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

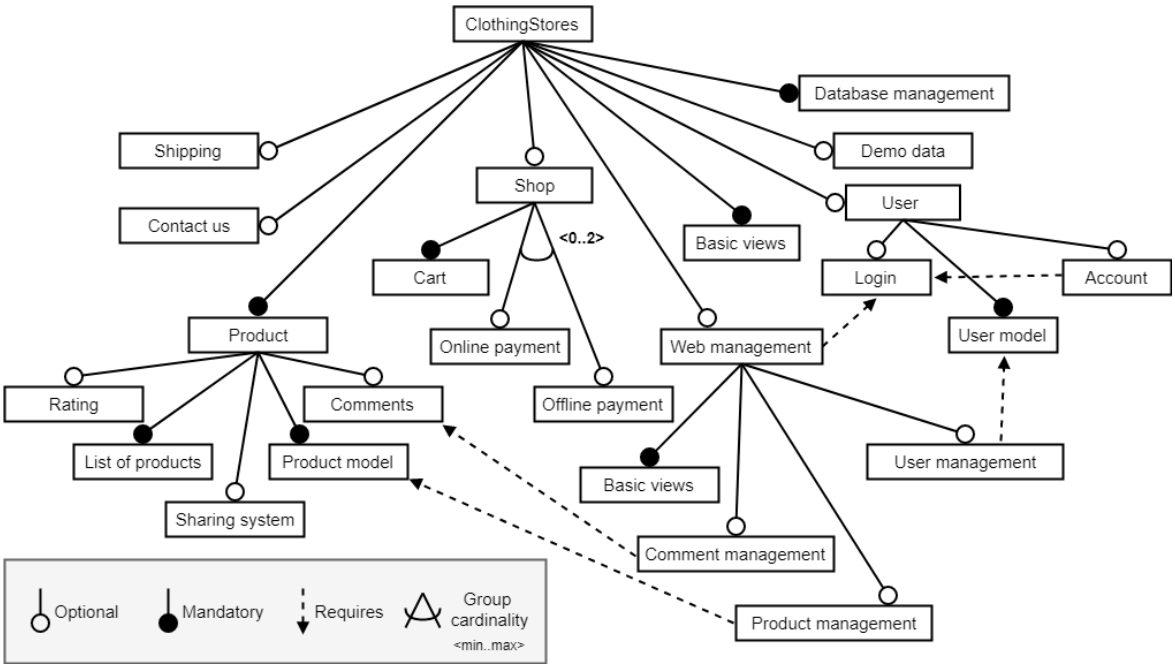


Table 2-1: ClothingStores list of features

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 that only contain the mandatory requirements, to very complete ones that contain all mandatory and all optional 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.

