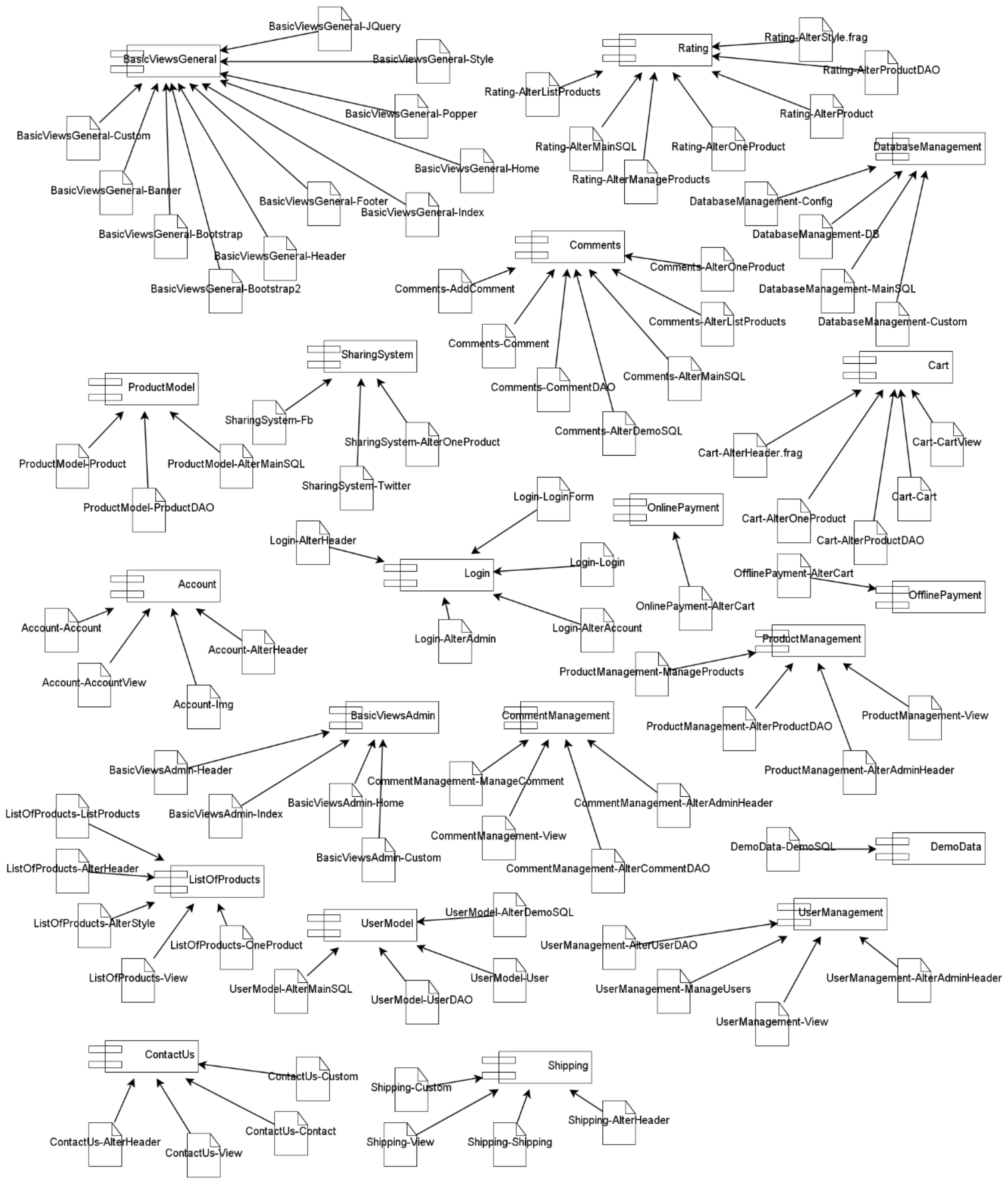
Figure 5-5: VariaMos “Domain component model” view main elements


New Diagram* - Req. Model - VariaMos 1.1.0.1b20180622-1600
File Layout Model Expressions Individual Verification Verification Options Domain Implementation Window Help
100%
Perspectives: Modeling Model Configuration/Simulation
Feature Model DomainComponents Model Binding Model
DomainComponents Palette
Component
File
ContactUs
ContactUs-Contact
Element Properties Element Expressions (Text) Element State Element Expressions (Visual) Messages
Element Description
446, 13

Auto Identifier: File1
ID: ContactUs-Contact
Filename: Contact.java
Destination: src/controllers/Contact.java

Finally, we use the VariaMos component model view to represent the ClothingStores domain components and files. An SPL developer can create these domain components and files manually (by using the information in Table 5-1) or load a pre-developed model (Correa, 2018) which contains the ClothingStores complete domain component model as shown in Figure 5-6.

Figure 5-6: ClothingStores complete domain component model



5.3 Implementing domain components

Implementing the reusable domain component is one of the most important tasks in an SPLE. These components will serve as the base from which any SPL product will be derived. As explained in Section 4.1, depending on the selected approach, this activity will require to codify object classes (OOP), aspects (AOP), delta modules (DOP), web services, agents, and features (FOP), among others. Therefore, any reusable asset, such as images, scripts, HTML views, among others need to be included.

Additionally, it is important to highlight that the domain components are not always designed and implemented from scratch. Domain components could also be acquired or rented through external companies, like with commercial off-the-shelf (COTS) components (Lago *et al.*, 2004); inherited from previous developments; or outsourced through third party companies.

In FragOP, this activity requires that SPL developers implement the (i) domain components, (ii) domain files, (iii) fragmentation points, (iv) fragments, (v) customization points, and (vi) customization files. We have already explained how to implement these elements in Chapter 4, where we showed some of the differences between the FragOP approach and other similar approaches.

Finally, these components have to be verified and validated after the domain component implementation. There are different kinds of tests that could be applied, such as (i) unit tests, which allows a class, component or a piece of code to be tested independently of the entire system. (ii) integration tests, which allows the integration of two or more components to be tested. And (iii) regression tests, which are carried out when changes are made to components or pieces of code that have already been tested (Neto *et al.*, 2011).

5.3.1 IDE support

To start the domain components implementation, PL developers must create a domain component pool directory to store the corresponding components and files. In FragOP, the domain component pool directory must be consistent with the Component Identifiers defined in Table 5-1. Figure 5-7-A shows an example of the default folder structure, which can be used as a template to create the final folder structure (*e.g.*, for the ClothingStores SPL) as can the one presented in Figure 5-7-B.

We also suggest connecting the component pool directory with a web-based version control repository, such as Bitbucket[1] or GitHub[2], to manage the domain component evolution and traceability.

Figure 5-7: Component pool folders and files structure

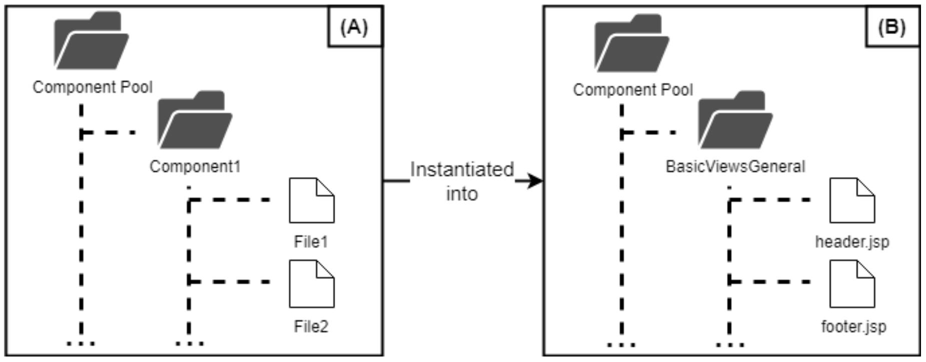
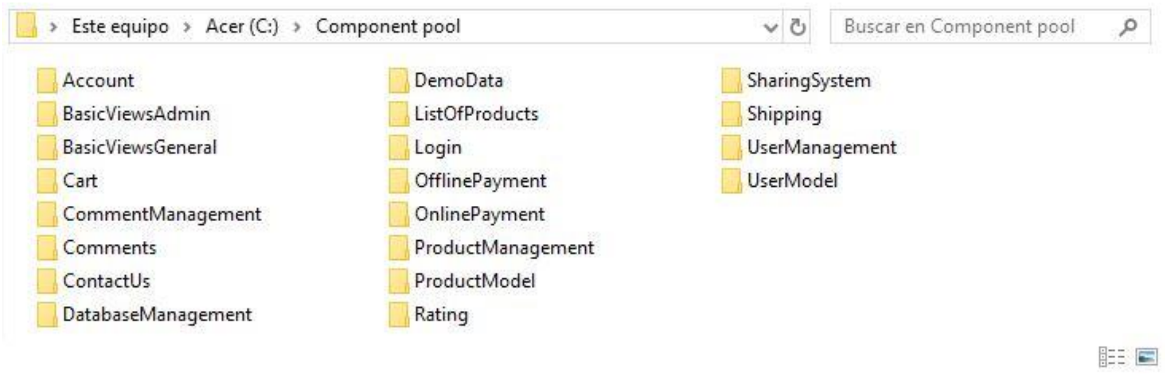


Figure 5-8 shows the ClothingStores component pool folder structure based on the 20 domain components defined in Table 5-1.

Figure 5-8: Clothing stores component pool folder



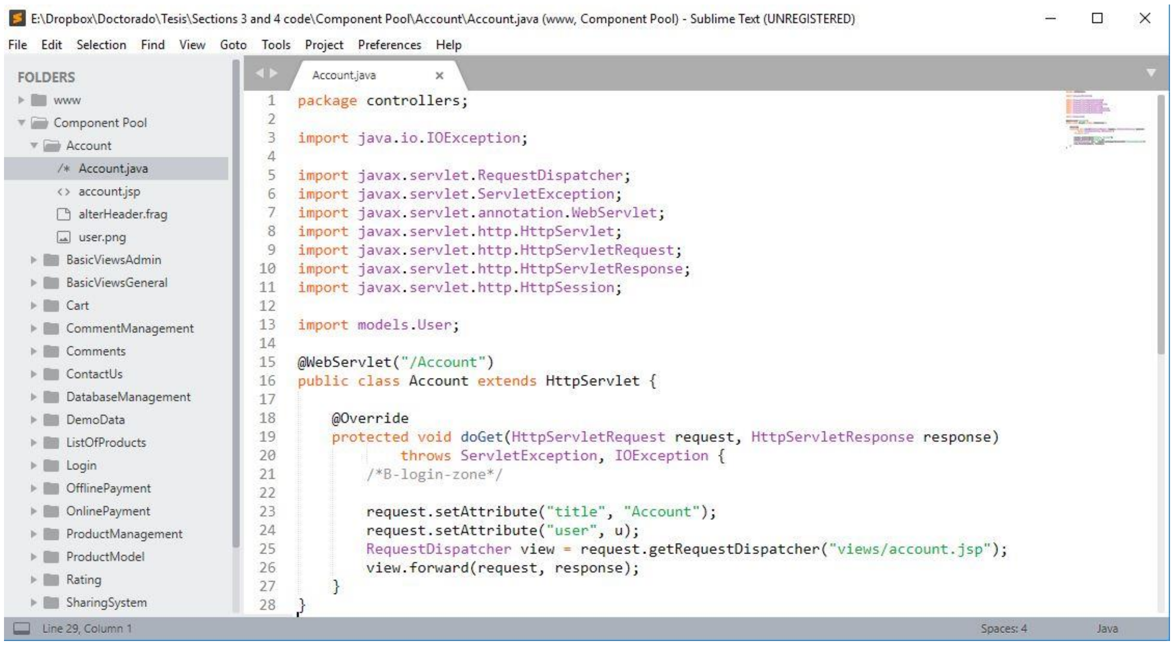
Account
BasicViewsAdmin
BasicViewsGeneral
Cart
CommentManagement
Comments
ContactUs
DatabaseManagement
DemoData
ListOfProducts
Login
OfflinePayment
OnlinePayment
ProductManagement
ProductModel
Rating
SharingSystem
Shipping
UserManagement
UserModel

After the definition of the component pool, we recommend selecting a preferred IDE to develop the component file source code such as Sublime[3], IntelliJ[4], NetBeans[5], and Eclipse[6], among others. Figure 5-9 shows the example of Sublime being used to create the

[1] https://bitbucket.org/
[2] https://github.com/
[3] https://www.sublimetext.com/
[4] https://www.jetbrains.com/idea/
[5] https://netbeans.org/projects/www/
[6] https://www.eclipse.org/

component files. Finally, the PL developer has to codify all of the domain component files, fragments, fragmentation points, customization points, and customization files. Additionally, we created an online folder (Correa, 2018) which contains the complete domain component file source codes.

Figure 5-9: Sublime IDE with an example of a component file development



```java
package controllers;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import models.User;

@WebServlet("/Account")
public class Account extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        /*B-login-zone*/

        request.setAttribute("title", "Account");
        request.setAttribute("user", u);
        RequestDispatcher view = request.getRequestDispatcher("views/account.jsp");
        view.forward(request, response);
    }
}
```

Note: most of the SPL implementation approaches (such as AHEAD, FeatureHouse, DeltaJ, CIDE, and Antenna) are designed as Eclipse plugins. In this case, the SPL developers create a new Eclipse project and use Eclipse to develop and store the domain components.

5.4 Binding domain requirements and domain components

As seen in previous sections, domain requirements can be implemented by means of several approaches. The relationship between the domain requirements and their implementations (commonly domain components) is very important for the SPLE because it connects the work of the PL requirement engineers with the PL software developers and the component suppliers.

The relationship could be defined as a one-to-one relationship (for example, one domain component linked to one domain requirement). Nevertheless, the relationship could be complex when there are multiple domain components that satisfy a specific domain requirement, or multiple component suppliers.

There are some models and approaches that have been proposed for linking the domain requirements and the domain components:

- **Weaving models** are intermediary models which define the relationships between the variability model and the component model. The features are located on one side and the components on the other (Cetina *et al.*, 2013).
- **Mappings** are used to define complex connections between features from feature models and the software artefacts that are realizing those features (Heidenreich *et al.*, 2008). For example, in FeatureMapper, these mappings usually do not only contain mappings between features and software artefacts, but also between feature expressions and software artefacts, where a feature expression is a logical combination of features (e.g. `FeatureA` AND `FeatureB`).
- **Cardinality constraints** are used in models such as DOPLER (Dhungana *et al.*, 2011), and they connect the decision models (variability elements) with the asset models (domain components). These constraints group components and allow min and max values to be defined (cardinality) which are used to implement specific features. For example, a DOPLER decision `DOPLER_tools` (variability element) could be linked to three possible values `ConfigurationWizard`, `DecisionKing` and `ProjectKing` (components), and the user should select between 0 and 3 of the previous possible values (cardinality).

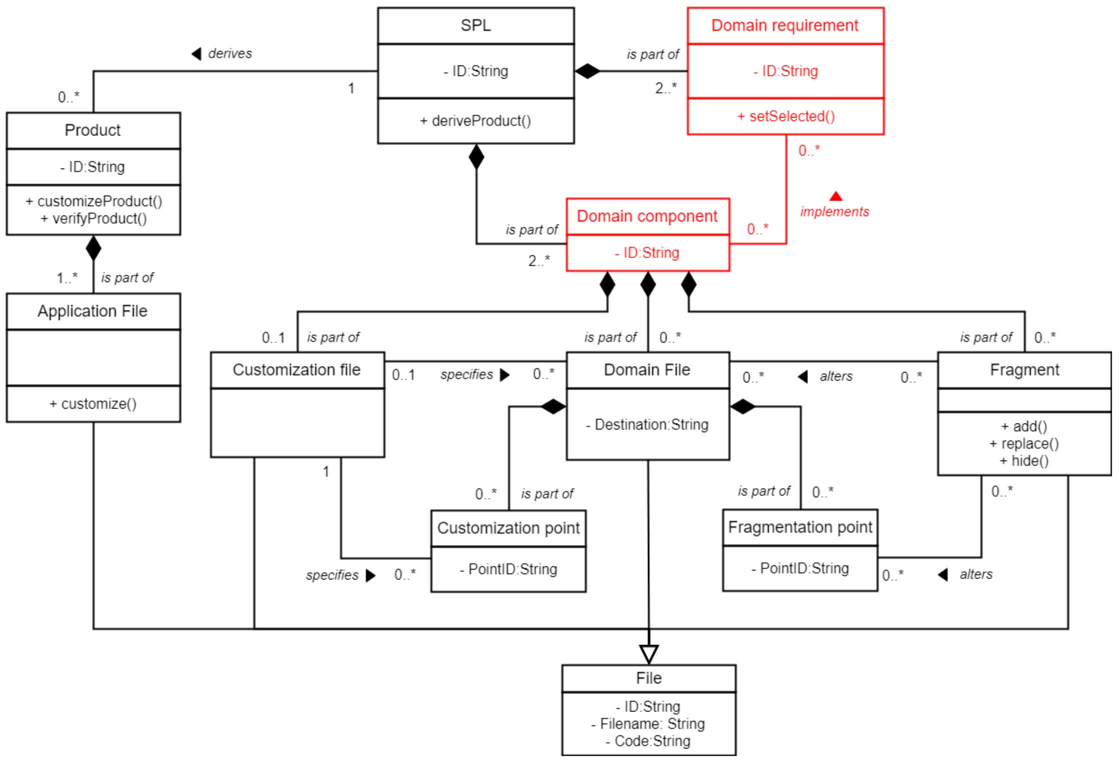
The reality is that most current approaches do not use graphical binding models, using instead a direct connection between the features and the components. For example:

- In DeltaJ (DOP), there is a file with an extension ".spl". That file includes a section called *partitions*, in which the user manually types the name of the features that are connected to the delta modules.
- In CIDE (annotative), the SPL developer is able to select the specific files or pieces of code that are related to specific features. He/she makes a right click on the specific piece of code or file and select the name of the feature that wants to link it with.

- In AspectJ with FeatureIDE (AOP), once a new feature is created in the feature model, the application generates a new file with the same name of the feature and with ".aj" extension.
- In Antenna with FeatureIDE (annotative), the file code that belongs to a specific feature is surrounded by IF and ENDIF statements. The IF statement must contain the name of the feature that links to it.
- Approaches such as AHEAD (FOP) and FeatureHouse (FOP) with FeatureIDE create a folder with the same name of each leaf feature; there the user stores the domain components that operationalize those features. Again, there is a direct link between the feature and the component.

In FragOP, this activity consists of developing a binding model between the domain requirements model (such as a feature model) and the domain component model. The binding model represents how the domain components operationalize the domain requirements. Figure 5-10 shows how the FragOP metamodel relates the domain requirements to the domain components in a many-to-many relationship.

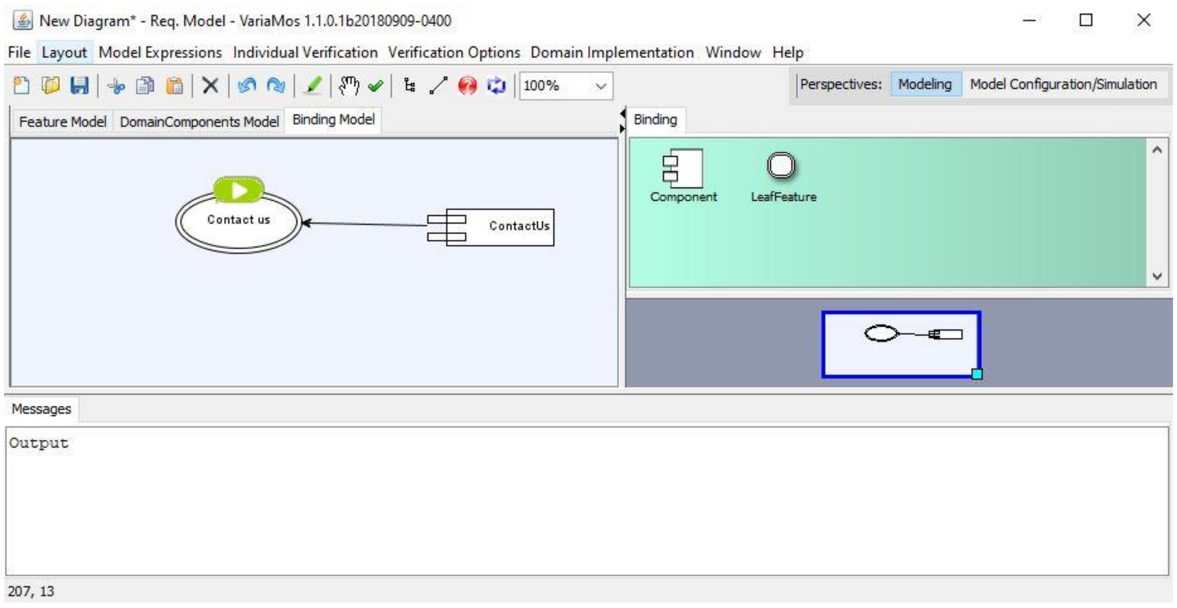
Figure 5-10: FragOP metamodel highlighting the domain requirement and domain component relationship



5.4.1 VariaMos support

Modeling the binding with VariaMos requires the developer to navigate to the third view "Binding model" (see Figure 5-11). This view presents the same distribution as the previous feature model and domain component model views. There are two elements components and leaf features which are automatically loaded from the two previous views. Here, the developer only needs to link the domain components with the leaf features that they operationalize.

Figure 5-11: VariaMos "Binding model" view main elements

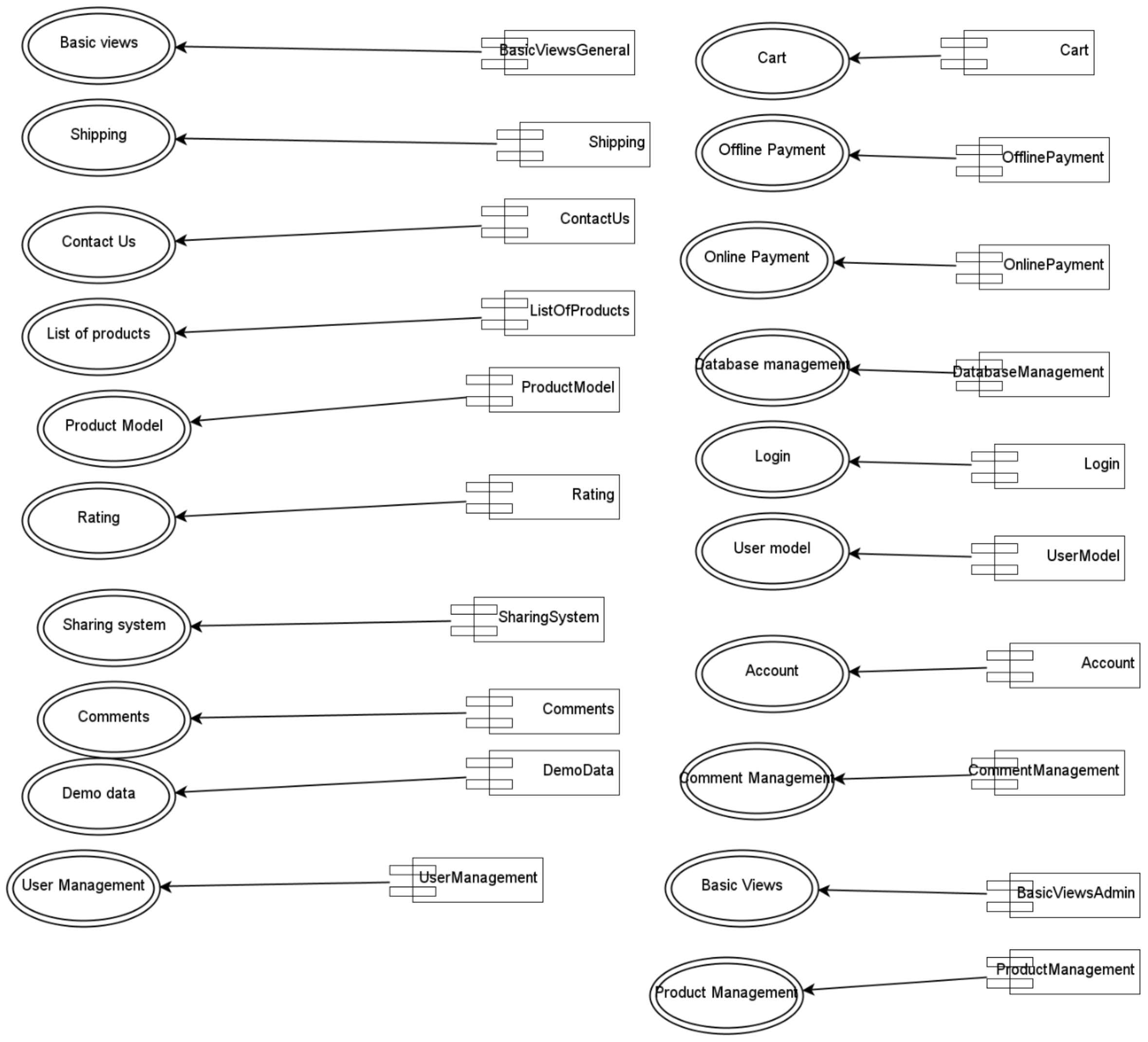

New Diagram* - Req. Model - VariaMos 1.1.0.1b20180909-0400
File Layout Model Expressions Individual Verification Verification Options Domain Implementation Window Help
100%
Perspectives: Modeling Model Configuration/Simulation
Feature Model DomainComponents Model Binding Model
Binding
Contact us
ContactUs
Component
LeafFeature
Messages
Output
207, 13

At present, VariaMos only allows a one-to-one binding relationship. Nevertheless, we plan to implement a constraint network (Lecoutre, 2009) to graphically represent more complex domain implementation relationships such as "Domain components C1 or C2, but not both, can be used to implement feature F".

ClothingStores SPL

The ClothingStores feature model (presented in Figure 5-3) and the ClothingStores component model (presented in Figure 5-6) are connected by means of the binding model. Figure 5-12 shows the ClothingStores complete binding model. The complete feature model, component model, and binding model can be found online (Correa, 2018).

Figure 5-12: ClothingStores complete binding model



5.5 Configuring products

In SPLE, the configuration is a step-wise process that aims to deliver new software products that both satisfy the domain constraints, provided by the product line model, and the stakeholders' requirements. A product configuration can be a complex task because variability models can contain thousands of options (Siegmund *et al.*, 2008), and feature selection must consider several factors, such as technical limitations, implementation costs, requirements and the stakeholders' expectations. Therefore, the product configuration activity is usually tool supported. These software tools accelerate the configuration activity through the propagation of decisions and constraints, which reduce the number of errors.

In some cases, these tools auto-complete the configurations or guide the users through the configuration process.

Once the product requirements are identified, there are different ways to carry out the product configuration (Mazo *et al.*, 2018).

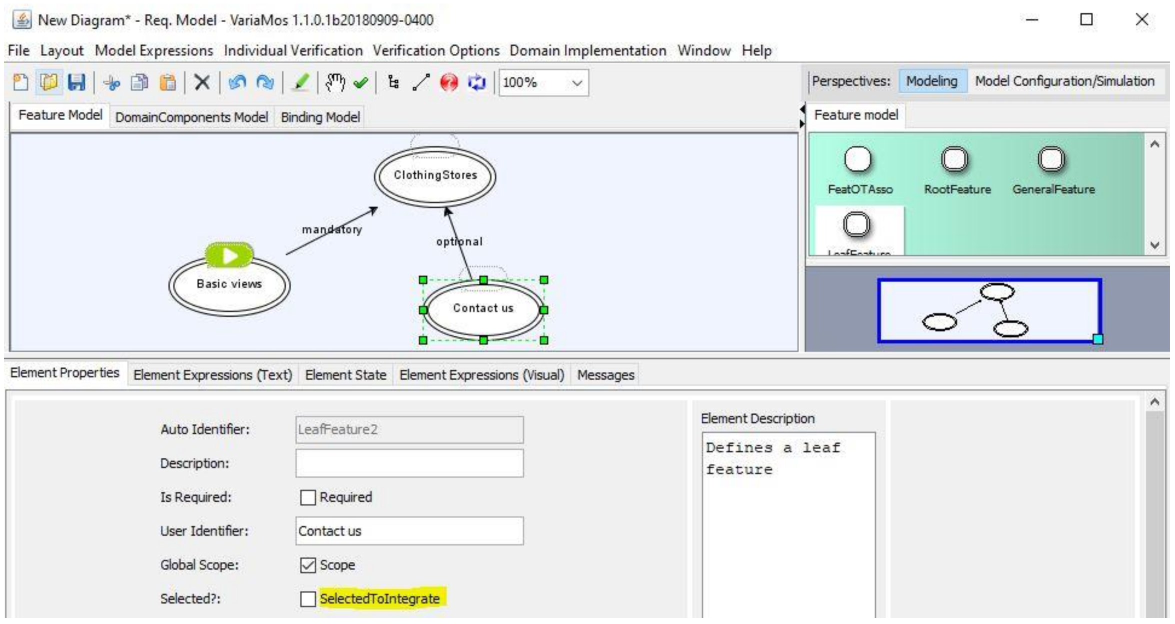
- **Selection approach** is an iterative process which consists of selecting the desired features that will be included in the final product. This approach starts with the selection of one desired feature `F1`, and the system looks for all possible configurations that include `F1`. The process continues by selecting a second desired feature `F2` and the system looks for all possible configurations that include `F1` and `F2`. The process is repeated until a final valid solution is found.
- **Reject approach** is an iterative process which consists of rejecting the features that will be excluded in the final product. This approach starts with the selection of one unwanted feature `F1`, and the system looks for all possible configurations that exclude `F1`. The process continues by selecting a second unwanted feature `F2` and the system looks for all possible configurations that exclude `F1` and `F2`. The process is repeated until a final valid solution is found.
- **Value reduction approach** consists of reducing step by step the values of some variables of the variability model, for example, reducing the range of a group cardinality.
- **Optimization approach** consists of finding a product configuration based on specific requirements that optimize the product with respect to specific criteria, for example, finding the product with the highest security level.

In FragOP, this activity consists of selecting the specific domain requirements that a certain product will contain. This configuration activity must satisfy the domain restrictions (which are represented in the PL models) and the customer needs.

5.5.1 VariaMos support

Product configuration is carried out in the feature model. There, the SPL developer selects the leaf features that the new SPL software product will contain, based on the customer needs.

Figure 5-13: VariaMos product configuration elements


New Diagram* - Req. Model - VariaMos 1.1.0.1b20180909-0400
File Layout Model Expressions Individual Verification Verification Options Domain Implementation Window Help
100%
Perspectives: Modeling Model Configuration/Simulation
Feature Model DomainComponents Model Binding Model
Feature model
ClothingStores
mandatory
optional
Basic views
Contact us
FeatOTAsso
RootFeature
GeneralFeature
LeafFeature
Element Properties Element Expressions (Text) Element State Element Expressions (Visual) Messages

Auto Identifier:	LeafFeature2
Description:	
Is Required:	☐ Required
User Identifier:	Contact us
Global Scope:	☑ Scope
Selected?:	☐ SelectedToIntegrate

Element Description

Defines a leaf feature

By default, VariaMos presents all the leaf features with a green arrow above them. This means that the leaf feature is already selected to be part of a new software product. Then, the SPL developer should deselect the `SelectedToIntegrate` option of those features that the SPL developer does not want to include in the new SPL software product (see Figure 5-13).

Note: VariaMos also provides a “Model configuration/Simulation” perspective which allows product configurations to be simulated. This way, SPL developers can verify if a specific product configuration is valid or not and compare it with the previous product configuration. The “Model configuration/Simulation” contains some capabilities such as create a first random configuration, go to the next random configuration, specify features that must be configured, and specify features that must not be configured, among others. A document explaining the “Model configuration/Simulation” perspective and its functionalities can be found online (Correa, 2018).

Other current approaches also support the product configuration, for example:

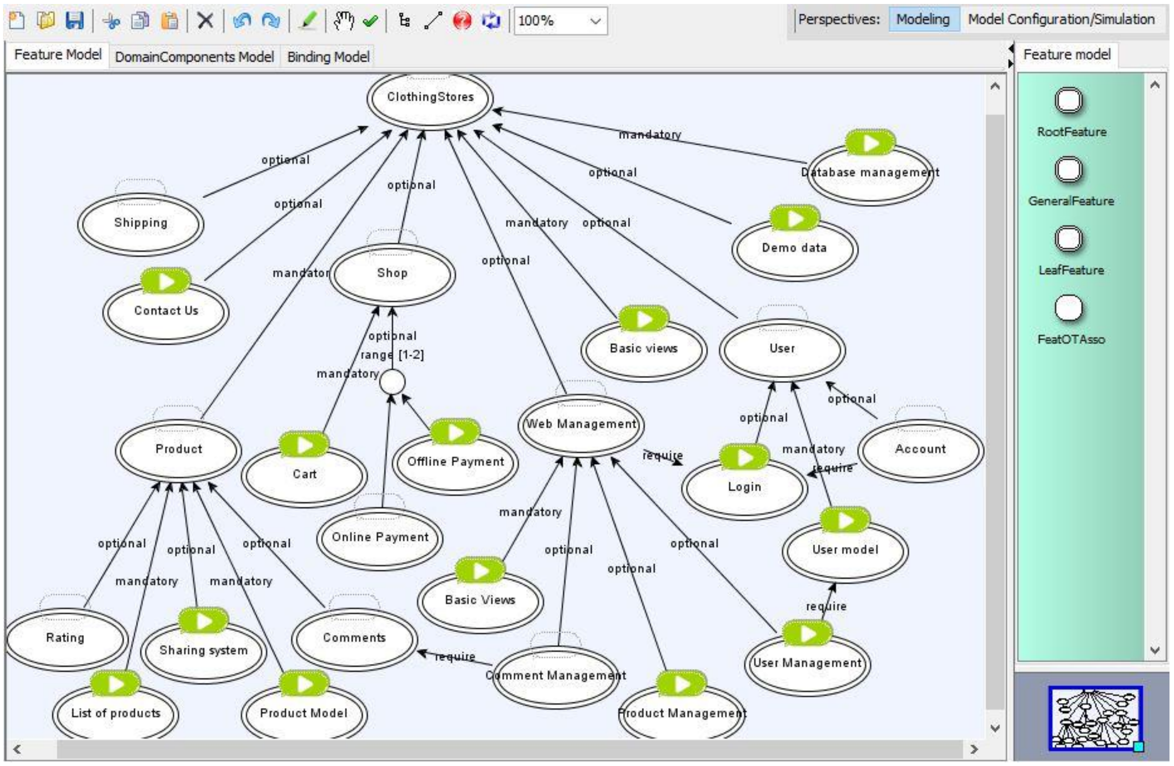
- **FeatureIDE with AHEAD (FOP) or FeatureHouse (FOP) or AspectJ (AOP).** A FeatureIDE project allows creating a configuration file “.config”. The SPL developer should open the configuration file and select the features that he/she wants to include in the current configuration.

- **DeltaJ (DOP).** This approach provides a .spl file, in this file, there is a variable called “Products” where the SPL developer types the name of a product and types the names of the deltas that he/she wants to include in each product.
- **CIDE (annotative).** This approach provides an option called "Generate Variant" from the project's context menu. There, the SPL developer has to select the features that he/she wants to include in the current configuration.

ClothingStores SPL

In order to present a realistic scenario, suppose that there is a “Customer A” who requires a new e-commerce clothing store application. After an elicitation process, the SPL developer deduced that “Customer A” requires the following list of functionalities: manage products, manage users, a sharing system for the products, a contact us section, a login system, demo data, and an offline payment system. Using this information, the SPL developer should try to configure a new SPL product.

Figure 5-14: ClothingStores final product configuration (VariaMos)

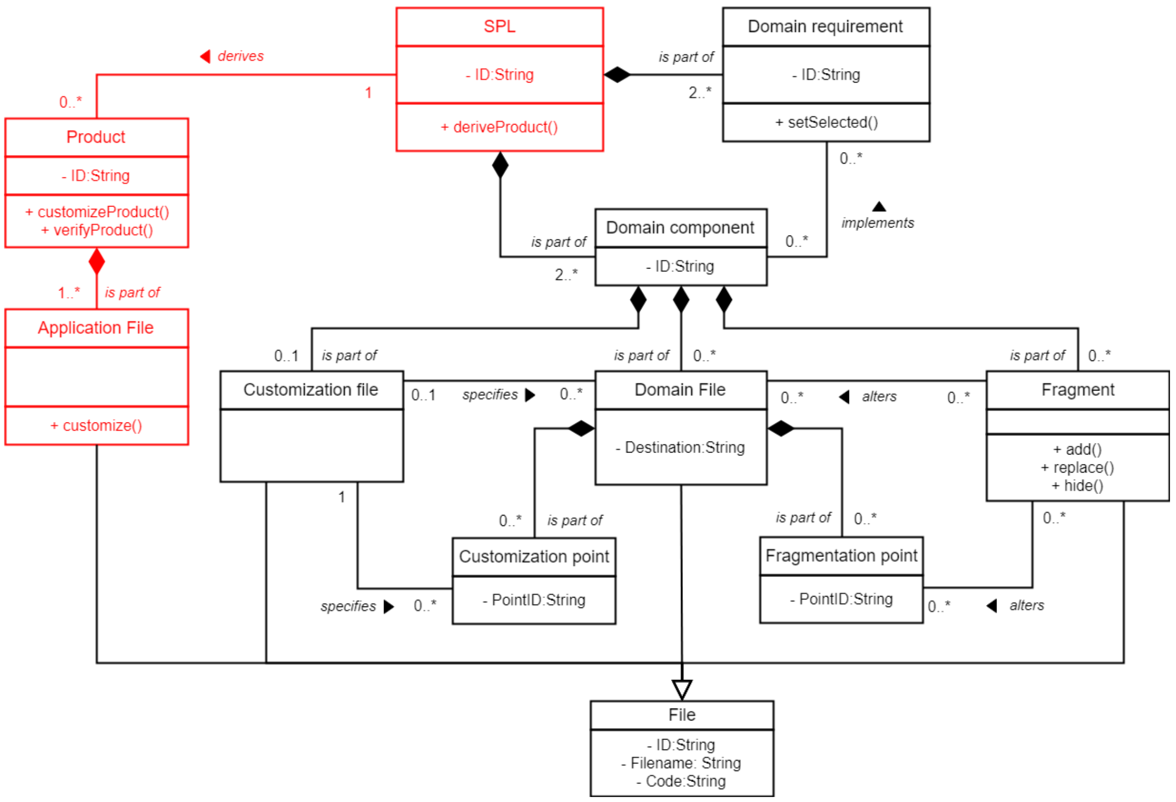


Based on the customer requirements, an SPL developer must deselect the `SelectedToIntegrate` option for each of the following features: (i) `shipping`, (ii) `rating`, (iii) `comments`, (iv) `comment management`, (v) `online payment`, and (vi) `account`. Figure 5-14 shows the final product configuration.

5.6 Deriving products

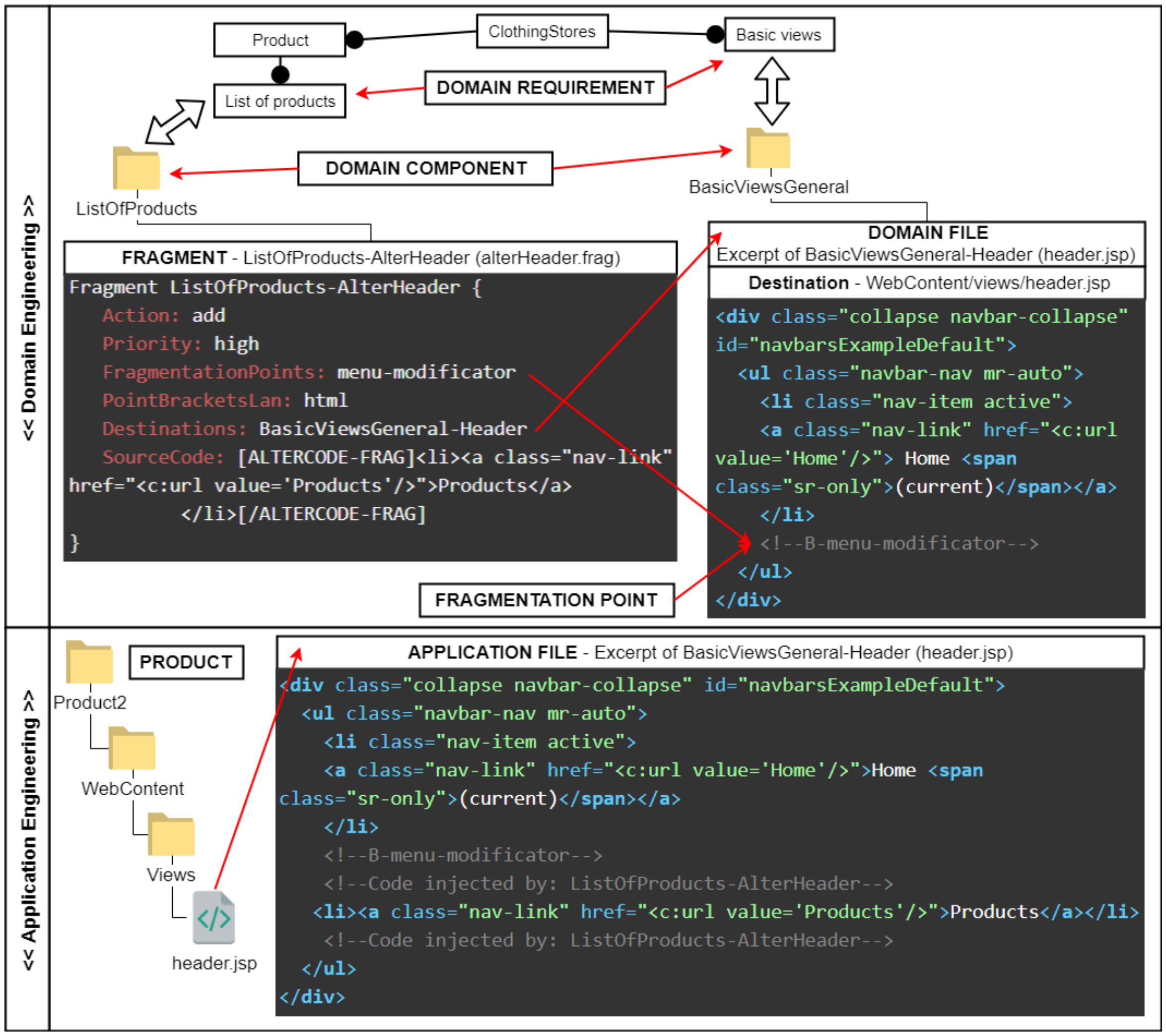
One of the key activities for an SPLE is the product derivation. Product derivation aims to create specific software products based on the assembling of the reusable domain components. Depending on the approach selected to build the domain components and the tool that supports the product derivation activity, the product derivation could be a quick automated activity or a slow manual activity. Therefore, effective product derivation activity is critical to ensure that the effort required to develop the common assets will be lower than the benefits achieved through their use (de Souza *et al.*, 2015).

Figure 5-15: FragOP metamodel highlighting the SPL, product and application file relationship



In the FragOP approach, the product derivation activity consists of generating a specific product which contains application files (see Figure 5-15). Figure 5-16 shows a realistic product derivation scenario. In this case, a domain file (header.jsp) and a fragment (alterHeader.frag) will be assembled. These two component files belong to two components (`BasicViewsGeneral` and `ListOfProducts`), that are bounded to two mandatory leaf features (`List of products` and `Basic views`). This means, that any ClothingStores product configuration will include these two leaf features. At application level, this means that the files of `BasicViewsGeneral` and `ListOfProducts` will be assembled. The product derivation activity requires tool support, so, the complete product derivation process will be explained in the following section.

Figure 5-16: An example of two component files being assembled



<< Domain Engineering >>
Product
ClothingStores
Basic views
List of products
DOMAIN REQUIREMENT
DOMAIN COMPONENT
ListOfProducts
BasicViewsGeneral
FRAGMENT - ListOfProducts-AlterHeader (alterHeader.frag)
DOMAIN FILE
Excerpt of BasicViewsGeneral-Header (header.jsp)
Destination - WebContent/views/header.jsp
FRAGMENTATION POINT
<< Application Engineering >>
PRODUCT
Product2
WebContent
Views
header.jsp
APPLICATION FILE - Excerpt of BasicViewsGeneral-Header (header.jsp)

```
Fragment ListOfProducts-AlterHeader {
   Action: add
   Priority: high
   FragmentationPoints: menu-modificator
   PointBracketsLan: html
   Destinations: BasicViewsGeneral-Header
   SourceCode: [ALTERCODE-FRAG]<li><a class="nav-link"
href="<c:url value='Products'/>">Products</a>
          </li>[/ALTERCODE-FRAG]
}
```

```
<div class="collapse navbar-collapse"
id="navbarsExampleDefault">
  <ul class="navbar-nav mr-auto">
    <li class="nav-item active">
    <a class="nav-link" href="<c:url
value='Home'/>"> Home <span
class="sr-only">(current)</span></a>
    </li>
    <!--B-menu-modificator-->
  </ul>
</div>
```

```
<div class="collapse navbar-collapse" id="navbarsExampleDefault">
  <ul class="navbar-nav mr-auto">
    <li class="nav-item active">
    <a class="nav-link" href="<c:url value='Home'/>">Home <span
class="sr-only">(current)</span></a>
    </li>
    <!--B-menu-modificator-->
    <!--Code injected by: ListOfProducts-AlterHeader-->
   <li><a class="nav-link" href="<c:url value='Products'/>">Products</a></li>
    <!--Code injected by: ListOfProducts-AlterHeader-->
  </ul>
</div>
```

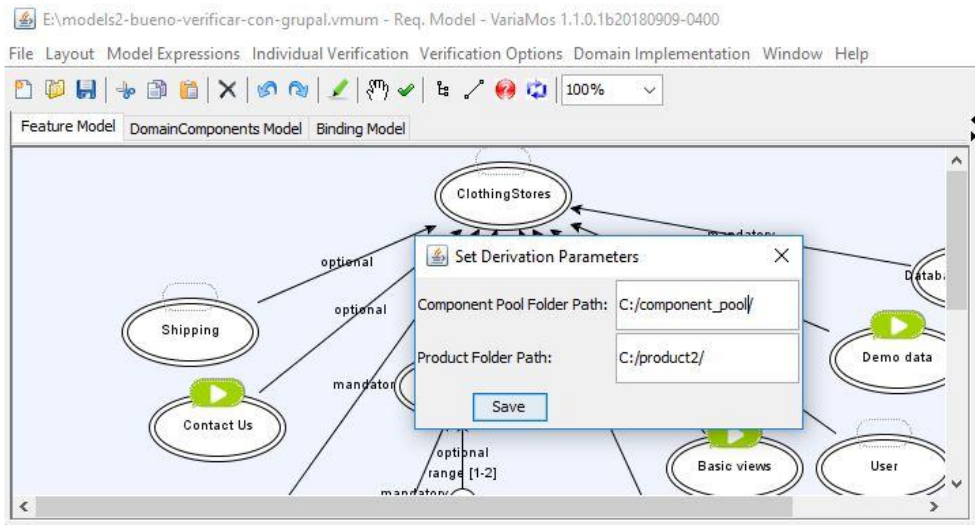
5.6.1 VariaMos support

VariaMos offers two functionalities to support the FragOP product derivation activity. In the VariaMos “domain implementation” menu, the SPL developer can find **Set derivation parameters** and **Product derivation**.

The **Set derivation parameters** option allows the definition of (i) the “component pool folder path” which is the path where components and files are stored, and (ii) the “product folder path” which is the path where the configured product will be derived. Figure 5-17 shows a “set derivation parameters” configuration.

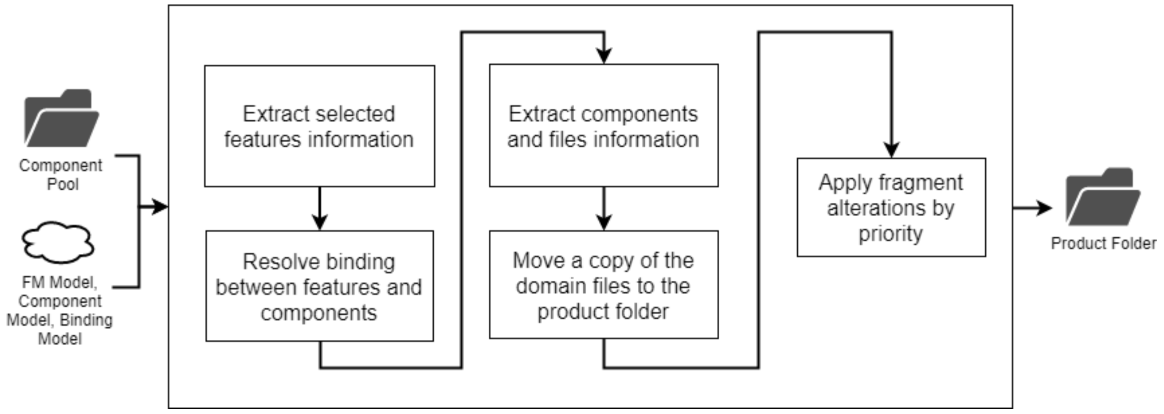
The **Product derivation** option allows the specific software product to be derived based on an automated algorithm that follows a series of instructions as presented in Figure 5-18.

Figure 5-17: VariaMos “set derivation parameters” configuration


E:\models2-bueno-verificar-con-grupal.vmum - Req. Model - VariaMos 1.1.0.1b20180909-0400
File Layout Model Expressions Individual Verification Verification Options Domain Implementation Window Help
100%
Feature Model DomainComponents Model Binding Model
ClothingStores
optional
optional
mandatory
Shipping
Contact Us
Datab
Demo data
Basic views
User
optional range [1-2]
Set Derivation Parameters
Save

Component Pool Folder Path: C:/component_pool/
Product Folder Path: C:/product2/

Figure 5-18: VariaMos product derivation activity


Component Pool
FM Model, Component Model, Binding Model
Extract selected features information
Resolve binding between features and components
Extract components and files information
Move a copy of the domain files to the product folder
Apply fragment alterations by priority
Product Folder

At the beginning of the VariaMos product derivation algorithm, the information is taken from the component pool folder and the developed models. Then, the algorithm (i) extracts the information of the selected leaf features (based on the product configuration activity, see Section 5.5); (ii) resolves the binding relationships of the selected features to establish the corresponding components and files (based on the binding activity, see Section 5.4); (iii) creates a copy of the component domain files (from the domain component pool) and moves the copied files to the product folder (these files represent the **application files**). The application files are moved to a specific subfolder based on the domain file destination. Finally, the algorithm (iv) applies the domain component fragment alterations to the application files by priority order. The output is a product folder, which contains the assembled domain components and the specific software product. This algorithm also provides different alerts, such as invalid fragment definition, missing fields, invalid fragmentation point definition, invalid actions, and invalid filenames and paths.

The FragOP product derivation activity and the VariaMos algorithm allows us to answer RQ2. Therefore, the VariaMos support to the product derivation activity also allows us to answer RQ4.

Other current approaches also support the product derivation, for example:

- **FeatureIDE with AHEAD (FOP) or FeatureHouse (FOP) or AspectJ (AOP).** After the SPL developer completes the configuration file which contains the selected features. The SPL developer must save the configuration file, and FeatureIDE will compose the features (or aspects) and compile the generated Java code.
- **DeltaJ (DOP).** After the SPL developer completes the .spl file information. The SPL developer saves the file and a popup displays the available products to be derived. The SPL developer selects the desired product, and the application composes the deltas. After all deltas are composed, the generated classes are serialized and written into the JAVA source files in the file system (in a folder called src-gen/). Therefore, Eclipse will automatically compile the generated files.
- **CIDE (annotative).** Once the product is configured (*i.e.*, the features are selected), CIDE will copy the source code to the target project and remove all colored code that is not included in the product configuration.

ClothingStores SPL

The first step is to "set the derivation parameters" according to the "component pool folder path" and the "product folder path". Then, the SPL developer should click on the "product derivation" option. In this case, the reusable domain components are assembled and stored in the "product folder", and as stated previously, the fragment alterations are applied.

Following the running example and based on the current product configuration (see Figure 5-14), the `ListOfProducts-AlterHeader` (alterHeader.frag) and the `Login-AlterAdmin` (alterAdmin.frag) fragments (see Listing 4-6) are executed in the `BasicViewsGeneral-Header` (header.jsp) and `UserManagement-ManageUsers` (ManageUsers.java) application files respectively (see Listing 4-4). The component assembly results are shown in Listing 5-1.

Listing 5-1: Derived BasicViewsGeneral-Header (header.jsp) and UserManagement-ManageUsers (ManageUsers.java) application files

BasicViewsGeneral-Header (header.jsp)

```
<%@ page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix = "fn" %>
<html>
  <head>
    <title>${title}</title>
    <link rel="stylesheet" type="text/css" href="<c:url value =
"/assets/css/bootstrap.min.css"/>" />
    <link rel="stylesheet" type="text/css" href="<c:url value =
"/assets/css/style.css"/>" />
  </head>

  <body>
    <nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark">
      <button class="navbar-toggler" type="button" data-toggle="collapse"
data-target="#navbarsExampleDefault" aria-controls="navbarsExampleDefault"
aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>

      <div class="collapse navbar-collapse" id="navbarsExampleDefault">
        <ul class="navbar-nav mr-auto">
          <li class="nav-item active">
            <a class="nav-link" href="<c:url value='Home'/>">Home <span
class="sr-only">(current)</span></a>
          </li>
          <!--B-menu-modificator-->
            <!--Code injected by: ListOfProducts-AlterHeader-->
            <li>
               <a class="nav-link" href="<c:url
value='Products'/>">Products</a>
            </li>
            <!--Code injected by: ListOfProducts-AlterHeader-->
            <!--Code injected by: Login-AlterHeader-->
            <c:choose>
               <c:when test="${sessionScope.logged != '1'}">
                  <li>
                     <a class="nav-link" href="<c:url
value='Login'/>">Login</a>
                  </li>
               </c:when>
               <c:otherwise>
                  <!--B-menu-modificator-login-->
                  <!--Code injected by: Account-AlterHeader-->
                  <li>
                     <a class="nav-link" href="<c:url
value='Account'/>">Account</a>
                  </li>
                  <!--Code injected by: Account-AlterHeader-->
                  <li>
                     <a class="nav-link" href="<c:url
value='Login?logout=1'/>">Logout</a>
                  </li>
               </c:otherwise>
            </c:choose>
            <!--Code injected by: Login-AlterHeader-->
            <!--Code injected by: ContactUs-AlterHeader-->
            <li>
               <a class="nav-link" href="<c:url
value='Contact'/>">Contact Us</a>
            </li>
            <!--Code injected by: ContactUs-AlterHeader-->
            <!--Code injected by: Cart-AlterHeader-->
            <li>
               <a class="nav-link" href="<c:url value='Cart'/>">Cart</a>
            </li>
            <!--Code injected by: Cart-AlterHeader-->
        </ul>
      </div>
    </nav>
   <div>
```

UserManagement-ManageUsers (ManageUsers.java)

```
package controllers.admin;
import java.io.IOException; import javax.servlet.RequestDispatcher; import
javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet; import
javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest; import
javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession; import models.User; import
models.UserDAO;

@WebServlet(urlPatterns = {"/Admin/Users"})
public class ManageUsers extends HttpServlet {

       /*B-validation-function*/
       /*Code replaced by: Login-AlterAdmin*/
       protected boolean validation(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException{
         HttpSession session = request.getSession();
         User u = (User) session.getAttribute("datauser");
         if(u == null) { response.sendRedirect("../Home"); return false; }
         else if(!u.getType().equals("admin")){
              response.sendRedirect("../Home"); return false;
         }
         return true;
       }
       /*Code replaced by: Login-AlterAdmin*/
       /*E-validation-function*/


            @Override
            protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {

             if(this.validation(request, response)){
                  String remove = request.getParameter("remove");
                  if(remove != null){
                   UserDAO.remove(Integer.parseInt(remove));
                  }

                  request.setAttribute("users",UserDAO.getUsers());
                  request.setAttribute("title", "Admin Panel - Users");
                         RequestDispatcher view =
request.getRequestDispatcher("../views/admin/users.jsp");
                         view.forward(request, response);
             }
             }

             protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {

             if(this.validation(request, response)){
                   String user = request.getParameter("user");
                   String pass = request.getParameter("pass");
                   String name = request.getParameter("name");
                   String type = request.getParameter("type");

                   User u = new User(name,type,user,pass); UserDAO.insert(u);
                   response.sendRedirect("Users");
             }
       }
}
```

Now we can see the `BasicViewsGeneral-Header` which contains a new menu element that links to the products section (highlighted code), which is also presented in Figure 5-16. It also contains other menu element modifications carried out by the `Login-AlterHeader`, `Account-AlterHeader`, `ContactUs-AlterHeader`, and `Cart-AlterHeader` fragments. The `UserManagement-ManageUsers` was also modified by the `Login-AlterAdmin` fragment. This fragment injected a new validation function which replaced the default version (highlighted code).

5.7 Customizing products

Product customization is a typical activity within SPLE. A product derivation hardly ever ends with a finalized software product. There are different kinds of product customizations, such as component parameterization, component adaptation, and component augmentation, among others (Cobaleda *et al.*, 2018).

- **Parameterization** consists of providing values to the product parameters, with the objective of adjusting them to the specific customer needs and the environment, for example, the configuration files may need to be parameterized.
- **Adaptation** customization should be applied when the domain component does not satisfy the customer needs fully. Commonly, it consists of modifying the component code.
- **Augmentation** is about the development of new components to supply product functionalities that are not included in the domain components. Commonly, augmentation is carried out when there are very specific requirements that were not considered during the domain engineering.

In FragOP, the use of customization points and customization files serve to make clear the specific places in which the products should be customized. These elements allow some specific product customizations to be applied, such as component parameterization and some component adaptations. Nevertheless, most of the complex product customizations such as a component augmentation, which involves the development of a new component, must be carried out manually by the SPL developer.

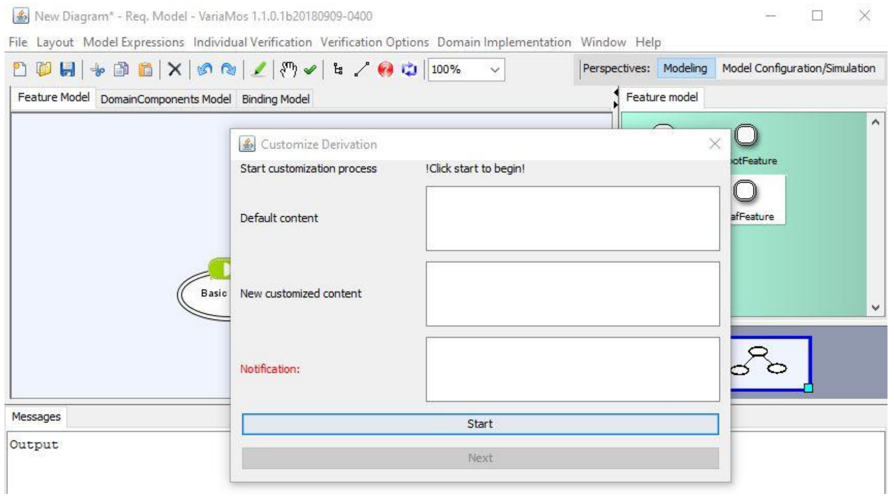
The FragOP product customization activity is supported by VariaMos and explained in the next section.

5.7.1 VariaMos support

In the VariaMos “domain implementation” menu, the SPL developer can find “product customization” option. After clicking the “product customization” option, VariaMos displays a popup menu which contains two buttons: start and next (see Figure 5-19). The **start button** initiates an algorithm that looks for customization files (based on the derived components). If a customization file is found, then the algorithm analyzes (i) the customization points, looks for the application files that contain the previous customization points and collects the code surrounded by each customization point. That code is displayed in the `Default content` text area. Next, the developer provides a `new customized content`. Here the developer customizes each application file, as is presented in Figure 5-19. And (ii) the domain files to be customized that do not include customization points. In this case, two buttons, `upload` and `save`, are presented. The developer uploads the new customized file using the `upload` button, and the `save` button stores the new customized file in the product folder. The **next button** sends the provided information and modifies the derived application files, therefore, if there are other pending customization points or customization files, the default content is refreshed with the new code or file to be customized. Finally, once there are no pending customization points or customization files, the next button is disabled and the customization activity finishes.

As mentioned in the previous chapter, most of the SPL implementation approaches do not provide a product customization capability (these include CIDE, DeltaJ, Munge, Antenna, AspectJ, and AHEAD, among others). Even when VariaMos (FragOP) does not automatically customize the application files (because the SPL developers should modify the code that appears in the popups), the reality is that this activity is streamlined. Otherwise, without the use of customization points and customization files, the SPL developers would need to manually look over each derived application file, trying to figure out what pieces of code and files should be customized.

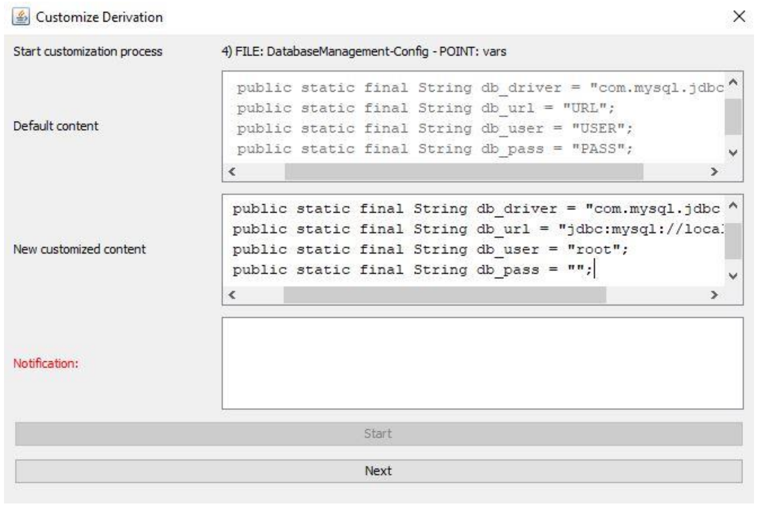
Figure 5-19: VariaMos product customization activity


New Diagram* - Req. Model - VariaMos 1.1.0.1b20180909-0400
File Layout Model Expressions Individual Verification Verification Options Domain Implementation Window Help
100%
Perspectives: Modeling Model Configuration/Simulation
Feature Model DomainComponents Model Binding Model
Feature model
Customize Derivation
Start customization process
!Click start to begin!
Default content
New customized content
Notification:
Start
Next
Basic
otFeature
afFeature
Messages
Output

ClothingStores SPL

In the ClothingStores SPL, the `DatabaseManagement-Config` contained a customization point which was surrounded by some variables that had to be parameterized (presented in Listing 4-8). This parametrization could be easily carried out using VariaMos. The SPL developer executes the "product customization" option, and a popup with the customization points is presented. Figure 5-20 shows the `vars` customization point which belongs to the `DatabaseManagement-Config` (Config.java) derived application file. Finally, the developer should manually provide the final database variable values for the `new customized content`.

Figure 5-20: ClothingStores product customization execution (VariaMos)


Customize Derivation

Start customization process — 4) FILE: DatabaseManagement-Config - POINT: vars

Default content

```
public static final String db_driver = "com.mysql.jdbc
public static final String db_url = "URL";
public static final String db_user = "USER";
public static final String db_pass = "PASS";
```

New customized content

```
public static final String db_driver = "com.mysql.jdbc
public static final String db_url = "jdbc:mysql://loca
public static final String db_user = "root";
public static final String db_pass = "";
```

Notification:

5.8 Verifying products

After the product derivation and customization, it is critical to verify and validate the software product by applying tests, such as integration tests, system tests and acceptance tests (Engström & Runeson, 2011).

- **Integration tests** allow the assembly of the components to be tested. In this kind of test, it is suggested that the critical subsystems of the entire product be identified, and the proper tests executed. There are different strategies to execute integration tests, such as *big-bang* in which all the components of the subsystem are assembled, and the test is executed. *Top-down* is a strategy in which a specific component is tested, and some additional components are later included for testing. The process is repeated until all components are tested.

- **System tests** take the complete assembled product as an entity for testing. This test tries to find the resulting errors of the subsystem interactions. It also allows the compliance of the functional and non-functional requirements to be evaluated. There are different kinds of system tests, such as security tests, user interface tests, recovery tests, and compatibility tests, among others.
- **Acceptance tests** are applied after the system tests. These tests include the execution of the system functionalities of the finalized product. The tests have to be executed by the user or group of users who will use the system. If the product passes the tests the stakeholder will accept it, which confirms that it fulfills the stakeholder needs.

In FragOP it is critical to verify the product application files before the integration, system and acceptance tests. The FragOP approach allows multiple pieces of code which are developed in several languages to be injected. This is due to the FragOP structure and the FragOP concepts, such as fragmentation points, fragments, customization points, and customization files, among others. It implies that any time a fragment is executed, a derived application file will probably be altered through code injection. Therefore, as part of the product customization activity, the SPL developer could inject wrong code or remove essential code. As a consequence, ensuring a proper application file code structure and grammar is essential.

This code verification can be carried out using an IDE. For instance, an SPL developer can move the product application files to a new Eclipse project, and the Eclipse platform will highlight the code errors. Alternatively, the VariaMos tool can also be used.

5.8.1 VariaMos support

VariaMos provides automated **product syntax verification**. To execute it, the SPL developer goes to the “domain implementation” menu and clicks on the “product verification” option. If the product was properly derived and the files contain the correct syntax, VariaMos shows the “no errors found” message. Nevertheless, if there are files with incorrect structure or grammar, VariaMos shows an alert with the specific code line in which each file contains errors.

How does the product syntax verification activity work?

The product syntax verification activity allows grammar errors to be found over the derived application files. To this end, VariaMos uses ANother Tool for Language Recognition (ANTLR; Parr, 2013) which is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions. VariaMos implements ANTLR 4.7.1 and uses a series of parsers and lexers for languages, such as PHP, Java, CSS, and MySQL, among others. Once the product derivation is executed and once the SPL developer clicks the "product verification" option, VariaMos extracts the product folder information. Based on each derived application file extension, VariaMos analyses the grammar of each file and generates alerts if errors are found. This product syntax verification support allows us to answer RQ3.

ClothingStores SPL

To explore the workings of the product syntax verification, we will apply a very small modification to the `Login-AlterAdmin` (alterAdmin.frag) fragment source code (see Listing 5-2). In this case, instead of giving `protected` visibility to the `validation function`, we wrote the `protecte`, which is a typographical error.

Listing 5-2: Introducing an error to the Login-AlterAdmin (alterAdmin.frag) fragment source code

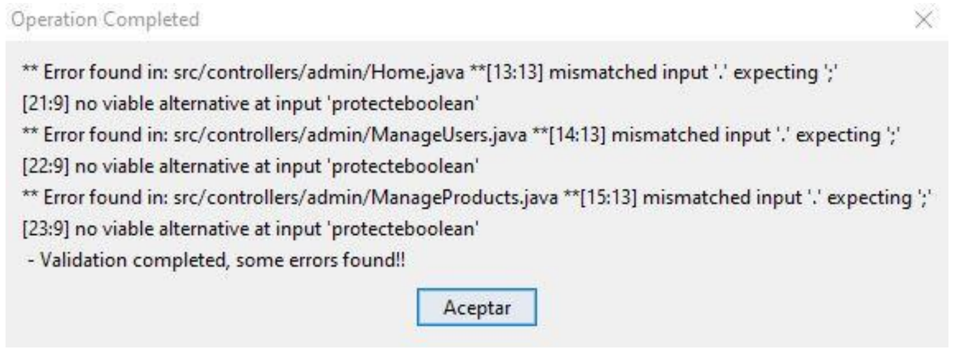
Login-AlterAdmin (alterAdmin.frag)

```
Fragment Login-AlterAdmin {
   Action: replace
   Priority: high
   FragmentationPoints:  validation-function,  validation-function,  validation-
function, validation-function
   PointBracketsLan: java
   Destinations:     BasicViewsAdmin-Home,     CommentManagement-ManageComment,
ProductManagement-ManageProducts, UserManagement-ManageUsers
   SourceCode:  [ALTERCODE-FRAG]protecte  boolean  validation(HttpServletRequest
request, HttpServletResponse response) throws ServletException, IOException{
             HttpSession session = request.getSession();
       User u = (User) session.getAttribute("datauser");
       if(u == null) { response.sendRedirect("../Home"); return false;   }
       else if(!u.getType().equals("admin")){
             response.sendRedirect("../Home"); return false;
       }
       return true;
       }[/ALTERCODE-FRAG]
}
```

The `Login-AlterAdmin` (alterAdmin.frag) will inject the wrong code into four different files: `BasicViewsAdmin-Home`, `UserManagement-ManageUsers`, `CommentManagement-ManageComment` and `ProductManagement-ManageProducts` (depending on whether the corresponding components will be derived or not). Based on the current example and configuration (see Section 5.5), three files will contain the wrong code: `BasicViewsAdmin-Home`, `UserManagement-ManageUsers`, and `ProductManagement-ManageProducts`.

Now, in order to see how VariaMos displays the alerts, we must (i) modify the `Login-AlterAdmin` (alterAdmin.frag) as shown in Listing 5-2, (ii) click the "product derivation" option and (iii) click the "product verification" option. Figure 5-21 shows the VariaMos alerts for the current derivation and configuration, which states that "protecte boolean" is invalid for the following derived application files: src/controllers/admin/Home.java, src/controllers/admin/ManageUsers.java, and src/controllers/admin/ManageProducts.java.

Figure 5-21: Verify derivation alert (VariaMos)



```
** Error found in: src/controllers/admin/Home.java **[13:13] mismatched input '.' expecting ';'
[21:9] no viable alternative at input 'protecteboolean'
** Error found in: src/controllers/admin/ManageUsers.java **[14:13] mismatched input '.' expecting ';'
[22:9] no viable alternative at input 'protecteboolean'
** Error found in: src/controllers/admin/ManageProducts.java **[15:13] mismatched input '.' expecting ';'
[23:9] no viable alternative at input 'protecteboolean'
 - Validation completed, some errors found!!
```

It is important to note that VariaMos product syntax verification is not enough, there are more product verifications and product validations that should be carried out manually. We also recommend applying different system tests (Sawant *et al.*, 2012), such as security testing, graphical user interface testing, compatibility testing, and recovery testing, among others.

Finally, the software product is ready to be deployed in the production environment. Section 6.1 shows some finalized ClothingStores products that are running over on an Apache Tomcat server.

5.9 Summary

This chapter presented the FragOP process with its eight main activities. For each activity, we presented (i) the theory from the SPLE literature and how it should be applied using the FragOP approach, (ii) how VariaMos supported it, and (iii) a demonstration and exemplification in the running example which provided a realistic scenario. A summary of each activity is presented below.

1. Modeling PL requirements is about the elicitation and formally specification of the domain requirements. Commonly, this activity is carried out with the use of a variability model. FragOP and VariaMos allow for domain requirements to be specified in the form of a feature model. There are some plans to provide support to languages such as OVM, and goal models, among others.
2. Modeling domain components allows specifying the SPL domain components, which includes information about the components, their files and their relationships. FragOP and VariaMos allow for domain components to be graphical represented in the form of a component model. This representation allows having a complete separation between the domain requirements (problem space) and the domain components (solution space).
3. Implementing domain components refers to the realization of each component and file represented in the domain component model. This is one of the most complex activities in the FragOP process because it requires the codification of domain component files, fragmentation points, fragments, customization points, and customization files. For this activity there is not VariaMos support, so the SPL developers should use their preferred IDE to codify the components' code.
4. Binding domain requirements and domain components is about the connection between the domain requirements (problem space) and the domain components (solution space). This connection is commonly carried out through the use of models or configuration documents. FragOP and VariaMos allow creating a binding model which specifies how the domain components operationalize the domain requirements. At present, VariaMos only allows a one-to-one binding relationship (one component linked to one feature), however, we plan to implement constraint networks to graphically represent more complex relationships.
5. Configuring products is a step-wise process that aims to deliver new software products that both satisfy the domain constraints, provided by the product line

model, and the stakeholders' requirements. In FragOP and VariaMos, this activity consists of selecting the specific leaf feature that a certain product will contain, based on the customer needs.

6. Deriving products is a complex activity that aims to create specific software products based on the integration of the reusable domain components. FragOP and VariaMos allow an automated product derivation through the execution of an algorithm which takes as inputs the developed models, the component pool folder, and the selected leaf features; the output is a derivation folder which includes the assembled component files. Fragments' codes are injected in this activity.
7. Customizing products is about to apply the final modifications to the derived files based on the customer specific needs, such as component parameterizations, adaptations, and augmentations. Thanks to the customization points and customization files, FragOP and VariaMos provide a way to guide the SPL developers in the customization activity. However, complex product customizations such as a component augmentation must be carried out manually by the SPL developer.
8. Verifying products is about ensuring the quality of the derived and customized products. VariaMos allows verifying that the derived files contain a proper structure and proper grammar. In this instance, VariaMos uses ANTLR and based on each derived component file extension, it analyses the grammar of each file and generates alerts if errors are found. It is important to highlight that other verifications and tests such as integration tests, system tests, and acceptance tests must be applied manually.

To conclude, the definition of the FragOP product derivation activity and the VariaMos derivation algorithm (see Section 5.6) allows us to answer RQ2 because of that activity details and specifies the way in which the SPL components should be assembled. The definition of the FragOP product syntax verification activity and the VariaMos support (see Section 5.8) allows us to answer RQ3 because of that activity details and specifies the way in which the SPL components should be verified. And the development and enhancement of the VariaMos tool allow us to answer RQ4 because that tool supports and improves the SPL component implementation and assembly.

6. Evaluation

Having the FragOP approach defined, and a stable version of VariaMos, it is important to evaluate how FragOP and VariaMos support the SPL implementation. In order to do that, we defined 3 evaluation questions:

- EQ1. Is VariaMos (FragOP) expressive enough to implement a real world, variant-rich multi-language software system?
- EQ2. What are the differences between VariaMos (FragOP) and similar SPL implementation mechanism and tools?
- EQ3. Is VariaMos a useful tool that supports the FragOP approach?

In order to analyze and try to find an answer to the previous questions, we present in the next three subsections: (i) a discussion about the ClothingStores SPL results, (ii) a discussion about the comparison between VariaMos (FragOP) and other SPL implementation mechanisms, and (iii) a usability test of VariaMos to support the FragOP approach.

6.1 ClothingStores results

The ClothingStores SPL was designed with the intention to provide a real example of the use of VariaMos (FragOP) to design and implement an SPL. Therefore, it was designed as an initial case study to gain insights about if VariaMos (FragOP) is expressive enough to implement a real world, variant-rich multi-language software system.

We used the Koscielny *et al.* (2014) DeltaJ 1.5 case study as the base to present the VariaMos ClothingStores results. In their study, they: (i) evaluated DeltaJ 1.5 (which is a DOP approach), (ii) used a similar question to our EQ1 question, and (iii) used a SimpleTextEditor SPL as the subject system. In comparison, the SimpleTextEditor consisted of 11 features, while the ClothingStores consists of 25 features.

Koscielny *et al.* (2014) also included an extra research question about a comparison between DeltaJ 1.5 and a plugin-based approach (ECLIPSE RCP). In our case study, we decided to leave a discussion about the ClothingStores results compared with other approaches to the end of this section. This is due to most current SPL implementation tools do not support all the software languages involved in the ClothingStores implementation. Furthermore, in Section 6.2 we present a deeper analysis of the SPL implementation mechanisms compared with VariaMos (FragOP).

The subject study (ClothingStores SPL) was properly defined and detailed in Chapter 2. There, we described the SPL requirements and software architecture. The ClothingStores was designed as a real-world example, covering most of the problems SPL developers face when implementing an SPL (which is of particular importance for EQ1). These problems include crosscutting concerns, fine-grained extensions, coarse-grained extensions, product customization, and managing multiple language files. These problems are detailed in Chapter 2.

In the next subsections, we will show the evaluation metrics and the initial case study results.

6.1.1 Evaluation metrics

The analysis and evaluation of VariaMos source code are difficult, because of missing tool support to measure variability-aware metrics. To overcome this problem and to cover all the SPL features, we derived five products (as seen in Table 6-1), and we manually measured the generated application files source code. For each derived product, we recorded:

- **Name.** Product name.
- **Leaf features selected.** The leaf features that were selected in order to derive the product, which was based on the feature model (see Figure 5-3).
- **Linked domain files.** The number of domain files that were linked to the previous leaf features, which was based on the component model (see Figure 5-6) the binding model (see Figure 5-12).
- **Derived files.** The number of application files that were included in the product folder, after the product derivation activity.

- **Total derived LOC.** The number of lines of code of all derived files (image files were not included).
- **Total injected LOC.** The number of lines of code that were automatically injected by the fragments in the product derivation activity.
- **Customized LOC.** The number of lines of code that were customized to finalize the product.
- **Percent of injected LOC.** The percentage of lines of code that were injected, compared with the total derived LOC.
- **Time to derive (sec).** The seconds that took to derive each product. We applied a little change over the VariaMos tool, in order to record the time spent in the product derivation activity.

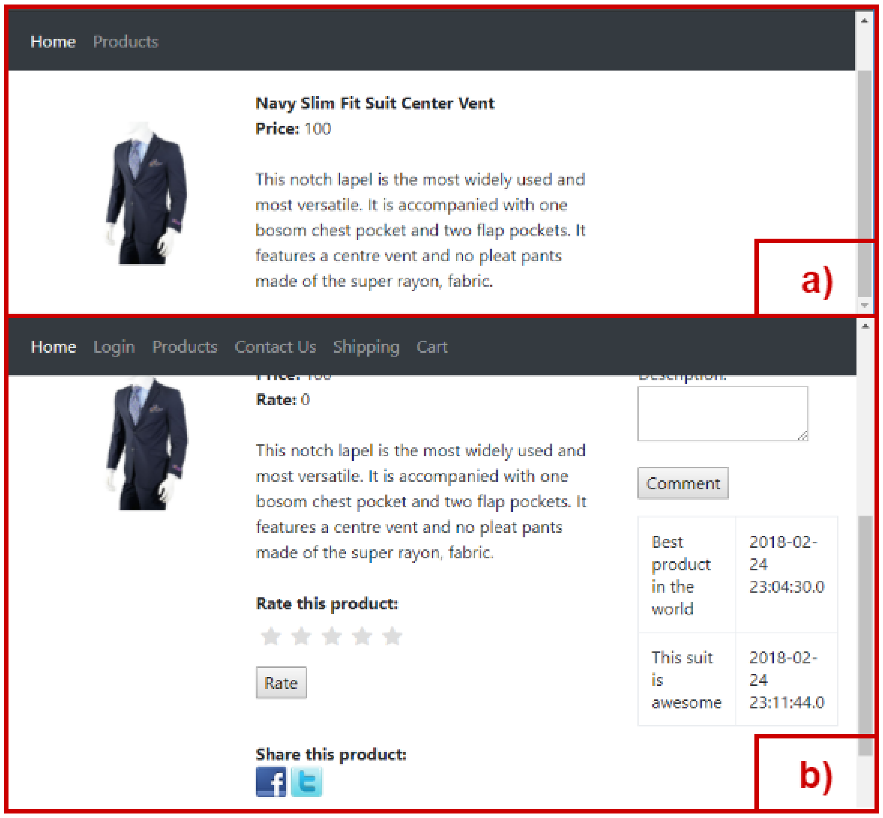
Table 6-1: Derived products

Product name	**SPL leaf features selected**
P1	Basic views, Database management, Demo data, List of products, Product Model
P2	Basic views, Comments, Contact Us, Database management, Demo data, List of products, Product Model, Shipping
P3	Basic views, Basic views (web management), Comment Management, Comments, Contact Us, Database management, Demo data, List of products, Login, Product Management, Product Model, Shipping, User model
P4	Basic views, Basic views (web management), Comment Management, Comments, Contact Us, Database management, Demo data, List of products, Login, Product Management, Product Model, Rating, Sharing system, Shipping, User model
P5	Account, Basic views, Basic views (web management), Cart, Comment Management, Comments, Contact Us, Database management, Demo data, List of products, Login, Product Management, Online Payment, Offline Payment, Product Model, Rating, Sharing system, Shipping, User Management, User model

6.1.2 Results

After following the FragOP process (see Chapter 5), we completed the derivation of the five products. The product finalization was very easy: (i) the derived products were located in Eclipse web projects, (ii) a couple of Java libraries were added to the projects (a MySQL connector, and a JSTL library), (iii) the SQL files were imported into a MySQL database, and (iv) the projects were run over an Apache Tomcat server.

Figure 6-1: Product section of a derived product (P1); Product section of a derived product (P5)



Navy Slim Fit Suit Center Vent
Price: 100

This notch lapel is the most widely used and most versatile. It is accompanied with one bosom chest pocket and two flap pockets. It features a centre vent and no pleat pants made of the super rayon, fabric.

Rate: 0

This notch lapel is the most widely used and most versatile. It is accompanied with one bosom chest pocket and two flap pockets. It features a centre vent and no pleat pants made of the super rayon, fabric.

Rate this product:

Share this product:

Best product in the world	2018-02-24 23:04:30.0
This suit is awesome	2018-02-24 23:11:44.0

Figure 6-1 shows the P1 and P5 products running over a web browser; Figure 6-1-a shows the P1 "product section" which contains a very basic configuration where the final user is able to read the product information; and Figure 6-1-b shows the P5 "product section" which contains a complete configuration where the final user is able to rate, share, comment and add the product to the cart.

The results show that VariaMos (FragOP) is *expressive enough* (EQ1) to implement a real-world, variant-rich multi-language software system. An inspection of the product code shows that:

- Multiple assets of different types (such as SQL, images, JavaScript, HTML, JSP, Java, and CSS) were automatically assembled and deployed in the respective project folder structure (see Figure 6-2).
- Several LOC were derived and automatically injected (see Figure 6-3 and Figure 6-4). For instance, 27.72 percent of the P5 LOC were automatically injected. This

means that a P5 product derivation carried manually without the use of VariaMos will require to manually modify 560 LOC.

- Between 21 and 50 lines of code were manually customized (supported by the VariaMos tool) to complete each product finalization (see Table 6-2 and Figure 6-5). Even, the database queries were automatically generated.
- If we try to derive the P5 with a compositional approach that is attached to a host language like Java (such as AspectJ, DeltaJ, AHEAD), (i) 26 files must be manually included in the product folder structure, and (ii) a minimum of 284 LOC (14% of the total derived LOC) must be manually modified to finalize the product derivation. Even, without counting the LOC of Java that implies fine-grained extensions that are not supported by these approaches.
- If we try to derive the P5 with annotative approaches, the results could vary depending on the annotative approach language support (for instance, Antenna only supports Java); however, as mentioned before, annotative approaches inject all code variations inside the domain files, which is not the case in VariaMos (FragOP). It means that a domain file such as `ListOfProducts-OneProduct` (oneproduct.jsp) will contain at least 104 LOC in an annotative approach. Nevertheless, in VariaMos (FragOP) it only contains 31 LOC and the code variations are located in separated files (fragments). This characteristic makes domain files of annotative approaches difficult to maintain and evolve.

Table 6-2: ClothingStores derivation results

Name	Leaf features selected	Linked domain files	Derived files	Total derived LOC	Total injected LOC	Customized LOC	% Injected LOC	Time to derive (sec)
P1	5	24	19	486	37	21	7.61	0.04386
P2	8	39	26	802	103	48	12.84	0.05396
P3	13	60	37	1410	240	50	17.02	0.08725
P4	15	70	39	1602	432	50	26.97	0.13434
P5	20	85	46	2020	560	50	27.72	0.18426

Figure 6-2: Number of files of each derived ClothingStores product by file type

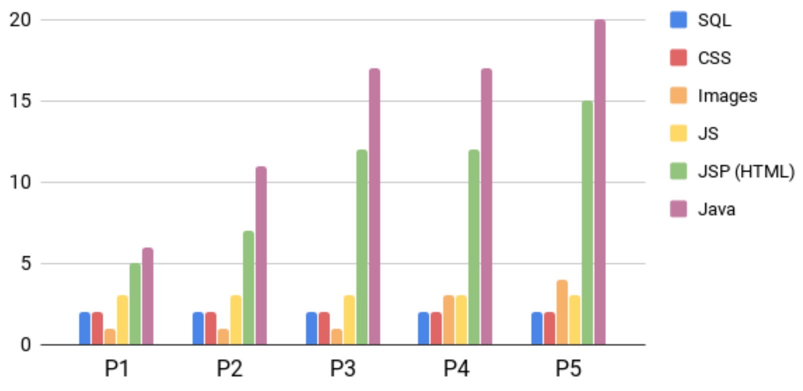


Figure 6-3: LOC of each derived ClothingStores product by file type

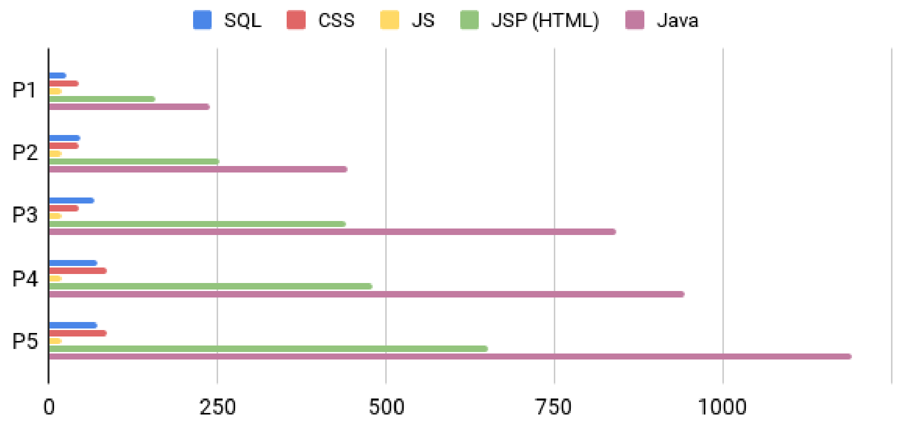


Figure 6-4: LOC automatically injected in each derived ClothingStores product by file type

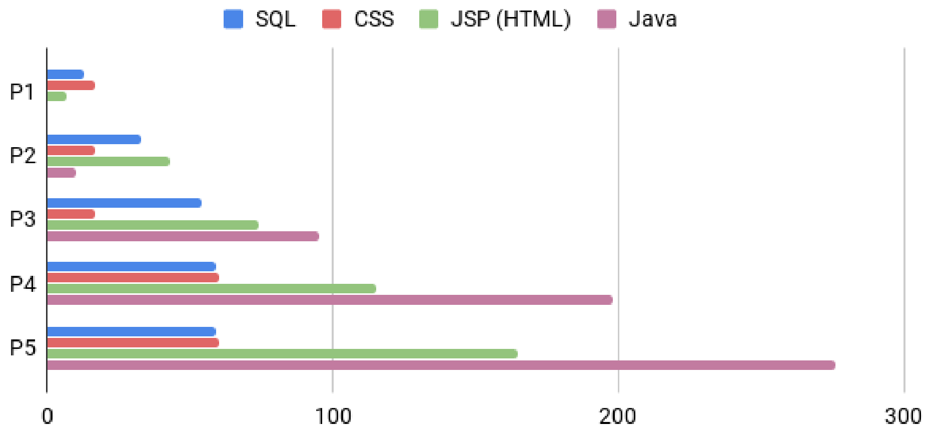
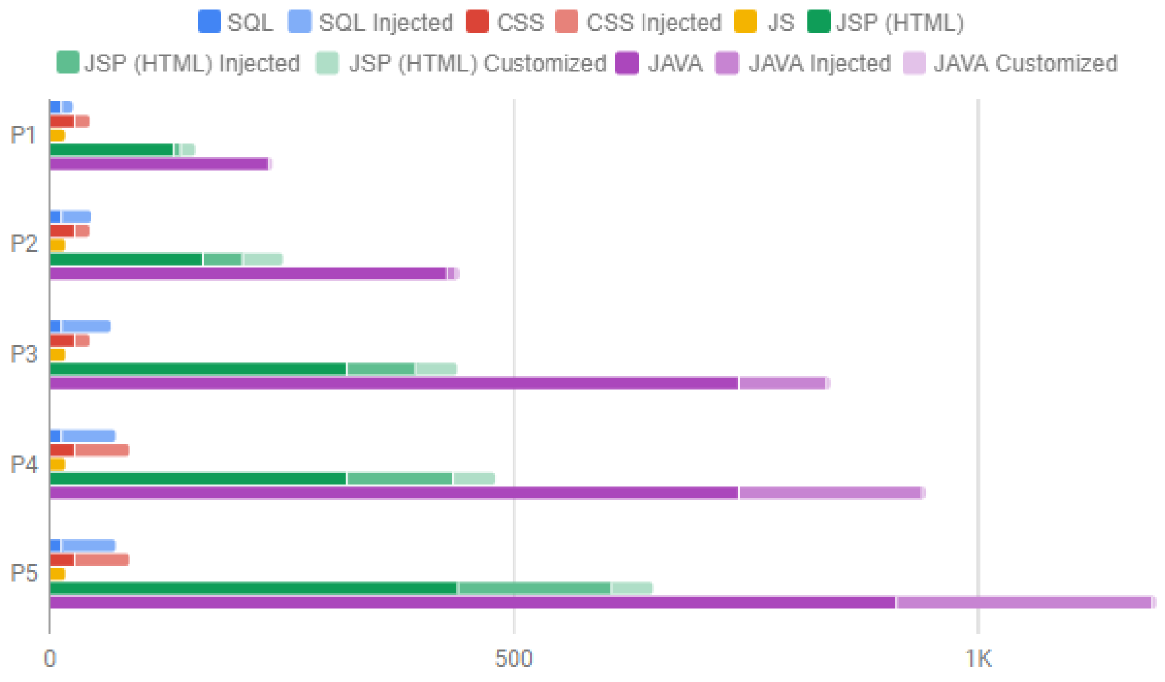


Figure 6-5: Summary of LOC reused, automatically injected and customized of each derived ClothingStores product by file type



6.1.3 Threads to validity

Although we conducted our evaluation with care, it exhibits some limitations. First, our case study consists of only one system that is small in size and has been implemented for this purpose only. Hence, our findings are not generalizable to other (large-scale) systems. Second, we have not implemented the system with other approaches for implementing variant-rich systems. However, we made some theoretical comparisons that show the benefits of the VariaMos (FragOP) approach. We also suggest implementing ClothingStores with multiple approaches in the future. Third, we analyzed concrete variants rather than the complete code base. Especially for quality-related measures, such as complexity or cohesion, this may be limited. However, the main focus of this paper was rather the technical realization of VariaMos (FragOP), whereas the evaluation was complementary in order to demonstrate the applicability of VariaMos (FragOP) for implementing SPLs. We will address the aforementioned shortcomings in a comprehensive case study in future work.

6.2 SPL implementation mechanisms comparison

The SMS developed in this thesis and many additional references have shown some different software tools and SPL implementation approaches used to implement an SPL. In this section, we compare five SPL implementation approaches which are automated with the use of non-commercial tools. The comparison is done based on (see Table 6-3):

- **Approach type**, compositional or annotative or a mix.
- **Approach language independence** refers to the way in which the approach is attached to a particular language and its structure, or if it is language independent.
- **Tool analyzed**, the SPL tool used to support the SPL implementation approach, and which was analyzed for this comparison.
- **Tool software language support**, the software languages that the selected tool currently supports.
- **Variation points** refer to the way to include the variation points inside the component files, or the elements that are used a based to include the variability.
- **Requires modifying the component code to include the variation points**, yes or no depending on the way the approach or tool support the variation points inclusion.
- **Variant units** refer to the way in which the code to be injected, modified, or removed is codified.
- **Allow injecting a variant unit in multiple variant points** refers to the possibility of injecting a code variant in multiple places.
- **Granularity support,** the type of extensions that can be applied with the current approach or tool (coarse-grained or fine-grained extensions).
- **Type of files used in the component implementation**.
- **Variability model support** refers to the variability models supported by the tool.
- **Product configuration** refers to the way to configure a specific product.
- **Support product customization**, yes or no depending on if the approach or tool support the product customization.

Table 6-3: SPL implementation approaches and tools comparison summary

	AOP	**CIDE**	**DOP**	**FOP**	**FragOP**
Approach type	Compositional	Annotative	Compositional	Compositional	Compositional and annotative

Approach language independence	Usually depending on a particular host language	Language independent	Usually depending on a particular host language	Usually depending on a particular host language	Language independent
Tool analyzed	AspectJ with FeatureIDE	CIDE	DeltaJ	AHEAD with FeatureIDE	VariaMos
Tool software language support	Java	Multiple	Java	Java	Multiple
Variation points	Object-oriented elements such as classes, methods, and attributes	Markers (annotations)	Object-oriented elements such as classes, methods, and attributes	Object-oriented elements such as classes, methods, and attributes	Fragmentation points (annotations)
Requires modifying the component code to include the variation points	No	Yes	No	No	Yes (minimum modification with language comments)
Variant units	Separated (aspect files)	Combined (with the original file)	Separated (delta module files)	Separated (Jak files)	Separated (fragment files)
Allow injecting a variant unit in multiple variant points	Yes	No	No	No	Yes
Granularity support	Coarse-grained extensions	Coarse-grained and fine-grained extensions	Coarse-grained extensions	Coarse-grained extensions	Coarse-grained and fine-grained extensions
Type of files used in the component implementation	Component files and aspects	Component files with annotations	Delta modules (delta files)	Feature modules (Jak files)	Component files (with fragmentation points and customization points), fragments, and customization files
Variability model support	Feature model (graphical)	Feature model (graphical and textual)	Feature model (textual)	Feature model (graphical)	Feature model (graphical)
Product configuration	Through feature selection (graphical)	Through feature selection (graphical)	Through delta selection (textual)	Through feature selection (graphical)	Through leaf feature selection (graphical)
Support product customization	No	No	No	No	Yes

As the previous table shows, FragOP (VariaMos) presents some important advantages versus other approaches (i) FragOP (VariaMos) as other annotative approaches (such as CIDE) is language independent, this characteristic allows implementing different components developed in different software language under the same approach. (ii)

FragOP (VariaMos) as other annotative approaches permits coarse-grained and fine-grained extensions, which is very useful for applying changes in very specific locations. (iii) FragOP (VariaMos) as other compositional approaches separates the variations unit from the domain component files, which reduce the complexity and improves the maintainability of the domain components files. (iv) FragOP (VariaMos) as other annotative approaches requires the modification of the domain component files to include the variations points, however, this modification is different from most annotative approaches; because the modification is minimum and does not interfere with the code, since it uses language comments blocks. (v) FragOP (VariaMos) as many other approaches supports feature model, however, it is designed in a way that new custom models can be included. (vi) FragOP (VariaMos) as AOP approaches allow injecting a single variant unit in multiple variant points, which improves the reusability and maintainability. And (vii) FragOP (VariaMos) compared to other approaches is the only one that supports product customization.

The comparison presented in this section was made based on some previous work (Schaefer *et al.*, 2010; Correa *et al.*, 2018) and the official documentation of these approaches[1,2,3,4]. Therefore, More details and comparisons of the new approach and other approaches can be found in Chapters 4 and 5.

6.3 Usability test

This section presents a usability test of VariaMos (version 1.1.0.1). The main idea is to test the VariaMos usability to support the FragOP approach (EQ3). And so, gain insight into how easy or difficult is to follow and understand the FragOP approach.

Usability is defined by the International Standard Organization (ISO 9241-11, 1998) as “the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use”. That means that if

[1] https://www.eclipse.org/aspectj/docs.php
[2] http://ckaestne.github.io/CIDE/
[3] http://deltaj.sourceforge.net/
[4] https://github.com/FeatureIDE/FeatureIDE/wiki/Tutorial

a product (a software tool in our case) does not provide effectiveness, efficiency and satisfaction to its users, it is not usable, and therefore will probably not be used.

We intended to evaluate the usability of the VariaMos tool to support the FragOP approach (EQ3), so, we took as a base two similar studies related to usability testing. First, we analyzed the work of Rabiser *et al.* (2012), which presented an implementation of the capabilities in a configuration tool called DOPLER CW. They performed a qualitative investigation on the usefulness of the tool's capabilities for user guidance in product configuration by involving nine business-oriented experts of two industry partners from the domain of industrial automation. Their tool was also used in software product lines. Second, we analyzed the work of Teruel *et al.* (2014), which presented a usability evaluation of the CSRML tool 2012; which is a Requirements Engineering CASE Tool for the Goal-Oriented Collaborative Systems Requirements Modeling Language (CSRML). They involved 28 fourth-year Computer Science students in the evaluation, which was reported by following the ISO/IEC 25062:2006 Common Industry Format for usability tests.

Subsequently, we decided to develop and conduct a usability test by using the ISO/IEC Common Industry Format (CIF) for usability tests (ISO/IEC 25062, 2006). We also applied three evaluation techniques: (i) one for the definition of the experimental tasks (Condori-Fernández *et al.*, 2013), (ii) another for evaluating user satisfaction by gathering their opinion through a survey (Lund, 2001), and finally a semi-structured interview to enrich this usability test. Eight graduate students participated in this study. They were asked to carry out a set of tasks about a sample of an e-commerce software product line. The usability of the tool was measured by several variables such as tasks completion rate, elapsed time, and a satisfaction questionnaire.

Following we present: (i) product description, (ii) test objectives, (iii) context of use, (iv) usability metrics, (v) usability test process, (vi) results, and (vii) validation threads.

Note: the usability test format completed for this evaluation can be found in an online repository (Correa, 2018). That repository also contains all the documents used in the usability test. Including pre-questionnaire, post-questionnaire, pre-experiment setup A, experiment part A, pre-experiment setup B, experiment part B, and program installation.

6.3.1 Product description

Product name: VariaMos

Product Version: 1.1.0.1

Website: https://variamos.com

GitHub: https://github.com/SPLA/VARIAMOS

VariaMos target users: this tool is intended to be used for researchers, students, software developers and industrials which are interested in the software product line methodology. This includes people with experience in the software product lines field and novice people who are interested in adopting a software product line methodology. Finally, this tool can be also used for people who are interested in component-based software development.

VariaMos assistive technologies: SWI-Prolog, MXGraph.

VariaMos evaluated parts: (a) feature modeling, (b) domain component modeling, (c) binding modeling, (d) product configuration, (e) product derivation, (f) product syntax verification, and (g) product customization. Additionally, we evaluated the “implementing domain components” FragOP activity, which is not supported by VariaMos, but it is very important inside the FragOP process.

6.3.2 Test objectives

As previously stated, the main idea is to test the VariaMos usability to support the FragOP approach (EQ3). This is why we focused on seven VariaMos features as seen in Figure 6-6. Additionally, we consider the domain component implementation, which was not supported by VariaMos, but it is very important inside the FragOP process. So, we plan to test:

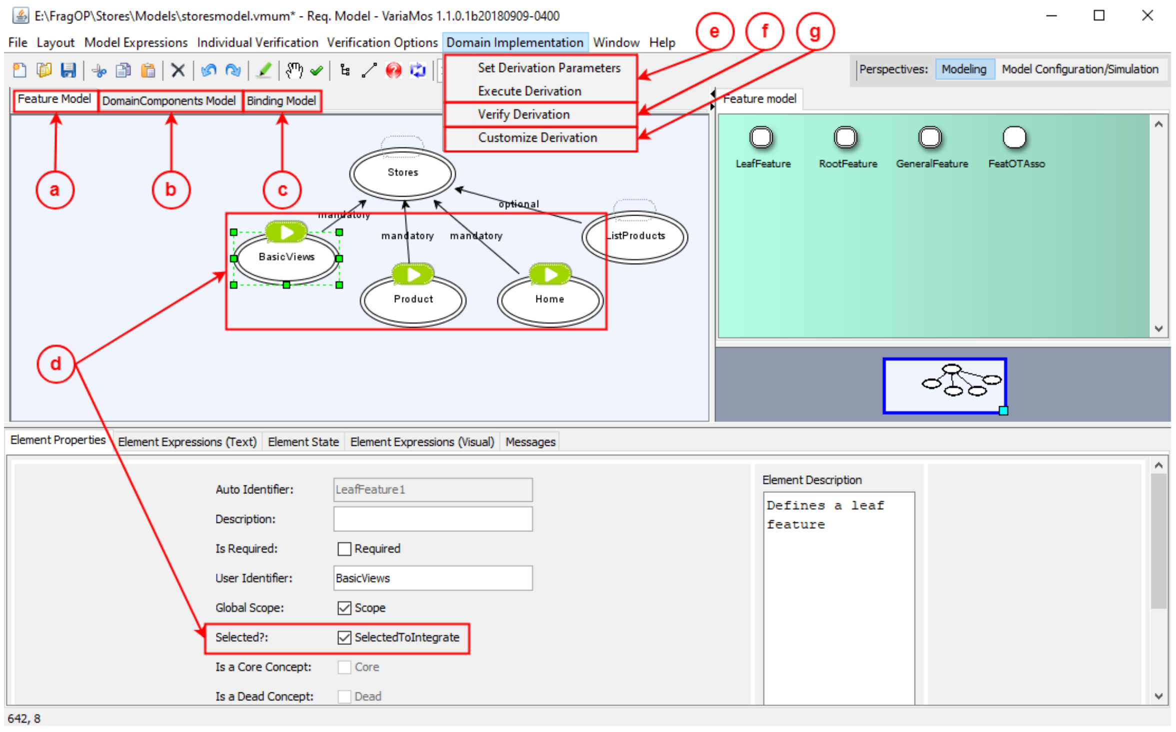
a) **Feature modeling:** the ability of the users to interact and modify the feature models.
b) **Domain component modeling:** the ability of the users to interact and modify the component models.
c) **Binding modeling:** the ability of the users to interact and modify the binding models.
d) **Product configuration:** the ability of the users to interact and create custom product configuration.
e) **Product derivation:** the ability of the users to derive software products and to find product derivation errors.

f) **Product syntax verification:** the ability of the users to verify software products and to find product syntax verification errors.

g) **Product customization:** the ability of the users to customize software products.

h) **Domain component implementation (not supported by VariaMos):** the ability of the users to interact, modify and create domain files.

Figure 6-6: VariaMos tool highlighting the evaluated features



Auto Identifier: LeafFeature1
Description:
Is Required: ☐ Required
User Identifier: BasicViews
Global Scope: ☑ Scope
Selected?: ☑ SelectedToIntegrate
Is a Core Concept: ☐ Core
Is a Dead Concept: ☐ Dead

Element Description
Defines a leaf feature

6.3.3 Context of use

Test facilities

- **Intended context of use:** any workplace with people interested in SPL or component-based software development.
- **Context used for this test:** the usability test was conducted in a laboratory of the *Facultad de Minas* at the *Universidad Nacional de Colombia*. The participants use their own computers with a list of software programs previously installed and configured. They were requested to record their tasks through the use of google drive documents. Additionally, two test administrators were observing and attending

the participants' questions. Finally, we use audio record programs to record the participants' opinions at the end of the experiment.

Participants' computer environment

- **Intended context of use:** VariaMos 1.1.0.1 is intended to use for any PC running Windows or Unix operating systems.
- **Context used for this test:** as previously stated, we requested the participants to bring their own PC. This way we will test how the tool performs over very different PC configurations. Some of the participant PC included: (i) operating systems, such as Windows, Linux, and macOS; (ii) RAM between 2 GB and 12 GB; (iii) display units between 14" and 17"; and (iv) the software programs required to execute the projects. The software programs included: VariaMos 1.1.0.1, Eclipse Oxygen EE 3, Java (JDK 8), Apache Tomcat 8.5.

6.3.4 Usability metrics

A product (or a software tool in our case) must provide effectiveness, efficiency, and satisfaction to its users, based on that, we defined the next usability metrics.

A) Effectiveness

- **Completion rate:** unassisted completion rate was defined as the percentage of participants who completed each task correctly without any assistance from the test administrator. The assisted completion rate was defined as the percentage of participants who completed each task correctly with the test administrator intervention.
- **Errors:** an error was defined as a task completed wrongly or not completed.
- **Assists:** an assist was defined as verbal help given by the test administrators to guide the participant to the next step in completing the task. Test administrators also provide help to the participants to understand a task definition that was not very clear to them, but this type of assists was not recorded.

B) Efficiency

- **Task time:** the amount of time to complete each task.
- **Completion rate efficiency:** mean completion task rate/mean task time.

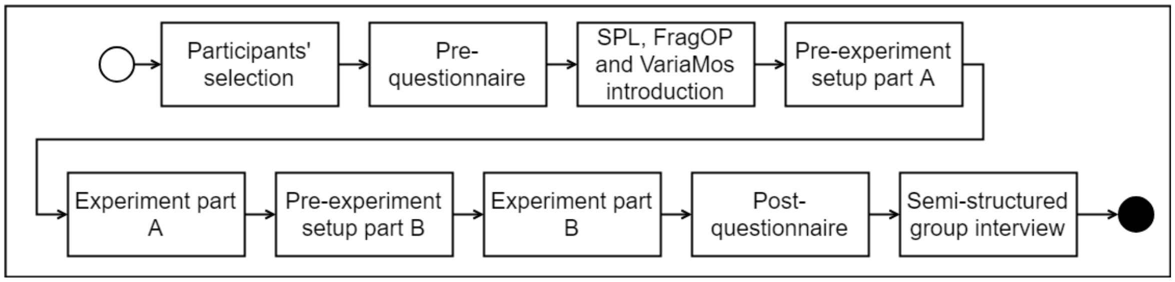
C) Satisfaction

At the end of the experiment, participants completed a post-questionnaire. Subjects rated some questions on a 5-point Likert scale, and there were other questions about VariaMos and FragOP. Scores were given for each participant's perception of ease of use, easy to learn, easy to remember and subjective satisfaction. Therefore, a semi-structured group interview was carried out about tool usability.

6.3.5 Usability test process

The usability test was designed as a process with nine activities (see Figure 6-7), which is described next.

Figure 6-7: Usability test process (UML activity diagram)


Participants' selection
Pre-questionnaire
SPL, FragOP and VariaMos introduction
Pre-experiment setup part A
Experiment part A
Pre-experiment setup part B
Experiment part B
Post-questionnaire
Semi-structured group interview

6.3.5.1 Participants’ selection

Eight graduate students from the *Universidad Nacional de Colombia* participated in this testing. These students were attending a postgraduate course in software modeling. The participants agreed to develop the usability test during two sessions of their software modeling course (each session lasted four hours). We told them that we want to find errors and evaluate the usability of a tool called VariaMos. We told them that their personal information (such as name, email, and company, among others) won’t be exposed so they will appear in the report as participant 1..N. We explained to them that the usability test was about a software tool used to design and implement SPL components, and they will be requested to complete some tasks. Next, participants were asked to sign a document to participate in the experiment.

6.3.5.2 Pre-questionnaire

The usability test started with a request to the participants to complete a pre-questionnaire about their background and software experience (see Appendix C). The pre-questionnaire was designed using a Likert scale (Likert, 1932), which had a five-point format: (1) strongly disagree, (2) somewhat disagree, (3) neither agree nor disagree, (4) some-what agree, and (5) strongly agree. The intention was to collect information about the participants' background and experience and in order to confirm the participants' lack of knowledge with FragOP and VariaMos (see Table 6-4 and Table 6-5). The pre-questionnaire took 15 minutes of the first session.

Table 6-4: Participants' pre-questionnaire information summary first part

Participant	Gender	Job occupation	Professional experience (years)
P1	Male	Software Development and Automatization	3
P2	Male	Software Development	7
P3	Male	Software Development	3
P4	Male	Software Development	3
P5	Male	Web development	3
P6	Male	Software Development and Database Analyst	5
P7	Male	Software Development	8
P8	Male	Web development	3

Table 6-5: Participants' pre-questionnaire information summary second part

Participant	Experience with software development	Experience with component development	Experience with SPL implementation	Experience with VariaMos	Experience with FragOP
P1	5	3	1	1	1
P2	4	3	2	1	1
P3	3	3	1	1	1
P4	4	4	2	1	1
P5	4	4	1	1	1
P6	4	3	1	1	1
P7	4	2	1	1	1
P8	4	3	1	1	1
Mean	4.000	3.125	1.250	1.000	1.000
Standard Deviation	0.535	0.641	0.463	0.000	0.000
Standard Error	0.189	0.227	0.164	0.000	0.000
Min	3	2	1	1	1
Max	5	4	2	1	1

The participants' background and experience were very important to develop the usability test. The pre-questionnaire results showed that participants have knowledge in software development (mean = 4,000; stdev = 0,535) and more or less knowledge in component development (mean = 3,125; stdev = 0,641). The pre-questionnaire also shows that the participants have lack of knowledge in SPL (mean = 1,250; stdev = 0,463), VariaMos (mean = 1; stdev = 0), and FragOP (mean = 1; stdev = 0). The last two results were mandatory to execute this usability test.

Based on the participants' background and experience, these participants could be classified as "novice people who are interested in adopting a software product line methodology" who are one of the VariaMos user target population.

6.3.5.3 SPL, FragOP and VariaMos introduction

After the pre-questionnaire, we started a magistral class. In this class, we presented the main concepts of SPL, FragOP, and VariaMos, and we developed a very small example of the use of FragOP and VariaMos. This introduction was important because the participants didn't have knowledge about SPL, so, we introduced topics, such as software product lines, software product line engineering, feature modeling, domain engineering, application engineering, fragment-oriented programming, and product derivation, among others. The introduction took 3 hours of the first session.

6.3.5.4 Pre-experiment setup part A

The second session started with the "pre-experiment setup part A". Here, the participants were introduced to a document which presented a series of steps to set up an SPL project with the use of VariaMos. Therefore, it showed how to run a derived product in the Eclipse EE environment. This project was the base to carry out the experiment part A. The pre-experiment setup took 30 minutes of the second session.

6.3.5.5 Experiment part A

Once all participants completed the pre-experiment setup part A, we shared with them a personal Google Drive folder with the experiment part A (see Appendix D). Then, they started to complete the five experiment part A tasks. There was a limit of 90 minutes for this part. Additionally, two test administrators were observing and attending the participants' questions. The experiment part A tasks were about:

- **Task 1:** Deriving and customizing a different product based on the pre-experiment base SPL project.
- **Task 2:** Understanding how the fragments work. Here the participant completes some information about the previous derived product.
- **Task 3:** Modifying a domain component. Here the participant tries to figure out how to modify a domain component based on a screenshot of a derived product.
- **Task 4:** Creating a fragmentation point and fragment. Here the participant is requested to create a modification of the component pool which includes a new feature, new component, and new binding, among others.
- **Task 5:** Deriving a new product which includes the previous created feature.

6.3.5.6 Pre-experiment setup part B

After the experiment part A, the participants were requested to follow a "pre-experiment setup part B", here, the participants were introduced to a document which presented a series of steps to set up a new SPL project with the use of VariaMos. The pre-experiment setup took 15 minutes of the second session.

6.3.5.7 Experiment part B

Once all participants completed the pre-experiment setup part B, we shared with them a personal Google Drive folder with the experiment part B (see Appendix E). Then, they started to complete the two experiment part B tasks. There was a limit of 30 minutes for this part. Additionally, two test administrators were observing and attending the participants' questions. The experiment part B tasks were about:

- **Task 1:** Finding and fixing product derivation errors on the pre-experiment base SPL project.
- **Task 2:** Finding and fixing product syntax verification errors on the pre-experiment base SPL project.

6.3.5.8 Post-questionnaire

The participants were submitted to a post-questionnaire, which included questions about (i) experiment environment, (ii) overall satisfaction, (iii) VariaMos and FragOP performance,

(iv) general question, and (v) specific questions about the VariaMos and FragOP theory (see Appendix F). The post-questionnaire was also designed using a Likert scale. The post-questionnaire took 15 minutes of the second session. The post-questionnaire results can be found in Section 6.3.6.

6.3.5.9 Semi-structured group interview

At the end of the testing, a semi-structured group interview with all the participants was carried out. We asked them four open questions about the tool usability, and we recorded the participants' answers. The questions were:

- What did you like?
- What did you dislike/What should be improved?
- What are the opportunities when using this tool in daily business?
- What are the risks when using this tool in daily business?

The group interview results can be found in Section 6.3.6.2.

6.3.6 Results

Next, we discuss the usability test results, based on the performance results and the satisfaction results.

6.3.6.1 Performance results

All the eight participants completed all the seven tasks. Three of them completed all seven tasks without assistance. A total of seven assistances were given to the participants, five of these assistances were requested to the Task 4 – Part A, which was the most the complex task (participants spend a mean of 31 minutes to complete this task). The mean total time to complete all the seven tasks was approximately 72 minutes. There were not errors because all the participants completed all the tasks properly (see Table 6-6).

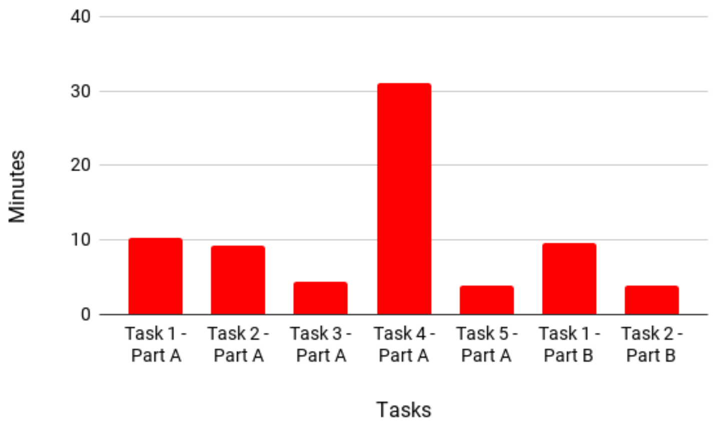
Table 6-6: Participants' performance result summary

Participant	Assisted task completion rate	Unassisted task completion rate	Total Task time	Errors	Assistance	Mean task time	Efficiency
P1	100	100	81	0	2	11.571	8.642
P2	100	100	72	0	1	10.286	9.722

P3	-	100	71	0	0	10.143	9.859
P4	100	100	85	0	1	12.143	8.235
P5	-	100	61	0	0	8.714	11.475
P6	-	100	53	0	0	7.571	13.208
P7	100	100	90	0	2	12.857	7.778
P8	100	100	64	0	1	9.143	10.938
Mean	100.000	100.000	72.125	0.000	0.875	10.304	9.982
Standard Dev	0.000	0.000	12.654	0.000	0.835	1.808	1.826
Standard Error	0.000	0.000	4.474	0.000	0.295	0.639	0.646
Min	100.000	100.000	53	0	0	7.571	7.778
Max	100	100	90	0	2	12.857	13.208

Figure 6-8 shows that the participants spend more time developing Task 4 – Part A (mean of 31 minutes). This task was about creating a fragmentation point and a fragment. This task involved: (i) the creation of a new feature, (ii) the creation of a new domain component and file, (iii) the development of a fragment and a new fragmentation point (without the support of VariaMos), and (iv) the update of the binding model. It also shows that the participants spend little time in the development of Task 5 – Part A and Task 2 – Part B. Task 5 – Part A was about a new product derivation, which took on average approximately 4 minutes; Task 2 – Part B was about finding validation errors, we included a syntax error over a domain file and on average the participants only spend approximately 4 minutes in finding and fixing the error.

Figure 6-8: Participants' average time to complete each task



The results of this study show that all the participants were able to complete a set of tasks in which VariaMos supported the FragOP process. Including:

- Feature modeling, through Task 4 – Part A.
- Domain component modeling, through Task 4 – Part A.

- Binding modeling, through Task 4 – Part A.
- Product configuration, through Task 1 – Part A, Task 5 – Part A.
- Product derivation, through Task 1 – Part A, Task 1 – Part B, Task 5 – Part A.
- Product syntax verification, through Task 2 – Part B.
- Product customization, through Task 1 – Part A.
- Domain component implementation, through Task 2 – Part A, Task 3 – Part A, Task 4 – Part A, Task 1 – Part B, Task 2 – Part B.

Finally, it is important to highlight that all the participants were novice SPL developers and FragOP novice developers. So, the results in this test provide preliminary evidence that VariaMos is a *usable tool* that properly supports the FragOP approach (EQ3). All the participants completed all the tasks (effectiveness), and the mean task time was approximately 10 minutes (efficiency). The tool also provides properly errors notifications that help developers to find fragment errors or domain component errors easily. Task 1 – Part B and Task 2 – Part B were about finding errors in fragments and domain components that even the participants did not develop. The mean for Task 1 – Part B time was approximately 10 minutes, and the mean for Task 2 – Part B time was approximately 4 minutes.

6.3.6.2 Satisfaction results

The satisfaction results were obtained from two sources. First, we analyzed 21 of the 26 post-questionnaire questions. Scores for the 21 questions were given for each participant, based on four usability attributes: ease of use, ease of learning, ease of remembering and subjective satisfaction. It is important to realize that usability is not a single, one-dimensional property of a user interface. Usability has multiple components and is traditionally associated with different usability attributes (Nielsen, 1993). Second, we analyzed the semi-structured interview results which will be presented at the end of this section. Finally, the other five post-questionnaire questions results are used in Section 6.3.7 as a source of information for the validation threads.
The summary of the 21 questions results can be seen in Table 6-7. The highest satisfaction result was about the “ease of use” of VariaMos with a mean of 4.153 (see Figure 6-9). Therefore, on average the participants had 4.6 correct answers of a total of 6 when asked about VariaMos and FragOP functionalities (see Figure 6-10).

- **Ease of use** was calculated as the mean value between the results obtained by the participant in the post-questionnaire “About VariaMos and FragOP performance” (Question VF1 to question VF9). The results show that the most perceived easy FragOP process was binding modeling (mean = 4.375; stdev = 0.518) and the less perceived easy FragOP process were product configuration (mean = 3.625; stdev = 0.916) and domain component implementation (mean = 3.875; stdev = 1.246).
- **Ease of learning** was calculated based on the result obtained by the participant in the post-questionnaire “General Questions” (Question G1).
- **Ease of remembering** was calculated based on the result obtained by the participant in the post-questionnaire “About VariaMos and FragOP performance” (Question VF11).
- **Subjective satisfaction** was calculated as the mean value between the results obtained by the participant in the post-questionnaire “About VariaMos and FragOP performance” (Question VF10 and question VF12) and “General Questions” (Question G2 and Question G3).
- **VariaMos and FragOP correct answers** were calculated as the mean value between the results obtained by the participant in the post-questionnaire “Specific Questions” (Question SQ1 and question SQ6). This also provides support to the “ease of remembering” attribute.

Table 6-7: Participants’ satisfaction result summary

Participant	**Ease of use**	**Ease of learning**	**Subjective satisfaction**	**Ease of remembering**	**VariaMos and FragOP correct answers**
P1	4.2	4.0	2.5	5.0	5.0
P2	4.1	5.0	5.0	4.0	5.0
P3	3.9	4.0	4.0	4.0	5.0
P4	4.0	4.0	3.5	3.0	3.0
P5	5.0	1.0	3.8	4.0	4.0
P6	4.6	5.0	3.3	4.0	5.0
P7	3.8	4.0	3.5	4.0	5.0
P8	3.7	2.0	2.8	3.0	5.0
Mean	4.153	3.625	3.531	3.875	4.625
Standard Dev	0.440	1.408	0.773	0.641	0.744
Standard Error	0.156	0.498	0.273	0.227	0.263
Min	3.7	1.0	2.5	3.0	3.0
Max	5.0	5.0	5.0	5.0	5.0

Figure 6-9: Satisfaction question average results

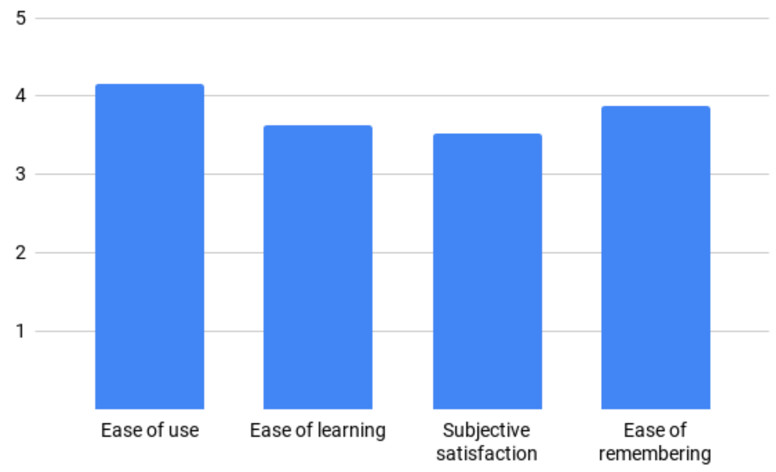
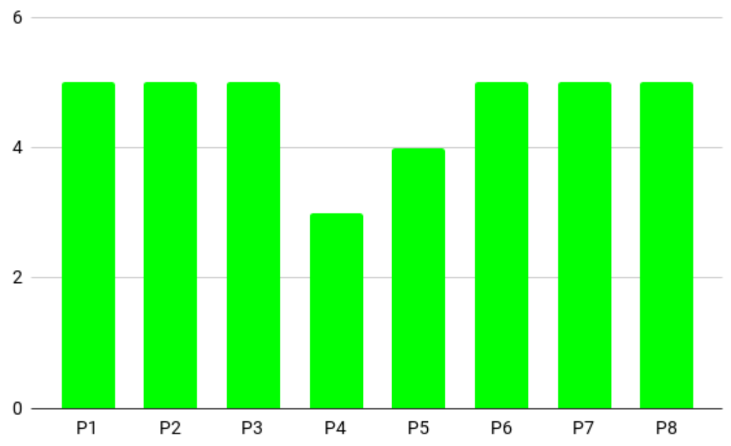


Figure 6-10: Participants' correct answers about VariaMos and FragOP



Finally, the semi-structured interview showed that in general participants liked the software application and saw the potential of this tool and the FragOP approach. They mentioned that is a good strategy to reuse the domain components and assembled them. Some of them think this tool could improve their work at their companies and liked the way the FragOP approach worked.

There were also some recommendations to improve the tool: (i) the graphical interface could be improved. A participant mentioned that future work could be to move the graphical interface into a web project. Allowing to use cellphones or tablet to open the application or to avoid the installation of software programs. And (ii) another participant suggested to automatically generate the component model based on the component pool folder information, which will save time.

6.3.7 Validation threads

The outcome of validation is to gather enough scientific evidence to provide a sound interpretation of the results. Validation threats are issues and scenarios that may distort that evidence and thus incorrectly support (or discard) expected results. Each validation threat should be expected and addressed a priori in order to yield unbiased results or, at least, minimize posterior with effective counter-measures. This section addresses expected validation threats and how these were discarded, while others should be attentively focused in future experiments. Table 6-8 shows the participants post-questionnaire results of some questions which help to discard some threats.

Table 6-8: Participants' post-questionnaire external factors and overall satisfaction result summary

Participant	**EF1**	**EF2**	**EF3**	**OS1**	**OS2**
P1	3	3	2	5	5
P2	4	4	2	5	5
P3	4	4	1	4	4
P4	2	4	3	4	3
P5	1	3	2	4	4
P6	1	4	1	5	5
P7	3	4	3	3	4
P8	2	2	2	4	2
Mean	2.500	3.500	2.000	4.250	4.000
Standard Dev	1.195	0.756	0.756	0.707	1.069
Standard Error	0.423	0.267	0.267	0.250	0.378
Min	1	2	1	3	2
Max	4	4	3	5	5

- **Participants sample**. The number of participants may seem relatively small. However, the ISO/IEC Common Industry Format (CIF) for usability tests states "eight or more subjects are recommended" (ISO/IEC 25062, 2006). Therefore, these participants belong to one of the VariaMos user target population (novice people who are interested in adopting a software product line methodology).
- **Conclusion validity**. There is a threat that many of the results are not based on statistical relationships but on qualitative data. However, given that the main aim of

the study was to study the behavior and opinions of users of a tool, qualitative research methods are well suited. The analysis of the collected data still depends on our interpretation. The work was performed by a single researcher, but the result was carefully checked by two other researchers.

- **Project size**. We selected a basic SPL project due to target users that participated in this usability test (novice people who are interested in adopting a software product line methodology). However, we have shown in previous sections that the tool also works with complex SPL projects (see Section 6.1).
- **Insufficient skills to execute the tasks**. The tasks required participants to have the necessary skill to work with a modeling tool (VariaMos) and to have knowledge in software development. We designed a basic SPL project for the novice SPL developers, and we prepared an introduction to the modeling tool and SPL. This threat was discarded by the participants' pre-questionnaire results, through questions 2 to 6.
- **Lack of documentation and guides**. Due to the length of the tasks, and the fact that participants were novice in SPL, VariaMos, and FragOP, the lack of proper documentation or guides could hinder the outcome. This threat is discarded by post-questionnaire questions OS1 and OS2.
- **External factors**. Despite the noise or disturbance that could be generated in a laboratory by other students, it was necessary to make sure that the experiment environment was not a threat to validity. This threat was analyzed by the post-questionnaire questions EF1, EF2, and EF3. EF2 shows the participants more or less enjoyed modeling and developing in the experiment (mean 3.5). EF3 shows got a little distracted by other colleagues (mean 2), and EF1 shows found the experience environment a little intimidating (mean 2.5). However, the results show the participant completed all the tasks.

6.4 Summary

This chapter presented three evaluations which were designed to confirm how FragOP and VariaMos support the SPL implementation.

The first evaluation showed that VariaMos (FragOP) is expressive enough to implement a real-world, variant-rich multi-language software system. After a code inspection of the result of deriving five different products, we found that multiple assets of different types were automatically assembled and deployed in the respective project folder structure, several LOC were derived and automatically injected, between 21 and 50 lines of code were manually customized (with VariaMos support), and we described some theoretical differences in the case of trying to derive those products with other approaches.

The second evaluation showed the differences between VariaMos (FragOP) and similar SPL implementation mechanism and tools. In this case, we compared VariaMos FragOP with AspectJ (AOP), CIDE (annotative), DeltaJ (DOP), and AHEAD (FOP). We found many differences and some advantages of the FragOP versus the other approaches.

The third evaluation performed a usability test of the VariaMos tool to support the FragOP approach by using the ISO/IEC Common Industry Format (CIF) for usability tests. For the usability test, we defined some experimental tasks, evaluated the user satisfaction and developed a semi-structured interview. Eight graduate students participated in the usability test and were asked to carry out a set of tasks about a sample of an e-commerce software product line. In the end, all the participants completed all the seven tasks, and the mean task time was approximately 10 minutes. The results in this test provide preliminary evidence that VariaMos supports the SPLE and the FragOP approach.

Finally, there is much future work and research to do, such as the development of more tests and experiments under real industrial settings, development of more complex SPL projects under the FragOP approach, and practical comparisons of the FragOP approach and other approaches, among others.

7. Conclusions and future research

The main objective of this thesis is to answer the research question: How can software product line components be automatically assembled independently of their software language in a generic and reusable way? To answer this question, this thesis proposes:

- An SMS in SPL implementation, including SPL component implementation.
- A new SPL implementation approach called FragOP, which is a mix of compositional and annotative approaches. The new approach includes a metamodel and a process to implement SPLs. It also defines two main capabilities, generic assembling, and generic customization.
- A running example to exemplify and demonstrate the use of the new approach.
- A tool to automate the new SPL implementation approach.
- An analysis of multiple product derivations.
- A comparison of the new approach with other current approaches.
- A usability test of the new approach and tool.

7.1 Conclusions

The proposals used to answer the research questions of this thesis are summarized in the next sub-sections.

7.1.1 State of art in SPL implementation

State of the art in SPL implementation including SPL component implementation shows:

- The most discussed SPL implementation approaches are AOP, SOA, annotative approaches, FOP, and DOP; which are commonly grouped in two categories, compositional and annotative approaches.
- There are a few studies that mix compositional and annotative approaches.

- Most of the current approaches use Java as a base to design the SPL components or are attached to a specific software language. However, software applications use on average of 5 different software languages.
- All the current approaches present some limitations, for example, some of them support only coarse-grained extensions, are attached to a specific software language, and include the code variations inside the reusable component files which increases the component complexity, among others.
- There is no approach that allows the generic assembling of SPL components independent of their software language and reduces component complexity.

7.1.2 FragOP SPL implementation approach

This thesis proposes a new SPL implementation approach called FragOP. FragOP is a framework used to design, implement and reuse domain components in the context of an SPL. This framework is a mix of compositional and annotative approaches, which is based on the definition of six fundamental elements: (i) domain components, (ii) domain files, (iii) fragmentation points, (iv) fragments, (v) customization points, and (vi) customization files.

FragOP contains two main capabilities generic assembling and customization. The assembling capability provides an effective and generic way to support component variability and improve product derivation. The customization capability provides an effective and generic way to support component customization and guide SPL developers in the product customization activity. Both capabilities are designed to support multiple software languages, which includes assembly and customization at code level (in a generic way), or in the cases that this is not possible, then assembly and customization at entire file level (in a generic way).

Therefore, we defined the FragOP process which describes the way to implement an SPL under the FragOP approach. This process contains eight activities that go from modeling PL requirements to verify products.

Additionally, a running example was designed as a real-world example, covering most of the problems SPL developers face when implementing an SPL. The use of the running example helps to demonstrate and exemplify the capabilities of the FragOP approach in a practical way.

7.1.3 VariaMos automation

FragOP approach includes the use of six fundamental elements and eight FragOP process activities, and there is a need to automate the FragOP approach, otherwise, the benefits of this approach are dismissed. There is a tool that already supports some of the FragOP process activities such as modeling PL requirements and configuring products; that tool is called VariaMos. VariaMos was enhanced to support seven of the eight FragOP process activities.

VariaMos connects and uses some software tools to support some of the previous activities. For example, (i) the use of MxGraph which is a diagramming library that enables interactive graph and charting applications to be quickly created, which allows creating the different models such as feature models and component models. (ii) The use of ANTLR which is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions. This tool allows us to check the derived component file grammar and execute the product syntax verification activity. And (iii) the use of SWI-Prolog which is a free implementation of the programming language Prolog that allows checking the models' semantics.

7.1.4 FragOP and VariaMos evaluation

Three evaluations were applied to confirm how FragOP and VariaMos support the SPL implementation.

- The first evaluation included the development and analysis of the ClothingStores and a derivation of five products. The results show that VariaMos (FragOP) is expressive enough to implement a real-world, variant-rich multi-language software system. The derivations showed that (i) multiple assets of different types and software languages were automatically assembled and deployed in the respective project folder structure. (ii) Several LOC were derived and automatically injected. And (iii) the product customization was successfully supported. In the end, no single line of code had to be manually modified (without the use of VariaMos) to complete the product derivation; this is an important result that shows that the manual intervention can be reduced at minimum.

- The second evaluation was about SPL implementation mechanisms comparison. It described in detail the differences between FragOP and other approaches, therefore, it highlighted some benefits of the FragOP approach.
- The third evaluation presented and evaluated the usability of the VariaMos tool to support the FragOP approach. This test was applied with the use of the ISO/IEC Common Industry Format (CIF) for usability tests. Eight graduate students participated and carried out a set of tasks about a sample of an e-commerce software product line. The results provide preliminary evidence that VariaMos supports the SPLE and the FragOP approach.

7.2 Future Research

A desirable aspect of any research is that in addition to providing solutions to initial issues or questions, it should identify new research topics that would allow researchers to further work to eventually produce more useful knowledge and progress. This section presents some research directions and required additional work on SPL implementation, component implementation, and also some particular research directions in the improvement of the FragOP approach and VariaMos, including and further validation of the approach presented in this thesis.

7.2.1 Future work in variability modeling

Future works in SPL variability modeling include the following items:

- **Implement complex binding relationships**. Currently, VariaMos (FragOP) only allows a one-to-one binding relationship. One component connected to one feature. Nevertheless, we plan to implement a constraint network (Lecoutre, 2009) to graphically represent more complex domain implementation relationships such as “Domain components C1 or C2, but not both, can be used to implement feature F”.
- **Support more variability models**. There are different types of models that can be used to represent the SPL requirements and variability. Currently, VariaMos (FragOP) only supports feature models, however, we think it is important to support more variability models, such as goals models (Asadi *et al.*, 2011). For example,

goal models allow representing quality attributes (non-functional requirements) which is not supported by the feature models.

- **PLA variability**. The domain components developed for the FragOP approach or for most of the SPL implementation approaches are designed and codified based on a single SPL software architecture. So, implementing and supporting PLA models and refining SPL components to support variability at architectural level arises as an interesting research topic.

7.2.2 Future work in SPL testing

Future works in SPL testing include the following items:

- **Test variability**. SPL components support variability, so, how to design, store, and execute tests that can be applied to variable components is an important research topic.
- **Unit tests, integration tests, and systems tests**. Developing new test approaches that allow FragOP testing at different levels such as at domain and application level is an important future work.

7.2.3 Future work in tool automation

The result of the usability test, some VariaMos developers' suggestions, and the trend to develop applications in the cloud were considered to redesign the VariaMos project as a web application. Web SPL tools present some advantages versus desktop applications, such as distributed computing, responsive design (user interfaces), collaborative modeling, easy maintenance, improve usability, and improve connectivity, among others.

That is the reason why we decided to create a new VariaMos version called VariaMos web[1]. This new version was developed with trending technologies, such as HTML, CSS, JavaScript, Vue.js (a JavaScript framework), MxGraph-JS (the modeling library in which DRAW.IO is based), Java, Spring MVC (a Java framework), and SWI-PROLOG (an implementation of the programming language Prolog).

[1] https://variamos.com/home/variamos-web/

Figures 6-11 and 6-12 show the new VariaMos web user interface. This new version contains all the FragOP functionalities from the VariaMos old version (desktop version), from feature modeling to product derivation. And the source code can be found in two GitHub repositories, one contains the front-end application and the other contains the back-end application. This web version is currently under development and many other functionalities from the VariaMos old version are in process to be included.

Figure 7-1: VariaMos web feature model view

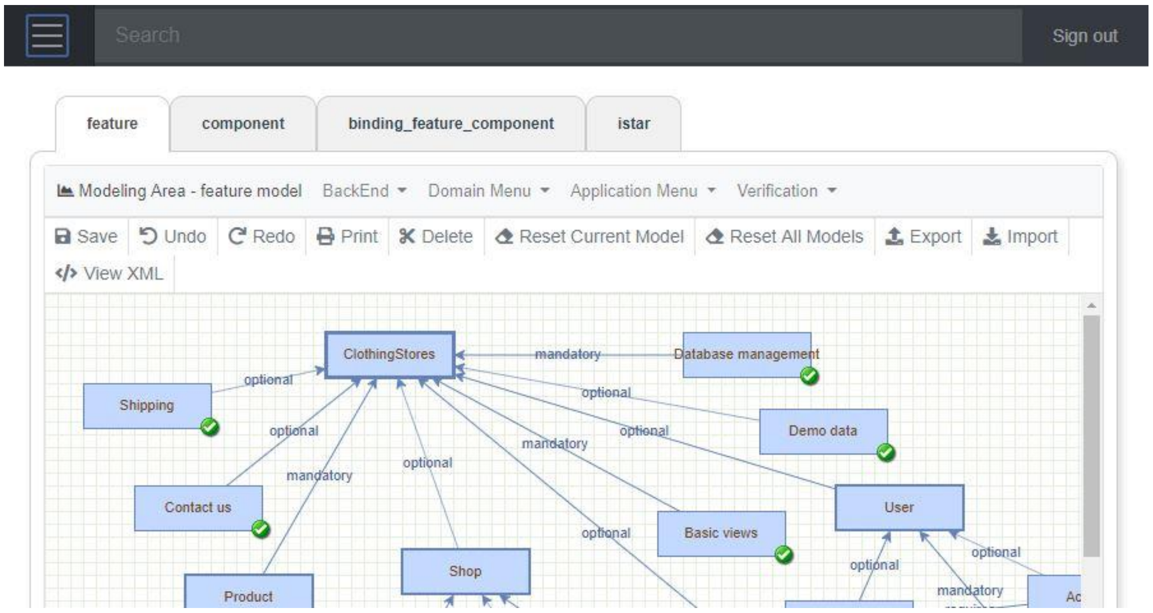
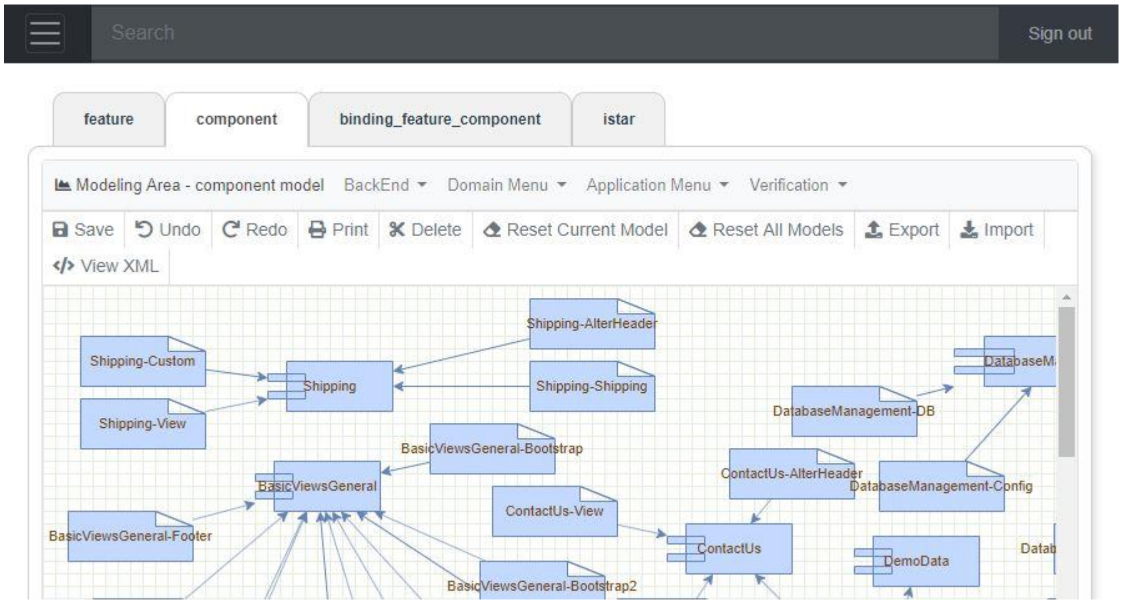


Figure 7-2: VariaMos web component model view



Finally, future works in SPL tool automation include the following items:

- **Support static code analysis.** There are different tools that support static code analysis, such as CheckStyle[1], SonarQube[2], and Infer[3]. We think we can take advantage of some of these tools to analyze the code of the domain and application files, in order to detect potential bugs and issues.
- **Support verification of more languages**. Currently, VariaMos (FragOP) support verification of some software languages, such as Java, PHP, and HTML. However, we plan to extend the support for more languages.
- **Support VariaMos component verification at domain level**. Currently, it is only possible to verify the application files after the product derivation, however, we plan to extend the support to verify the domain component files at domain level with the execution of some integration tests.
- **Improve VariaMos to automate partially the creation of the component model based on the directories**. We plan to use the information of the component pool directory to allow a partial automation of the domain component model creation.
- **Support the storage of component tests**. We also plan to extend the domain component pool structure and domain component model to support sorting and managing component tests.

7.2.4 Future work in evaluation

Future works in SPL implementation approach evaluations include the following items:

- **Test with real companies**. Future work can focus on testing the new approach in real industrial settings. Including the development and implementation of complex software product lines.
- **Complex comparison between the different approaches**. It is important to compare and test the different approaches with practical experiments and evaluations. To gain insights about the support, benefits, and issues with each approach.

[1] https://checkstyle.sourceforge.io/
[2] https://www.sonarqube.org/
[3] https://fbinfer.com/

A. Appendix: Publications

This appendix contains the titles and venues of the publications the Ph.D. research has produced so far:

Journals:

Correa, D., Mazo, R., & Giraldo-Goméz, G.L. (2018). Fragment-oriented programming: a framework to design and implement software product line domain components. DYNA, vol. 85(207), pp. 74-83.

Conferences with proceedings:

Correa, D., Mazo, R., & Giraldo, G. L. (2019, June). Extending FragOP Domain Reusable Components to Support Product Customization in the Context of Software Product Lines. In International Conference on Software Reuse (ICSR), pp. 17-33, Springer, Cham.

Book chapters:

Correa, D., & Mazo, R. (2018). Implementación de componentes reutilizables de dominio. In Guía para la adopción industrial de líneas de productos de software, Editorial Eafit, ISBN 978-958-720-506-0, pp. 307-368, Medellín-Colombia.

Correa, D., & Mazo, R. (2018). Verificación y validación de componentes reutilizables de dominio. In Guía para la adopción industrial de líneas de productos de software, Editorial Eafit, ISBN 978-958-720-506-0, pp. 369-384, Medellín-Colombia.

Correa, D., & Mazo, R. (2018). Selección de los mecanismos de ensamblaje y ensamblaje de los componentes de aplicación. In Guía para la adopción industrial de líneas de productos de software, Editorial Eafit, ISBN 978-958-720-506-0, pp. 459-480, Medellín-Colombia.

Cobaleda, L., Mazo, R., & Correa, D. (2018). Selección, personalización y aumento de componentes reutilizables de dominio para cada aplicación. In Guía para la adopción industrial de líneas de productos de software, Editorial Eafit, ISBN 978-958-720-506-0, pp. 447-458, Medellín-Colombia.

Cortés, A., Mazo, R., & Correa, D. (2018). Pruebas de integración, del sistema, de certificación y de aceptación de los productos derivados de una línea de productos. In Guía para la adopción industrial de líneas de productos de software, Editorial Eafit, ISBN 978-958-720-506-0, pp. 481-502, Medellín-Colombia.

Mazo, R., Noreña, G., Jaramillo, L., & Correa, D. (2018a). Marco de referencia para la adopción y la gestión de líneas de productos de software. In Guía para la adopción industrial de líneas de productos de software, Editorial Eafit, ISBN 978-958-720-506-0, pp. 39-58, Medellín-Colombia.

Other publications developed during the PhD program, but non-related with the thesis topic

Alvarez, D., Correa, D., & Arango, F. (2016, April). An analysis of XSS, CSRF and SQL injection in Colombian software and web site development. In 2016 8th Euro American Conference on Telematics and Information Systems (EATIS), pp. 1-5, IEEE.

Correa, D., Arango, F., Mazo, R., & Giraldo, G. L. (2018, June). CME – A Web Application Framework Learning Technique Based on Concerns, Micro-Learning and Examples. In International Conference on Web Engineering, pp. 17-32, Springer, Cham.

Lozano, S., Monsalve, E. S., Vallejo, P., Mazo, R., & Correa, D. Comparando dos Estrategias de Aprendizaje Activo para Enseñar SCRUM en un Curso Introductorio de

Ingeniería de Software. [To appear in the INGENIARE journal, Volumen 28 N°1 2019-2020, ISSN: 0718-3305].

Monsalve, E. S., Vallejo, P., Mazo, R., & Correa, D. (2017). Transparency as a learning strategy to teach Software Engineering. In Proceeding of the 12 Colombian Conference on Computing (CCC), Cali, Colombia.

B. Appendix: List of Systematic Mapping Study selected studies

The following contains the list of all the 88 SMS selected studies.

#	Reference
S1	Heider, W., Vierhauser, M., Lettner, D., & Grunbacher, P. (2012). A case study on the evolution of a component-based product line. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), pp. 1–10. IEEE Computer Society, Washington.
S2	Heymans, P., Boucher, Q., Classen, A., Bourdoux, A., & Demonceau, L. (2012). A code tagging approach to software product line development. International Journal On Software Tools For Technology Transfer, vol. 14(5), pp. 553-566.
S3	Shen, L., Peng, X., & Zhao, W. (2009, April). A comprehensive feature-oriented traceability model for software product line development. Australian Software Engineering Conference (ASWEC), pp. 210-219, IEEE.
S4	Falvo, V., Duarte Filho, N. F., Oliveira, E., & Barbosa, E. F. (2014, October). A contribution to the adoption of software product lines in the development of mobile learning applications. Frontiers in Education Conference (FIE), pp. 1-8, IEEE.
S5	Lee, J., Muthig, D., & Naab, M. (2010). A feature-oriented approach for developing reusable product line assets of service-based systems. Journal of Systems and Software, vol. 83(7), pp. 1123-1136.
S6	Alzahmi, S. M., Abu-Matar, M., & Mizouni, R. (2014, April). A Practical Tool for Automating Service Oriented Software Product Lines Derivation. International Symposium on Service Oriented System Engineering (SOSE), pp. 90-97, IEEE.
S7	Deelstra, S., Sinnema, M., & Bosch, J. (2004). A Product Derivation Framework for Software Product Families. Software Product-Family Engineering, pp. 473-484.
S8	Karam, M., Dascalu, S., Safa, H., Santina, R., & Koteich, Z. (2008). A product-line architecture for web service-based visual composition of web applications. Journal of Systems and Software, vol. 81(6), pp. 855-867.
S9	Usman, M., Iqbal, M. Z., & Khan, M. U. (2017). A product-line model-driven engineering approach for generating feature-based mobile applications. Journal of Systems and Software, vol. 123, pp. 1-32.

S10	Go, G., Kang, S., & Ahn, J. (2015, June). A software binding application tool based on the orthogonal variability description language for software product line development. International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), pp. 1-8, IEEE.
S11	Lee, J., Kang, S., & Lee, D. (2012, September). A survey on software product line testing. Software Product Line Conference (SPLC), vol. 1, pp. 31-40, ACM.
S12	Neto, P. A. D. M. S., do Carmo Machado, I., McGregor, J. D., De Almeida, E. S., & de Lemos Meira, S. R. (2011). A systematic mapping study of software product lines testing. Information and Software Technology, vol. 53(5), pp. 407-423.
S13	Laguna, M. A., & Crespo, Y. (2013). A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. Science of Computer Programming, vol. 78(8), pp. 1010-1034.
S14	Khoshnevis, S. (2012, June). An approach to variability management in service-oriented product lines. International Conference on Software Engineering (ICSE), pp. 1483-1486. IEEE.
S15	Tizzei, L. P., Rubira, C. M., & Lee, J. (2012, September). An aspect-based feature model for architecting component product lines. EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 85-92. IEEE.
S16	Ribeiro, H. B. G., de Lemos Meira, S. R., de Almeida, E. S., Lucredio, D., Alvaro, A., Alves, V., & Garcia, V. C. (2010, September). An assessment on technologies for implementing core assets in service-oriented product lines. Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), pp. 90-99, IEEE.
S17	Abu-Matar, M., & Gomaa, H. (2013, March). An automated framework for variability management of service-oriented software product lines. International Symposium on Service Oriented System Engineering (SOSE), pp. 260-267, IEEE.
S18	Anastasopoulos, M., & Muthig, D. (2004). An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. Software Reuse: Methods, Techniques, And Tools, pp. 141-156.
S19	Parra, C., Joya, D., Giral, L., & Infante, A. (2014, March). An SOA approach for automating software product line adoption. Symposium on Applied Computing (ASC), pp. 1231-1238, ACM.
S20	Cu, C., & Zheng, Y. (2016, May). Architecture-centric derivation of products in a software product line. International Workshop on Modeling in Software Engineering (MiSE), pp. 27-33, IEEE/ACM.
S21	Kim, K., Kim, H., Ahn, M., Seo, M., Chang, Y., & Kang, K. C. (2006, May). ASADAL: a tool system for co-development of software and test environment based on product line engineering. International conference on Software engineering (ICSE), pp. 783-786, ACM.
S22	Andrade, R., Rebêlo, H., Ribeiro, M., & Borba, P. (2013, September). AspectJ-based idioms for flexible feature binding. Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), pp. 59-68, IEEE.
S23	Anthonysamy, P., & Somé, S. S. (2008, March). Aspect-oriented use case modeling for software product lines. AOSD workshop on Early aspects, ACM.

S24	Lesaint, D., & Papamargaritis, G. (2004, June). Aspects and constraints for implementing configurable product-line architectures. Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 135-144, IEEE.
S25	Lee, K., Botterweck, G., & Thiel, S. (2009, July). Aspectual separation of feature dependencies for flexible feature composition. Computer Software and Applications Conference (COMPSAC), pp. 45-52, IEEE.
S26	Altintas, N. I., Surav, M., Keskin, O., & Cetin, S. (2005, September). Aurora software product line. Turkish Software Architecture Workshop, Ankara.
S27	Muhammad, R., & Setyautami, M. R. A. (2016, October). Automatic model translation to UML from software product lines model using UML profile. International Conference on Advanced Computer Science and Information Systems (ICACSIS), pp. 605-610, IEEE.
S28	Miranda Filho, S., Mariano, H., Kulesza, U., & Batista, T. (2010, September). Automating Software Product Line Development: A Repository-Based Approach. Conference on Software Engineering and Advanced Applications (SEAA), pp. 141-144, IEEE.
S29	Cirilo, E., Nunes, I., Kulesza, U., & Lucena, C. (2012). Automating the product derivation process of multi-agent systems product lines. Journal of Systems and Software, vol. 85(2), pp. 258-276.
S30	Pessoa, L., Fernandes, P., Castro, T., Alves, V., Rodrigues, G. N., & Carvalho, H. (2017). Building reliable and maintainable Dynamic Software Product Lines: An investigation in the Body Sensor Network domain. Information and Software Technology, vol. 86, pp. 54-70.
S31	Peña, J. (2005, September). Can agent oriented software engineering be used to build MASs product lines?. Workshop on Radical Agent Concepts (WRAC), pp. 98-108, Springer Berlin Heidelberg.
S32	Rosenmüller, M., Siegmund, N., Saake, G., & Apel, S. (2008, October). Code generation to support static and dynamic composition of software product lines. International conference on Generative programming and component engineering (GPCE), pp. 3-12, ACM.
S33	Mohabbati, B., Asadi, M., Gašević, D., Hatala, M., & Müller, H. A. (2013). Combining service-orientation and software product line engineering: A systematic mapping study. Information and Software Technology, vol. 55(11), pp. 1845-1859.
S34	Abdelmoez, W., Khater, H., & El-shoafy, N. (2012, May). Comparing maintainability evolution of object-oriented and aspect-oriented software product lines. International Conference on Informatics and Systems (INFOS), pp. SE-53, IEEE.
S35	Tizzei, L. P., Dias, M., Rubira, C. M., Garcia, A., & Lee, J. (2011). Components meet aspects: Assessing design stability of a software product line. Information and Software Technology, vol. 53(2), pp. 121-136.
S36	Nascimento, L. M., de Almeida, E. S., & de Lemos Meira, S. R. (2009). Cores assets development in software product lines-towards a practical approach for the mobile game domain. Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS), pp. 124-137.
S37	Schaefer, I., Bettini, L., Bono, V., Damiani, F., & Tanzarella, N. (2010). Delta-Oriented Programming of Software Product Lines. Software Product Lines: Going Beyond, pp. 77-91.

S38	de Jonge, M. (2009). Developing product lines with third-party components. Electronic Notes in Theoretical Computer Science, vol. 238(5), pp. 63-80.
S39	Günther, S., & Sunkle, S. (2010, October). Dynamically adaptable software product lines using Ruby metaprogramming. International Workshop on Feature-Oriented Software Development (FOSD), pp. 80-87, ACM.
S40	Capilla, R., & Dueñas, J. C. (2005). Evolution and Maintenance of Web Sites: A Product Line Model. Managing Corporate Information Systems Evolution and Maintenance, pp. 255-271.
S41	Tesanovic, A. (2007, March). Evolving embedded product lines: opportunities for aspects. Workshop on Aspects, components, and patterns for infrastructure software (ACP4IS), ACM.
S42	Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., & Dantas, F. (2008, May). Evolving software product lines with aspects: an empirical study on design stability. International conference on Software engineering (ICSE), pp. 261-270, ACM/IEEE.
S43	El-Sharkawy, S., Kröher, C., Eichelberger, H., & Schmid, K. (2015, October). Experience from implementing a complex eclipse extension for software product line engineering. Eclipse Technology eXchange, pp. 13-18, ACM.
S44	Peng, X., Shen, L., & Zhao, W. (2008). Feature Implementation Modeling Based Product Derivation in Software Product Line. High Confidence Software Reuse in Large Systems, pp. 142-153.
S45	Amja, A. M., Obaid, A., Mili, H., & Jarir, Z. (2016, November). Feature-Based Adaptation and Its Implementation. International Conference on Collaboration and Internet Computing (CIC), pp. 321-328, IEEE.
S46	Lee, K., Botterweck, G., & Thiel, S. (2009, May). Feature-modeling and aspect-oriented programming: Integration and automation. International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing (SNPD), pp. 186-191, IEEE.
S47	Günther, S., & Sunkle, S. (2009, October). Feature-oriented programming with Ruby. International Workshop on Feature-Oriented Software Development (FOSD), 11-18, ACM.
S48	Seidl, C., Schuster, S., & Schaefer, I. (2017). Generative software product line development using variability-aware design patterns. Computer Languages, Systems & Structures, vol. 48, pp. 89-111.
S49	Mefteh, M., Bouassida, N., & Ben-Abdallah, H. (2015, April). Implementation and evaluation of an approach for extracting feature models from documented UML use case diagrams. Symposium on Applied Computing (SAC), pp. 1602-1609, ACM.
S50	Dordowsky, F., Bridges, R., & Tschope, H. (2011, August). Implementing a software product line for a complex avionics system. Software Product Line Conference (SPLC), pp. 241-250, IEEE.
S51	Geertsema, B., & Jansen, S. (2010, August). Increasing software product reusability and variability using active components: a software product line infrastructure. European Conference on Software Architecture (ECSA), pp. 336-343, ACM.
S52	Cirilo, E., Kulesza, U., Coelho, R., de Lucena, C., & von Staa, A. (2008). Integrating Component and Product Lines Technologies. High Confidence Software Reuse in Large Systems, pp. 130-141.

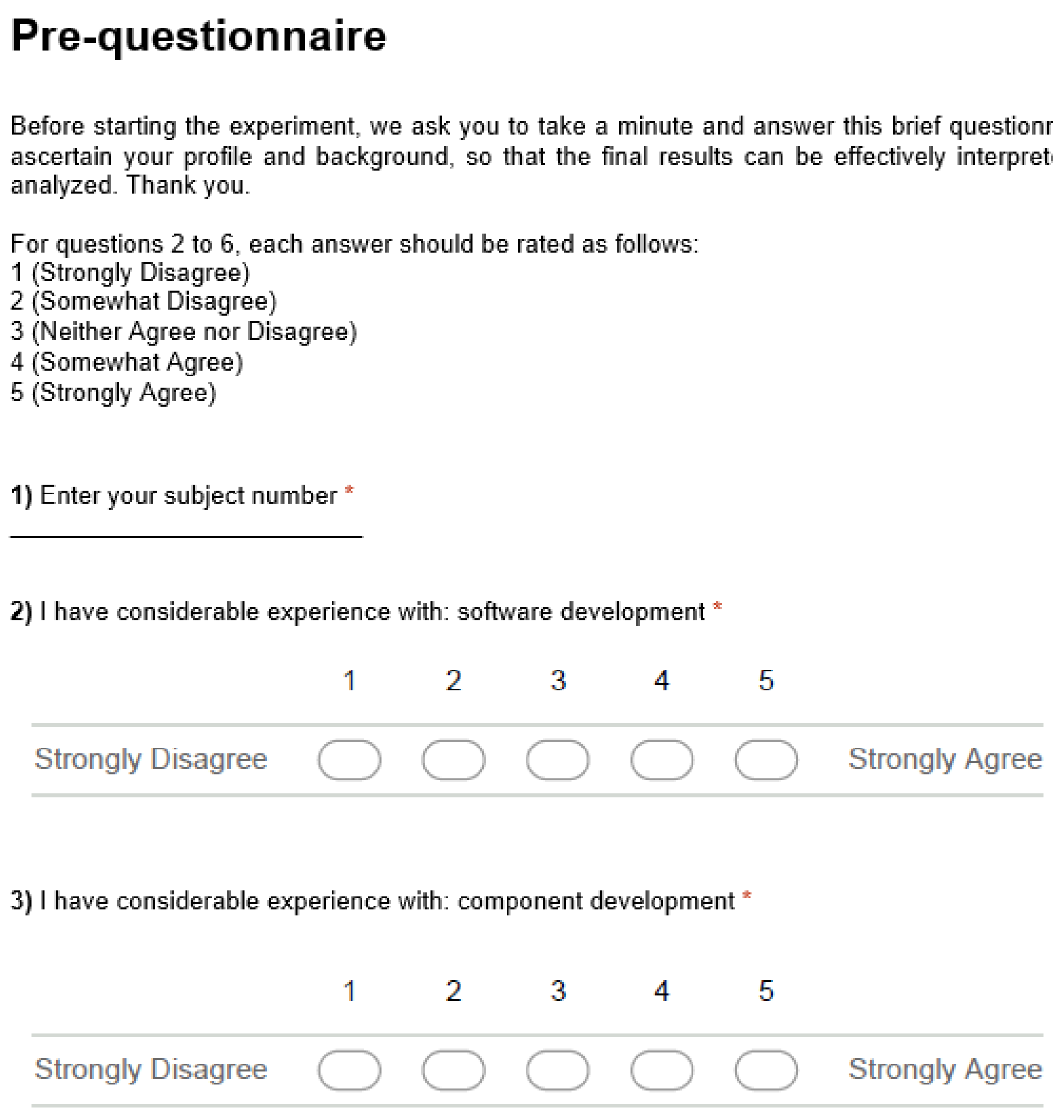
S53	Gurgel, A., Dantas, F., Garcia, A., & Sant'Anna, C. (2012, July). Integrating Software Product Lines: A Study of Reuse versus Stability. Computer Software and Applications Conference (COMPSAC), pp. 89-98, IEEE.
S54	Afzal, U., Mahmood, T., & Shaikh, Z. (2016). Intelligent software product line configurations: A literature review. Computer Standards & Interfaces, vol. 48, pp. 30-48.
S55	Méndez-Acuna, D., Galindo, J. A., Degueule, T., Combemale, B., & Baudry, B. (2016). Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review. Computer Languages, Systems & Structures, vol. 46, pp. 206-235.
S56	Freeman, G., Batory, D., & Lavender, G. (2008). Lifting Transformational Models of Product Lines: A Case Study. Theory And Practice Of Model Transformations, pp. 16-30.
S57	McRitchie, I., Brown, T., & Spence, I. (2004). Managing Component Variability within Embedded Software Product Lines via Transformational Code Generation. Software Product-Family Engineering, pp. 98-110.
S58	Thao, C. (2012, June). Managing evolution of software product line. International Conference on Software Engineering (ICSE), pp. 1619-1621, IEEE.
S59	Zhang, J., Cai, X., & Liu, G. (2008, December). Mapping features to architectural components in aspect-oriented software product lines. International Conference on Computer Science and Software Engineering (CSSE), vol. 2, pp. 94-97, IEEE.
S60	Kulesza, U., Alves, V., Garcia, A., Neto, A., Cirilo, E., de Lucena, C., & Borba, P. (2007). Mapping Features to Aspects: A Model-Based Generative Approach. Early Aspects: Current Challenges and Future Directions, pp. 155-174.
S61	Buchmann, T., Dotor, A., & Westfechtel, B. (2013). MOD2-SCM: A model-driven product line for software configuration management systems. Information and Software Technology, vol. 55(3), pp. 630-650.
S62	Sun, C. A., Rossing, R., Sinnema, M., Bulanov, P., & Aiello, M. (2010). Modeling and managing the variability of Web service-based systems. Journal of Systems and Software, vol. 83(3), pp. 502-516.
S63	Carvalho, M. L. L., Gomes, G. S. D. S., Da Silva, M. L. G., Machado, I. D. C., & de Almeida, E. S. (2016, September). On the Implementation of Dynamic Software Product Lines: A Preliminary Study. Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), pp. 21-30, IEEE.
S64	Barros, F. J. (2013, April). On the representation of product lines using pluggable software units: results from an exploratory study. Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium (TMS-DEVS), Society for Computer Simulation International.
S65	Ferreira, G. C. S., Gaia, F. N., Figueiredo, E., & de Almeida Maia, M. (2014). On the use of feature-oriented programming for evolving software product lines—A comparative study. Science of Computer Programming, vol. 93, pp. 65-85.
S66	Adachi Barbosa, E., Batista, T., Garcia, A., & Silva, E. (2011). PL-AspectualACME: An Aspect-Oriented Architectural Description Language for Software Product Lines. Software Architecture, pp. 139-146.

S67	Voelter, M., & Groher, I. (2007, September). Product line implementation using aspect-oriented and model-driven software development. Software Product Line Conference (SPLC), pp. 233-242, IEEE.
S68	Capilla, R., & Topaloglu, N. Y. (2005, September). Product lines for supporting the composition and evolution of service oriented applications. International Workshop on Principles of Software Evolution (IWPSE), pp. 53-56, IEEE.
S69	Caporuscio, M., Muccini, H., Pelliccione, P., & Di Nisio, E. (2006). Rapid System Development Via Product Line Architecture Implementation. Rapid Integration Of Software Engineering Techniques, pp. 18-33.
S70	Heo, S. H., & Choi, E. M. (2006, August). Representation of variability in software product line using aspect-oriented programming. International Conference on Software Engineering Research, Management and Applications (SERA), pp. 66-73, IEEE.
S71	Montalvillo, L., & Díaz, O. (2016). Requirement-driven evolution in software product lines: A systematic mapping study. Journal of Systems and Software, vol. 122, pp. 110-143.
S72	Derakhshanmanesh, M., Fox, J., & Ebert, J. (2014). Requirements-driven incremental adoption of variability management techniques and tools: an industrial experience report. Requirements Engineering, vol. 19(4), pp. 333-354.
S73	Santos, A. R., do Carmo Machado, I., & de Almeida, E. S. (2016, September). RiPLE-HC: Javascript systems meets SPL composition. Systems and Software Product Line Conference (SPLC), pp. 154-163, ACM.
S74	Mohamed, F., Abu-Matar, M., Mizouni, R., Al-Qutayri, M., & Al Mahmoud, Z. (2014, December). SaaS Dynamic Evolution Based on Model-Driven Software Product Lines. 6th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 292-299, IEEE.
S75	Pietsch, C., Kehrer, T., Kelter, U., Reuling, D., & Ohrndorf, M. (2015, November). SiPL--A Delta-Based Modeling Framework for Software Product Line Engineering. International Conference on Automated Software Engineering (ASE), pp. 852-857, IEEE/ACM.
S76	Mohabbati, B., Asadi, M., Gašević, D., & Lee, J. (2014). Software Product Line Engineering to Develop Variant-Rich Web Services. Web Services Foundations, pp. 535-562.
S77	Vale, T., de Almeida, E. S., Alves, V., Kulesza, U., Niu, N., & de Lima, R. (2017). Software product lines traceability: A systematic mapping study. Information and Software Technology, vol. 84, pp. 1-18.
S78	Parra, C., & Joya, D. (2015). SPLIT: an automated approach for enterprise product line adoption through SOA. Journal of Internet Services and Information Security, vol. 5(1), pp. 29-52.
S79	Groher, I., & Weinreich, R. (2013, January). Supporting variability management in architecture design and implementation. Hawaii International Conference on System Sciences (HICSS), pp. 4995-5004, IEEE.
S80	Carromeu, C., Paiva, D., & Cagnin, M. (2015). The Evolution from a Web SPL of the e-Gov Domain to the Mobile Paradigm. International Conference on Computational Science And Its Applications (ICCSA), pp. 217-231.
S81	Liu, J. J., Lutz, R. R., & Rajan, H. (2006, October). The role of aspects in modeling product line variabilities. Workshop on Aspect-oriented Product Line Engineering (AOPLE), pp. 32-39.

S82	dos Santos Rocha, R., & Fantinato, M. (2013). The use of software product lines for business process management: A systematic literature review. Information and Software Technology, vol. 55(8), pp. 1355-1373.
S83	Lago, P., Niemela, E., & Van Vliet, H. (2004, March). Tool support for traceable product evolution. Conference on Software Maintenance and Reengineering (CSMR), pp. 261-269, IEEE.
S84	Zheng, Y., & Cu, C. (2016, May). Towards implementing product line architecture. Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities (BRIDGE), pp. 5-10, IEEE/ACM.
S85	de Moraes, A. L., Brito, R. D. C., Junior, A. C. C., Ramos, M. C., Colanzi, T. E., Gimenes, I. M. D. S., & Masiero, P. C. (2010, November). Using aspects and the spring framework to implement variabilities in a software product line. International Conference of the Chilean Computer Science Society (SCCC), pp. 71-80, IEEE.
S86	Hartmann, H., Keren, M., Matsinger, A., Rubin, J., Trew, T., & Yatzkar-Haham, T. (2013). Using MDA for integration of heterogeneous components in software supply chains. Science of Computer Programming, vol. 78(12), pp. 2313-2330.
S87	Mazo, R., Assar, S., Salinesi, C., & Hassen, N. B. (2014). Using Software Product Line to improve ERP Engineering: literature review and analysis. Latin American Journal of Computing Faculty of Systems Engineering National Polytechnic School Quito-Ecuador, vol. 1(1), pp. 10.
S88	Humblet, M., Tran, D. V., Weber, J. H., & Cleve, A. (2016, May). Variability management in database applications. Workshop on Variability and Complexity in Software Design (VACE), pp. 21-27, ACM.

C. Appendix: Pre-questionnaire

The following is a copy of the pre-questionnaire handed to the participants before the beginning of the usability test.



Pre-questionnaire

Before starting the experiment, we ask you to take a minute and answer this brief questionnaire to ascertain your profile and background, so that the final results can be effectively interpreted and analyzed. Thank you.

For questions 2 to 6, each answer should be rated as follows:
1 (Strongly Disagree)
2 (Somewhat Disagree)
3 (Neither Agree nor Disagree)
4 (Somewhat Agree)
5 (Strongly Agree)

1) Enter your subject number *

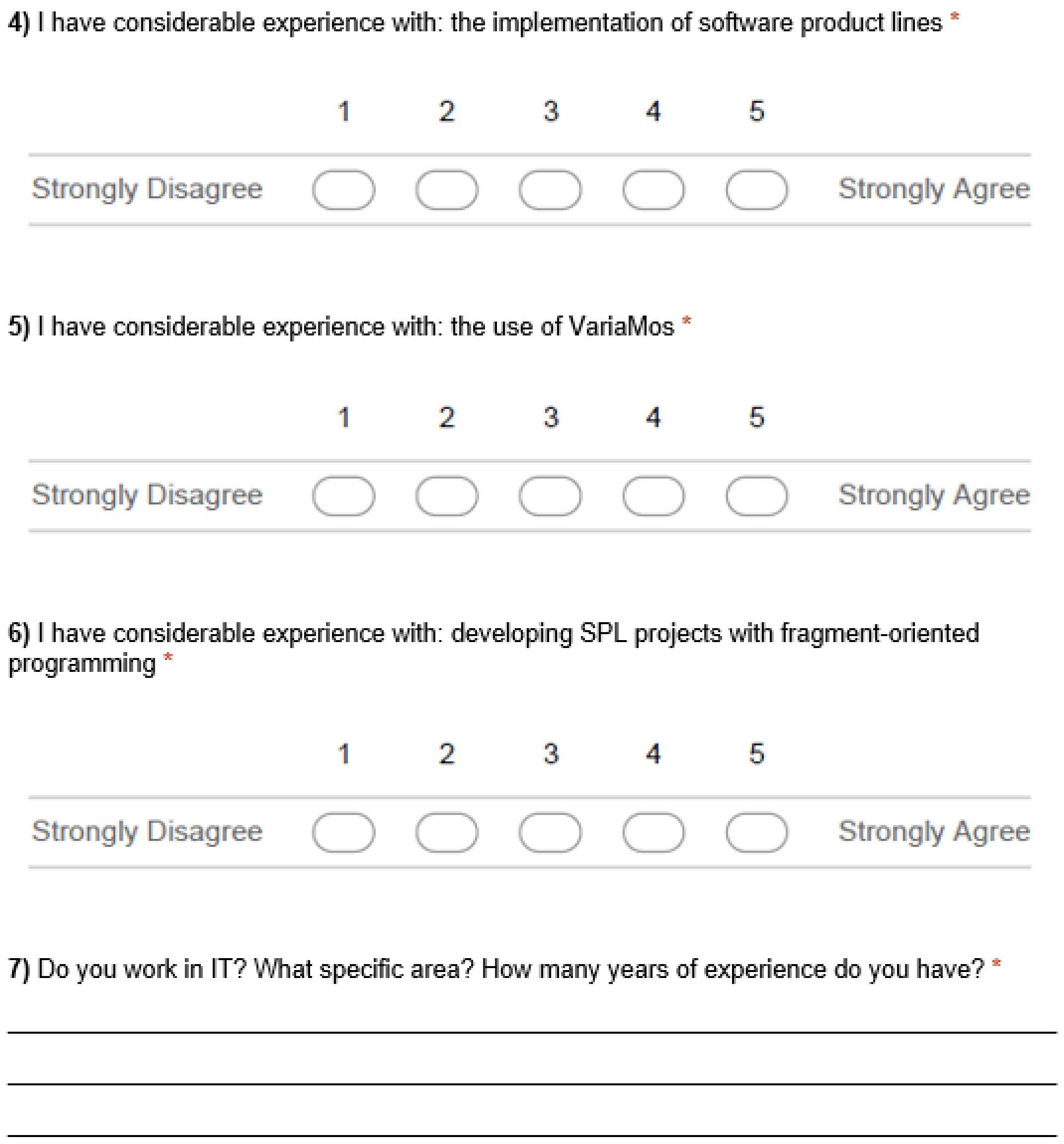
____

2) I have considerable experience with: software development *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

3) I have considerable experience with: component development *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree



4) I have considerable experience with: the implementation of software product lines *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

5) I have considerable experience with: the use of VariaMos *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

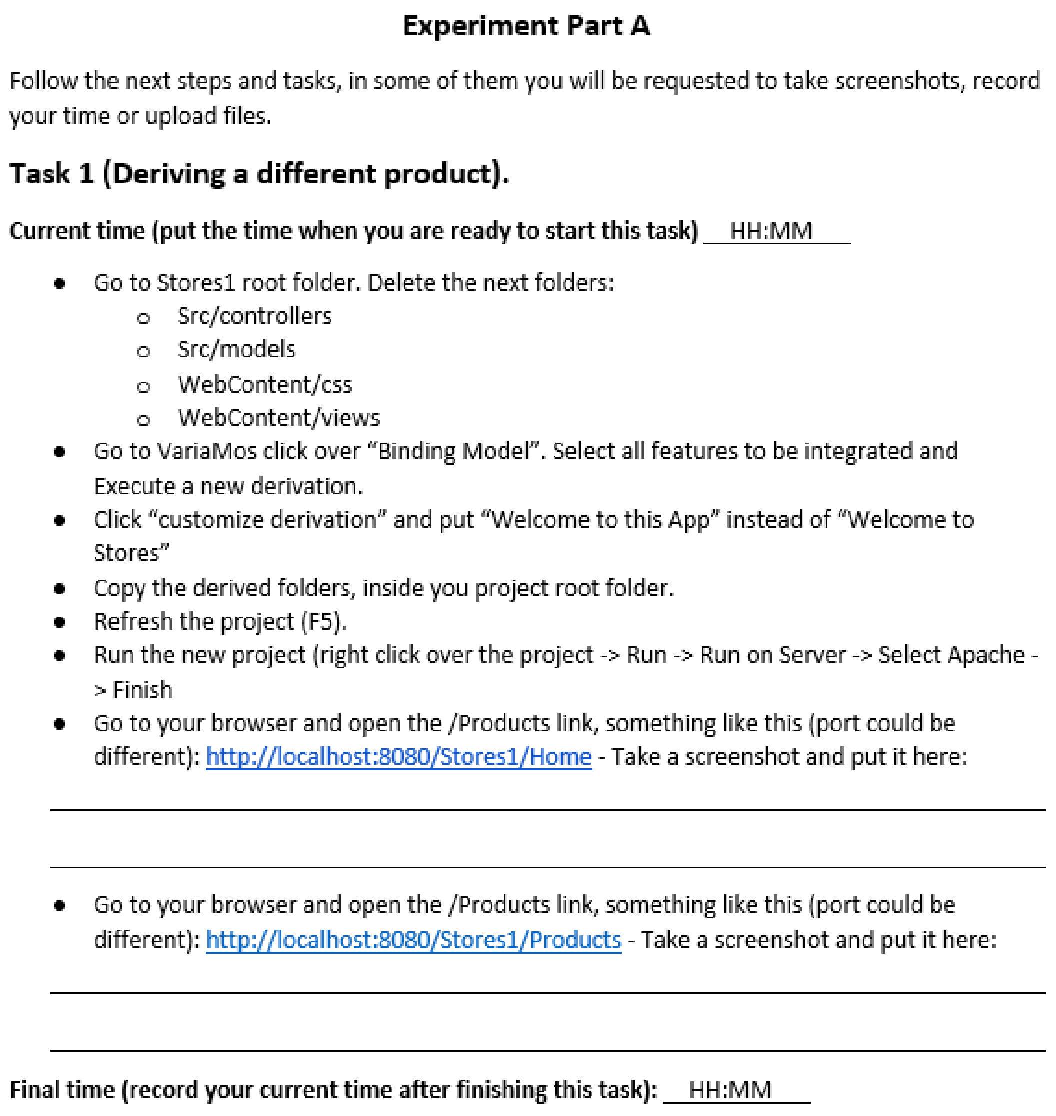
6) I have considerable experience with: developing SPL projects with fragment-oriented programming *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

7) Do you work in IT? What specific area? How many years of experience do you have? *

D. Appendix: Experiment Part A

The following is a copy of the Experiment Part A handed to the participants during the usability test.



Experiment Part A

Follow the next steps and tasks, in some of them you will be requested to take screenshots, record your time or upload files.

Task 1 (Deriving a different product).

Current time (put the time when you are ready to start this task) __HH:MM___

- Go to Stores1 root folder. Delete the next folders:
  - Src/controllers
  - Src/models
  - WebContent/css
  - WebContent/views
- Go to VariaMos click over "Binding Model". Select all features to be integrated and Execute a new derivation.
- Click "customize derivation" and put "Welcome to this App" instead of "Welcome to Stores"
- Copy the derived folders, inside you project root folder.
- Refresh the project (F5).
- Run the new project (right click over the project -> Run -> Run on Server -> Select Apache -> Finish
- Go to your browser and open the /Products link, something like this (port could be different): http://localhost:8080/Stores1/Home - Take a screenshot and put it here:

- Go to your browser and open the /Products link, something like this (port could be different): http://localhost:8080/Stores1/Products - Take a screenshot and put it here:

Final time (record your current time after finishing this task): __HH:MM___

Task 2 (Understanding fragments).

Current time (put the time when you are ready to start this task) __HH:MM___

- Open your WebContent/views/header.jsp and inspect the code
- What is the name of the component(s) which make alterations over the header.jsp file?

____

____

- What is the name of the fragmentation point involved in those alterations?

____

- What do those alterations mean, what did they added?

____

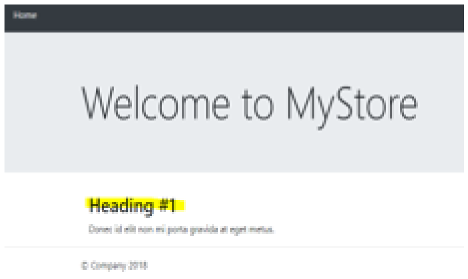
____

Final time (record your current time after finishing this task): __HH:MM___

Task 3 (Modifying a domain component).

Current time (put the time when you are ready to start this task) __HH:MM___

- Suppose you are requested to make this change directly over your pool of components.
- Replace the text "Heading #1" with "Shipping info"



- Try to find inside your component pool what file is responsible for showing that message and make the requested modification. Take a screenshot of your new component file code and put it here:

____

____

Final time (record your current time after finishing this task): __HH:MM___

Task 4 (Creating a new fragmentation point and fragment).

Current time (put the time when you are ready to start this task) __HH:MM___

- Suppose that you are requested to add a new feature and a new component.
- New feature will be called Discount (optional).
- New component will be called Discount.
- If this feature is selected to integrated will make the next change over the derived product:

This is the derived Product.Java getPrice method (by default)

```
public double getPrice() {
    return price;
}
```

This should be the derived Product.Java getPrice method (if Discount feature was selected to integrate)

```
public double getPrice() {
    return price*0.8;
}
```

- Try to create a fragment (inside the Discount component folder) and a fragmentation point (Over the Product.java file) which satisfy the previous requirement. Take a screenshot of your new fragment file code and put it here:

  ____

  ____

  Take a screenshot of the Product.java file and put it here:

  ____

  ____

  Take a screenshot of your new feature model (VariaMos) and put it here:

  ____

  ____

  Take a screenshot of your new domain components model (VariaMos) and put it here:

  ____

  ____

Take a screenshot of your new binding model (VariaMos) and put it here:

Final time (record your current time after finishing this task): HH:MM

Task 5 (Deriving a new product).

Current time (put the time when you are ready to start this task) HH:MM

- Go to Stores1 root folder. Delete the next folders:
    - Src/controllers
    - Src/models
    - WebContent/css
    - WebContent/views
- Go to VariaMos click over "Binding Model". Select all features to be integrated (including Discount) and Execute a new derivation.
- Copy the derived folders, inside you project root folder.
- Refresh the project (F5).
- Run the new project (right click over the project -> Run -> Run on Server -> Select Apache -> Finish
- Go to your browser and open the /Products link, something like this (port could be different): http://localhost:8080/Stores1/Products - Take a screenshot and put it here:

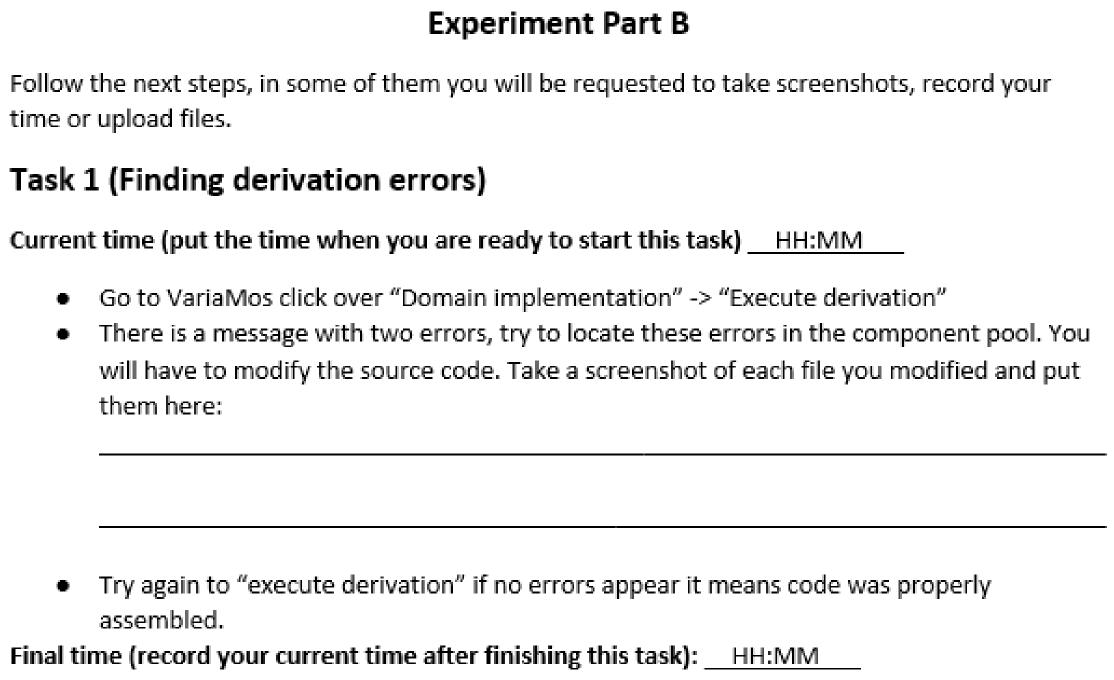
Final time (record your current time after finishing this task): HH:MM

FINALLY

Compress your component pool in a .rar file and upload it to your google drive subject folder.

E. Appendix: Experiment Part B

The following is a copy of the Experiment Part B handed to the participants during the usability test.



Experiment Part B

Follow the next steps, in some of them you will be requested to take screenshots, record your time or upload files.

Task 1 (Finding derivation errors)

Current time (put the time when you are ready to start this task) __HH:MM___

- Go to VariaMos click over "Domain implementation" -> "Execute derivation"
- There is a message with two errors, try to locate these errors in the component pool. You will have to modify the source code. Take a screenshot of each file you modified and put them here:

  ____

  ____

- Try again to "execute derivation" if no errors appear it means code was properly assembled.

Final time (record your current time after finishing this task): __HH:MM____

Task 2 (Finding validation errors)

Current time (put the time when you are ready to start this task) __HH:MM___

- Go to VariaMos click over “Domain implementation” -> “Verify derivation”
- There is a message with one error, try to locate that error in the **component pool.** You will have to modify the source code.
- Once you find the error and once you modified the source code, click “Execute derivation” again and the click “Domain implementation” -> “Verify derivation”. If no errors found, then take a screenshot of the file you modified and put it here:

  ____

  ____

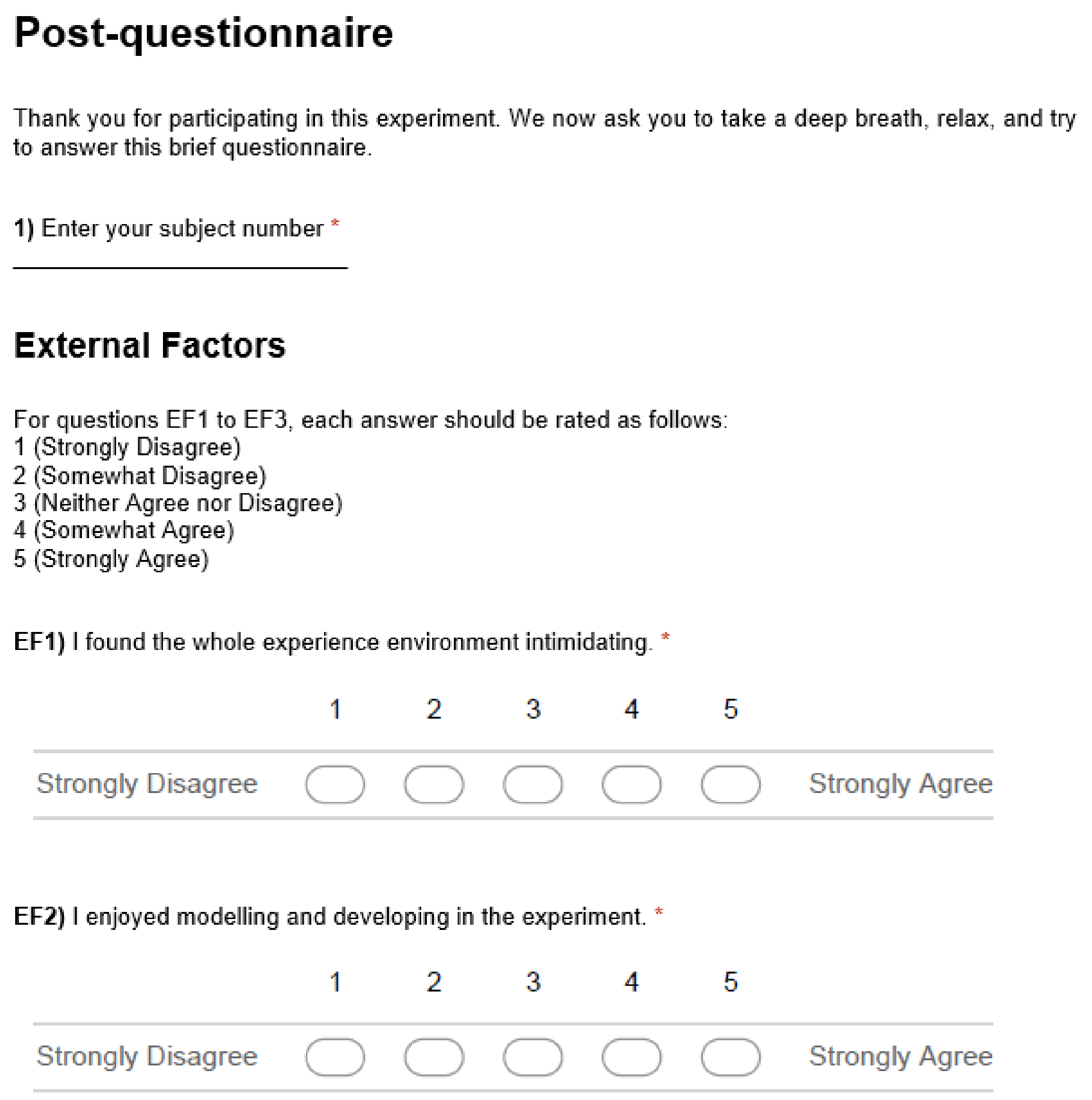
Final time (record your current time after finishing this task): __HH:MM___

FINALLY

Compress your component pool in a .rar file and upload it to your google drive subject folder.

F. Appendix: Post-questionnaire

The following is a copy of the post-questionnaire handed to the participants at the end of the usability test.



Post-questionnaire

Thank you for participating in this experiment. We now ask you to take a deep breath, relax, and try to answer this brief questionnaire.

1) Enter your subject number *

____

External Factors

For questions EF1 to EF3, each answer should be rated as follows:
1 (Strongly Disagree)
2 (Somewhat Disagree)
3 (Neither Agree nor Disagree)
4 (Somewhat Agree)
5 (Strongly Agree)

EF1) I found the whole experience environment intimidating. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

EF2) I enjoyed modelling and developing in the experiment. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

EF3) I kept getting distracted by other colleagues. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

Overall satisfaction

For questions OS1 to OS2, each answer should be rated as follows:
1 (Strongly Disagree)
2 (Somewhat Disagree)
3 (Neither Agree nor Disagree)
4 (Somewhat Agree)
5 (Strongly Agree)

OS1) Overall, this particular setup was suitable for solving every task presented. *

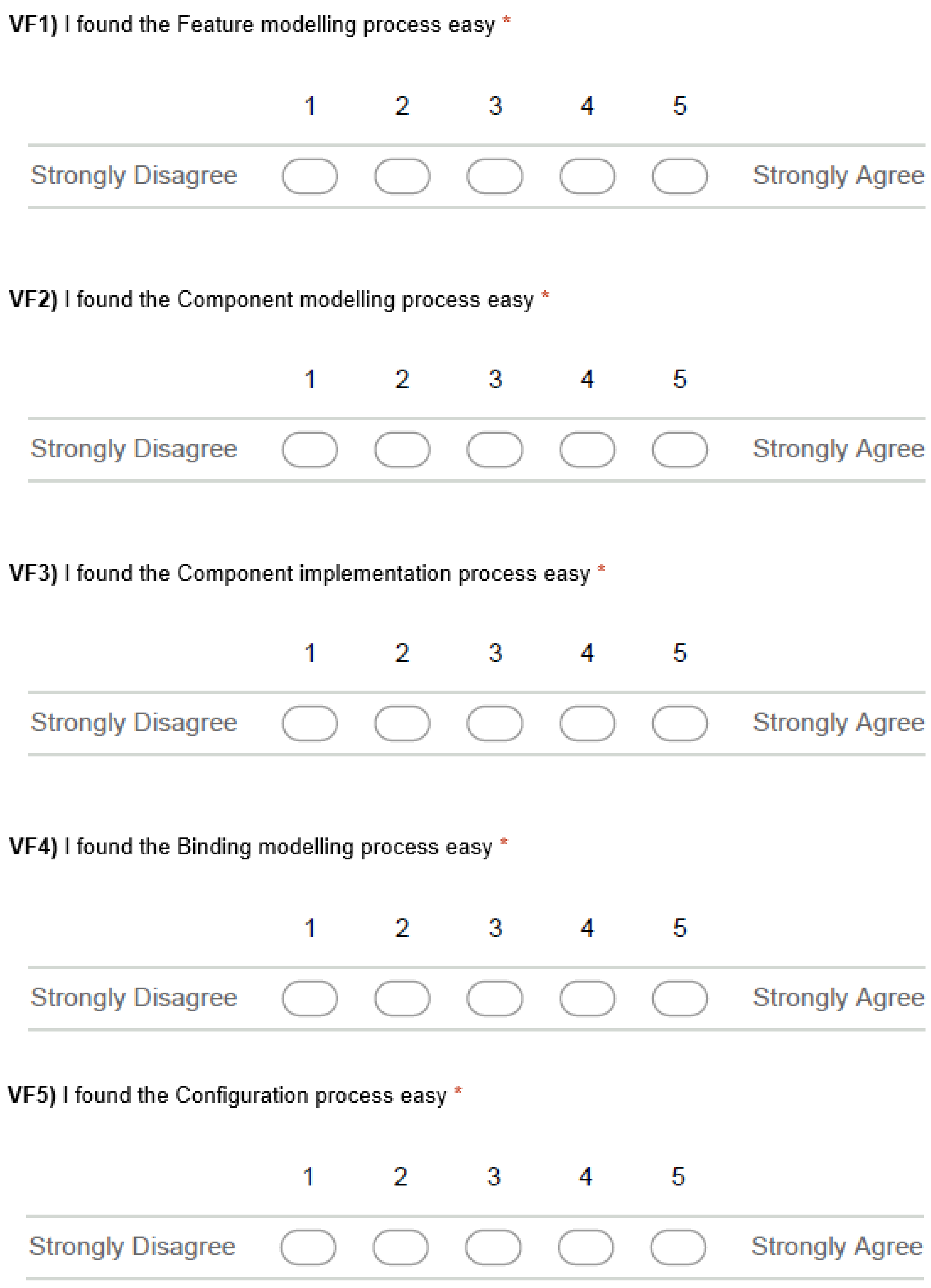
	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

OS2) I found the documentation available to be sufficient. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

About VariaMos and FragOP performance

For questions VF1 to VF12, each answer should be rated as follows:
1 (Strongly Disagree)
2 (Somewhat Disagree)
3 (Neither Agree nor Disagree)
4 (Somewhat Agree)
5 (Strongly Agree)



VF1) I found the Feature modelling process easy *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

VF2) I found the Component modelling process easy *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

VF3) I found the Component implementation process easy *

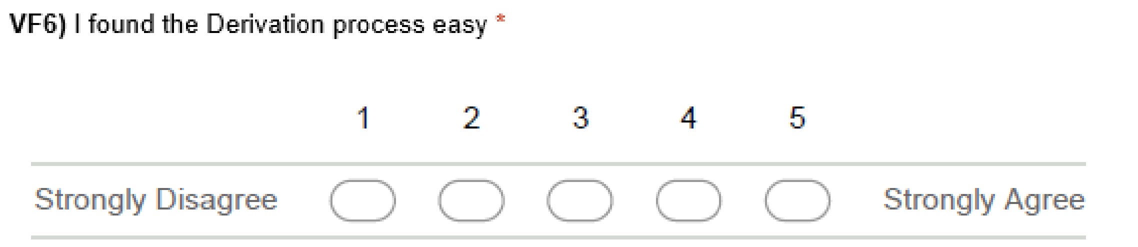
	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

VF4) I found the Binding modelling process easy *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

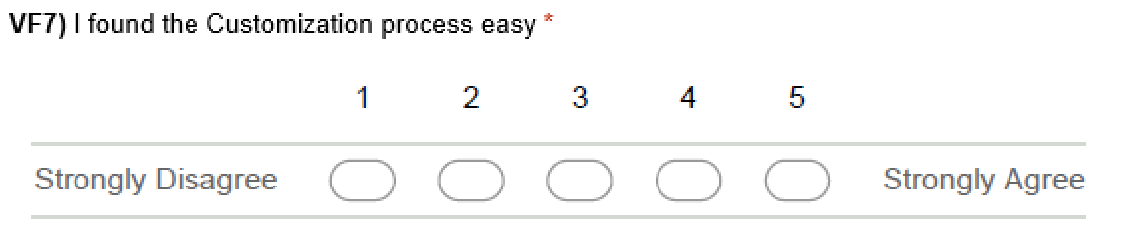
VF5) I found the Configuration process easy *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree



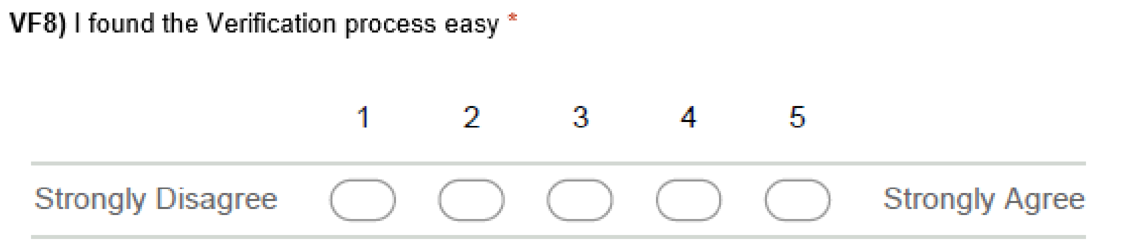
VF6) I found the Derivation process easy *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree



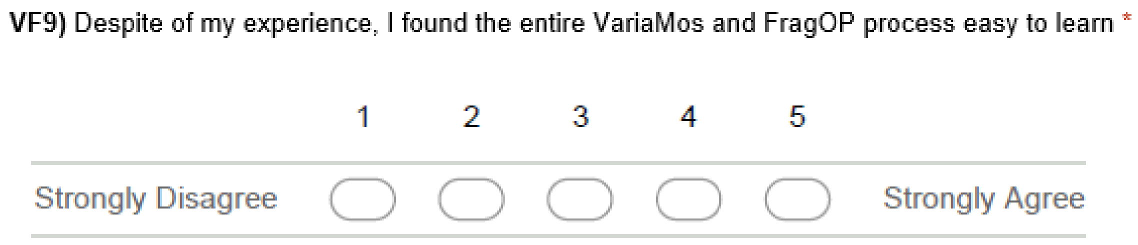
VF7) I found the Customization process easy *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree



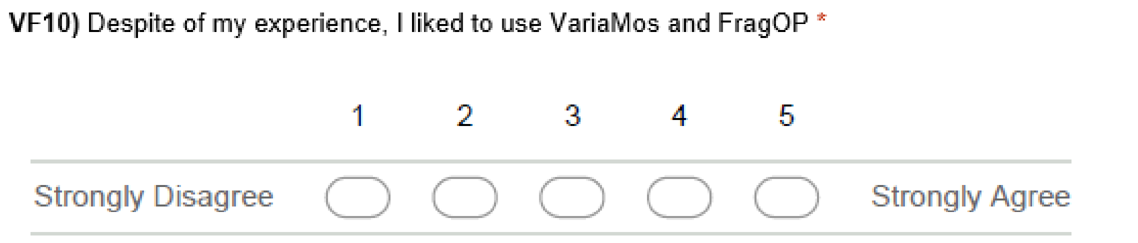
VF8) I found the Verification process easy *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree



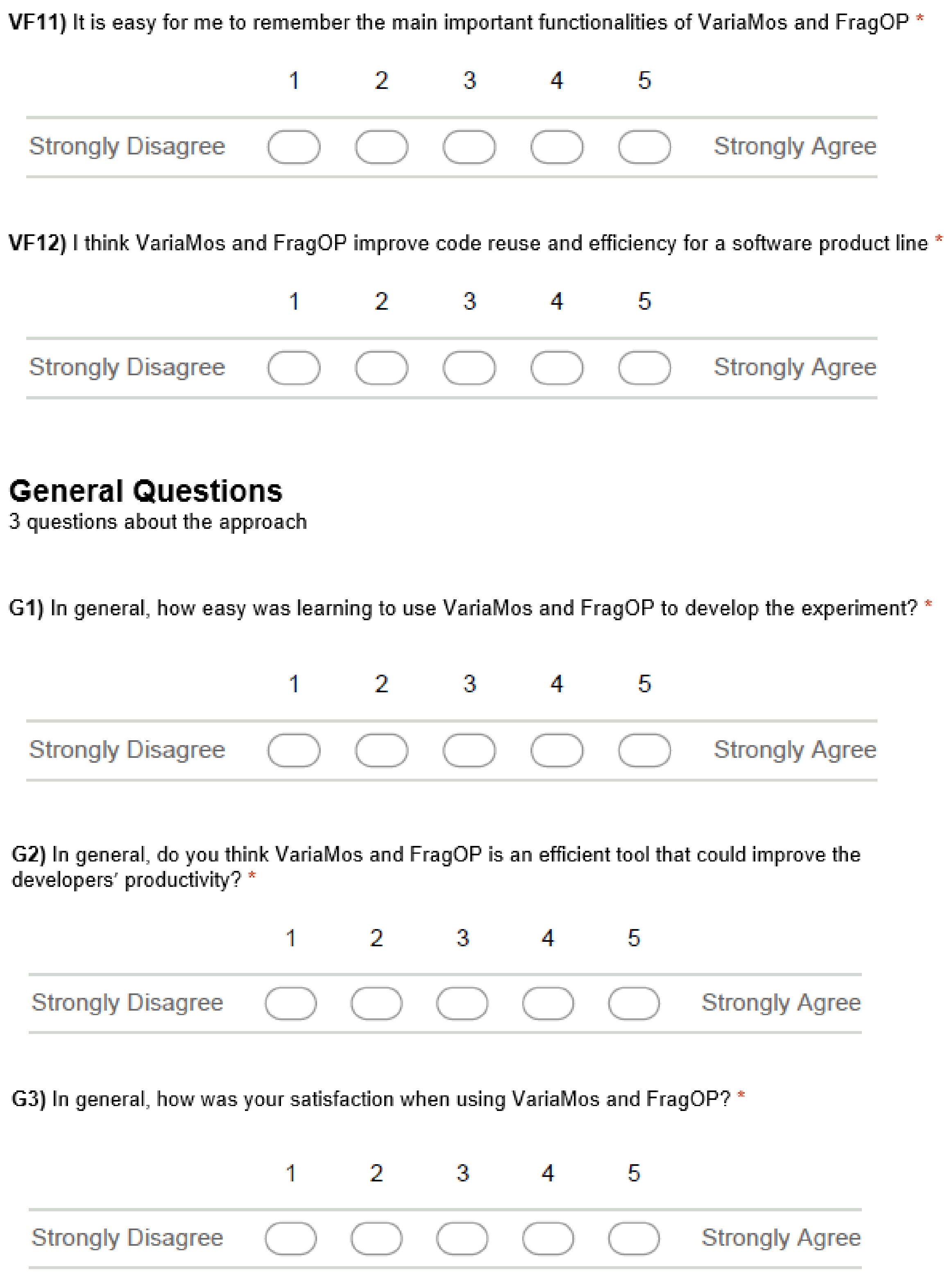
VF9) Despite of my experience, I found the entire VariaMos and FragOP process easy to learn *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree



VF10) Despite of my experience, I liked to use VariaMos and FragOP *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree



VF11) It is easy for me to remember the main important functionalities of VariaMos and FragOP *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

VF12) I think VariaMos and FragOP improve code reuse and efficiency for a software product line *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

General Questions

3 questions about the approach

G1) In general, how easy was learning to use VariaMos and FragOP to develop the experiment? *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

G2) In general, do you think VariaMos and FragOP is an efficient tool that could improve the developers' productivity? *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

G3) In general, how was your satisfaction when using VariaMos and FragOP? *

	1	2	3	4	5	
Strongly Disagree	◯	◯	◯	◯	◯	Strongly Agree

Specific Questions

6 Specific questions about VariaMos and FragOP

SQ1) How many different model views have VariaMos and FragOP (component-based project)? *

- 3 model views
- 2 model views
- 1 model view

SQ2) What is the name of the model were components and features are linked? *

____

SQ3) What is the name of the element that allows to define "variation points"? *

- Fragment
- Fragmentation point
- Domain component

SQ4) A fragment only allows to inject code in Java *

- True
- False

SQ5) The option "set derivation parameters" allows to define what models should be loaded *

- True
- False

SQ6) The "customization points" must contain a begin section (BCP) and an end section (ECP) *

- True
- False

Leave a comment (optional)

Thank you so much for completing this questionnaire

If you wish to leave any further comments, please use the following space

____

____

____

References

Aleixo, F. A., Kulesza, U., & Junior, E. A. O. (2013). Modeling variabilities from software process lines with compositional and annotative techniques: A quantitative study. In: Int. Conf. on Product Focused Software Process Improvement, pp. 153-168, Springer, Berlin.

Alzahmi, S., Matar, M. A., & Mizouni, (2014). R. A Practical Tool for Automating Service Oriented Software Product Lines Derivation. 8th Int. Symposium on Service Oriented System Engineering (SOSE), pp. 90-97, IEEE.

Apel, S., Batory, D., Kästner, C., & Saake, G. (2013). Feature-oriented software product lines. Springer-Verlag Berlin An.

Asadi, M., Bagheri, E., Gašević, D., Hatala, M., & Mohabbati, B. (2011, March). Goal-driven software product line engineering. In Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 691-698, ACM.

Azanza, M., Díaz, O., & Trujillo, S. (2010, July). Software factories: describing the assembly process. In International Conference on Software Process, pp. 126-137, Springer, Berlin, Heidelberg.

Behringer, B., & Rothkugel, S. (2016, April). Integrating feature-based implementation approaches using a common graph-based representation. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, pp. 1504-1511, ACM.

Beuche, D. (2008, September). Modeling and building software product lines with pure::variants. In 12th International Software Product Line Conference, pp. 358-358, IEEE.

Beuche, D., & Dalgarno, M. (2007). Software product line engineering with feature models. Overload Journal, vol. 78, pp. 5-8.

Bradley, A., & Manna, Z. (2007). The Calculus of Computation - Decision Procedures with Applications to Verification. ISBN 978-3-540-74112-1, Springer Berlin Heidelberg New York.

Cetina, C., Giner, P., Fons, J., & Pelechano, V. (2013). Prototyping Dynamic Software Product Lines to evaluate run-time reconfigurations. Science of Computer Programming, vol. 78(12), pp. 2399-2413.

Chen, L., & Babar, M. A. (2011). A systematic review of evaluation of variability management approaches in software product lines. Information and Software Technology, vol. 53(4), pp. 344-362.

Cirilo, E., Kulesza, U., & Lucena, C. (2007) GenArch: A Model-Based Product Derivation Tool. In: Proceedings of Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS 2007), Campinas – Brazil.

Clements, P., & Northrop, L. (2001). Software product lines: practices and patterns. Addison-Wesley.

Cobaleda, L., Mazo, R., & Correa, D. (2018). Selección, personalización y aumento de componentes reutilizables de dominio para cada aplicación. In Guía para la adopción industrial de líneas de productos de software. Editorial Eafit, ISBN 978-958-720-506-0, pp. 447-458, Medellín-Colombia.

Condori-Fernández, N., Panach Navarrete, J. I., Baars, A. I., Vos, T. E., & Pastor López, O. (2013). An empirical approach for evaluating the usability of model-driven tools. In Science of computer programming, vol. 78(11), pp. 2245-2258, Elsevier.

Correa, D. (2018). FragOP-Thesis GitHub repository, Available at: https://github.com/danielgara/FragOP-thesis

Correa, D., & Mazo, R. (2018). Implementación de componentes reutilizables de dominio. In Guía para la adopción industrial de líneas de productos de software. Editorial Eafit, ISBN 978-958-720-506-0, pp. 307-368, Medellín-Colombia.

Correa, D., Mazo, R., & Giraldo, G. L. (2019, June). Extending FragOP Domain Reusable Components to Support Product Customization in the Context of Software Product Lines. In International Conference on Software Reuse, pp. 17-33, Springer, Cham.

Correa, D., Mazo, R., & Giraldo-Goméz, G.L. (2018). Fragment-oriented programming: a framework to design and implement software product line domain components. DYNA, vol. 85(207), pp. 74-83.

de Souza, L. O., O'Leary, P., de Almeida, E. S., & de Lemos Meira, S. R. (2015). Product derivation in practice. Information and Software Technology, vol. 58, pp. 319-337.

Deelstra, S., Sinnema, M., & Bosch, J. (2005). Product derivation in software product families: a case study. Journal of Systems and Software, vol. 74(2), pp. 173-194.

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

Dyba, T., Dingsoyr, T., & Hanssen, G. K. (2007, September). Applying systematic reviews to diverse study types: An experience report. In First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), pp. 225-234, IEEE.

Engström, E., & Runeson, P. (2011). Software product line testing–a systematic mapping study. Information and Software Technology, vol. 53(1), pp. 2-13.

Epifani, I., Ghezzi, C., Mirandola, R., & Tamburrelli, G. (2009, May). Model evolution by run-time parameter adaptation. In Proceedings of the 31st International Conference on Software Engineering, pp. 111-121, IEEE Computer Society.

Heaven, W., & Finkelstein, A. (2004). UML profile to support requirements engineering with KAOS. IEE Proceedings-Software, vol. 151(1), pp. 10-27.

Heidenreich, F., Savga, I., & Wende, C. (2008, September). On Controlled Visualisations in Software Product Line Engineering. In Software Product Line Conference, pp. 335-341.

Horcas, J. M., Cortiñas, A., Fuentes, L., & Luaces, M. R. (2018, September). Integrating the common variability language with multilanguage annotations for web engineering. In Proceeedings of the 22nd International Conference on Systems and Software Product Line, pp. 196-207, ACM.

IEEE. (1990). Standard Glossary of Software Engineering Terminology. IEEE Standard 610.12-1990.

ISO 9241-11. (1998). Ergonomic requirements for office work with visual display terminals (VDTs) - Part 11: Guidance on usability.

ISO/IEC 25062. (2006). Software engineering—Software product Quality Requirements and Evaluation (SQuaRE)—Common Industry Format (CIF) for usability test reports.

Jordan, H. R., Russell, S. E., O'Hare, G. M., & Collier, R. W. (2012) Reuse by Inheritance in Agent Programming Languages. In: Intelligent Distributed Computing V, volume 382 of Studies in Computational Intelligence, pp. 279-289, Springer Berlin Heidelberg.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E. & Peterson, A. S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon Software Engineering Institute.

Kästner, C., Apel, S., & Kuhlemann, M. (2008). Granularity in software product lines. In: 30th Int. Conf. on Software Engineering (ICSE), pp. 311-320.

Kästner, C., Apel, S., & Ostermann, K. The road to feature modularity? (2011). In Proc. of the 15th Int. Software Product Line Conference, vol. 2, pp. 5, ACM.

Kästner, C., & Apel, S. (2008, October). Integrating compositional and annotative approaches for product line engineering. In Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering, pp. 35-40.

Kim, S. D., Min, H. G., & Rhew, S. Y. (2005, May). Variability design and customization mechanisms for COTS components. In International Conference on Computational Science and Its Applications, pp. 57-66, Springer, Berlin, Heidelberg.

Kitchenham, B., & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering.

Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., & Damiani, F. (2014, September). DeltaJ 1.5: delta-oriented programming for Java 1.5. In Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools, pp. 63-74, ACM.

Lago, P., Niemela, E., & Van Vliet, H. (2004, March). Tool support for traceable product evolution. In Eighth European Conference on Software Maintenance and Reengineering, 2004, pp. 261-269, IEEE.

Laguna, M. A., & Crespo, Y. (2013). A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. Science of Computer Programming, vol. 78(8), pp. 1010-1034.

Lahiani, N., & Bennouar, D. (2017). A DSL-based Approach to Product Derivation for Software Product Line. Acta Informatica Pragensia, vol. 5(2), pp. 138-143.

Le, D. M., Lee, H., Kang, K. C., & Keun, L. (2013). Validating Consistency between a Feature Model and Its Implementation. In: ICSR, pp. 1-16.

Lecoutre, C. (2009). Constraint Networks, Wiley-IEEE Press.

Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. Journal of Systems and Software, vol. 101, pp. 193-220.

Likert, R. (1932). A technique for the measurement of attitudes. Archives of psychology.

Lund, A. M. (2001). Measuring usability with the use questionnaire. Usability interface, vol. 8(2), pp. 3-6.

Marimuthu, C., & Chandrasekaran, K. (2017, September). Systematic Studies in Software Product Lines: A Tertiary Study. In Proceedings of the 21st International Systems and Software Product Line Conference, pp. 143-152, ACM.

Mayer, P., & Bauer, A. (2015, April). An empirical analysis of the utilization of multiple programming languages in open source projects. In Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, pp. 4, ACM.

Mazo, R. (2014). Avantages et limites des modèles de caractéristiques dans la modélisation des exigences de variabilité. Journal "Génie Logiciel", no. 111, Paris-France, pp. 42-48.

Mazo, R., Martínez, J. C., López, J. I. (2018). Proceso de configuración como un proceso de ingeniería de requisitos. In Guía para la adopción industrial de líneas de productos de software, Editorial Eafit, ISBN 978-958-720-506-0, pp. 397-431, Medellín-Colombia.

Mazo, R., Muñoz-Fernández, J. C., Rincón, L., Salinesi, C. & Tamura, G. (2015). VariaMos: an extensible tool for engineering (dynamic) product lines. In: Proc. of the 19th Int. Conf. on Software Product Line, pp. 374-379, ACM.

Mens, T. (2004). A survey of software refactoring. IEEE Transactions on software engineering, vol. 2, pp. 126-139.

Metzger, A., & Pohl, K. (2014, May). Software product line engineering and variability management: achievements and challenges. In Proceedings of the on Future of Software Engineering, pp. 70-84, ACM.

Montalvillo, L., Díaz, O., & Azanza, M. (2017, September). Visualizing product customization efforts for spotting SPL reuse opportunities. In Proceedings of the 21st International Systems and Software Product Line Conference, pp. 73-80, ACM.

Neto, P. A. D. M. S., do Carmo Machado, I., McGregor, J. D., De Almeida, E. S., & de Lemos Meira, S. R. (2011). A systematic mapping study of software product lines testing. Information and Software Technology, vol. 53(5), pp. 407-423.

Nielsen, J. (1993). Usability Engineering, Academic Press, Boston, MA.

Parr, T. (2013). The definitive ANTLR 4 reference. Pragmatic Bookshelf.

Peffers, K., Tuunanen T., Chatterjee M.A, & Rothenberger S. A. (2007). Design science research methodology for information systems research. Journal of Management Information Systems, vol. 24(3), pp. 45-77.

Pereira, J. A., Constantino, K., & Figueiredo, E. (2015, January). A systematic literature review of software product line management tools. In International Conference on Software Reuse, pp. 73-89, Springer, Cham.

Petersen, K., Feldt, R., Mujtaba, S., & Mattsson, M. (2008, June). Systematic mapping studies in software engineering. In Ease, vol. 8, pp. 68-77.

Pleuss, A., Hauptmann, B., Dhungana, D., & Botterweck, G. (2012, June). User interface engineering for software product lines: the dilemma between automation and usability. In Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems, pp. 25-34, ACM.

Prehofer, C. (1997). Feature-Oriented Programming: A Fresh Look at Objects. In: Proc. Europ. Conf. Object-Oriented Programming, pp. 419-443.

Rabiser, R., Grünbacher, P., & Lehofer, M. (2012, September). A qualitative study on user guidance capabilities in product configuration tools. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 110-119, ACM.

Rabiser, R., O'Leary, P., & Richardson, I. (2011). Key activities for product derivation in software product lines. Journal of Systems and Software, vol. 84(2), pp. 285-300.

Rabiser, R., Wolfinger, R., & Grunbacher, P. (2009, January). Three-level customization of software products using a product line approach. In 42nd Hawaii International Conference on System Sciences, pp. 1-10, IEEE.

Salvaneschi, G., Ghezzi, C., & Pradella, M. (2012). Context-oriented programming: A software engineering perspective. J. of Systems and Software, vol. 85(8), pp. 1801-1817.

Sawant, A. A., Bari, P. H., & Chawan, P. M. (2012) Software testing techniques and strategies. International Journal of Engineering Research and Applications (IJERA), vol. 2(3), pp. 980-986.

Sawyer, P., Mazo, R., Diaz, D., Salinesi, C., & Hughes, D. (2012). Using constraint programming to manage configurations in self-adaptive systems. Computer, vol. 45(10), pp. 56-63.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., & Tanzarella, N. (2010). Delta-oriented programming of software product lines. In: SPLC. LNCS, vol. 6287, pp. 77-91.

Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., & Saake, G. (2008, December). Measuring non-functional properties in software product line for product derivation. In 2008 15th Asia-Pacific Software Engineering Conference, pp. 187-194, IEEE.

Soltani, S., Asadi, M., Gašević, D., Hatala, M., & Bagheri, E. (2012, September). Automated planning for feature model configuration based on functional and non-functional requirements. In Proceedings of the 16th International Software Product Line Conference, pp. 56-65, ACM.

Souza, L. O., O'Leary, P., de Almeida, E. S., & de Lemos Meira, S. R. (2015). Product derivation in practice. Information and Software Technology, vol. 58, pp. 319-337.

Teruel, M. A., Navarro, E., López-Jaquero, V., Montero, F., & González, P. (2014). A CSCW requirements engineering CASE tool: development and usability evaluation. Information and Software Technology, vol. 56(8), pp. 922-949.

Tizzei, L. P., Rubira, C. M., & Lee, J. (2012). An aspect-based feature model for architecting component product lines. In: SEAA, pp. 85-92, IEEE.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., & Leich, T. (2014). FeatureIDE: An extensible framework for feature-oriented software development. Science of Computer Programming, vol. 79, pp. 70-85.

Van Ommering, R., & Bosch, J. (2002, August). Widening the scope of software product lines—from variation to composition. In International Conference on Software Product Lines, pp. 328-347, Springer, Berlin, Heidelberg.

Yu, Y., do Prado Leite, J. C. S., Lapouchnian, A., & Mylopoulos, J. (2008, March). Configuring features with stakeholder goals. In Proceedings of the 2008 ACM symposium on Applied computing, pp. 645-649, ACM.

Walkingshaw, E., & Erwig, M. (2012, September). A calculus for modeling and implementing variation. In ACM SIGPLAN Notices, vol. 48(3), pp. 132-140, ACM.

Wang, A. J. A., & Qian, K. (2005). Component-oriented programming. John Wiley & Sons.

Wileden, J. C., & Kaplan, A. (1999, May). Software interoperability: Principles and practice. In Proceedings of the 21st international conference on Software engineering, pp. 675-676, ACM.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2000). Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers.

Wohlin, C., Runeson, P., Neto, P. A. D. M. S., Engström, E., do Carmo Machado, I., & De Almeida, E. S. (2013). On the reliability of mapping studies in software engineering. Journal of Systems and Software, vol. 86(10), pp. 2594-2610.

Zheng, Y., & Cu, C. (2016, May). Towards implementing product line architecture. In IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities, pp. 5-10, IEEE.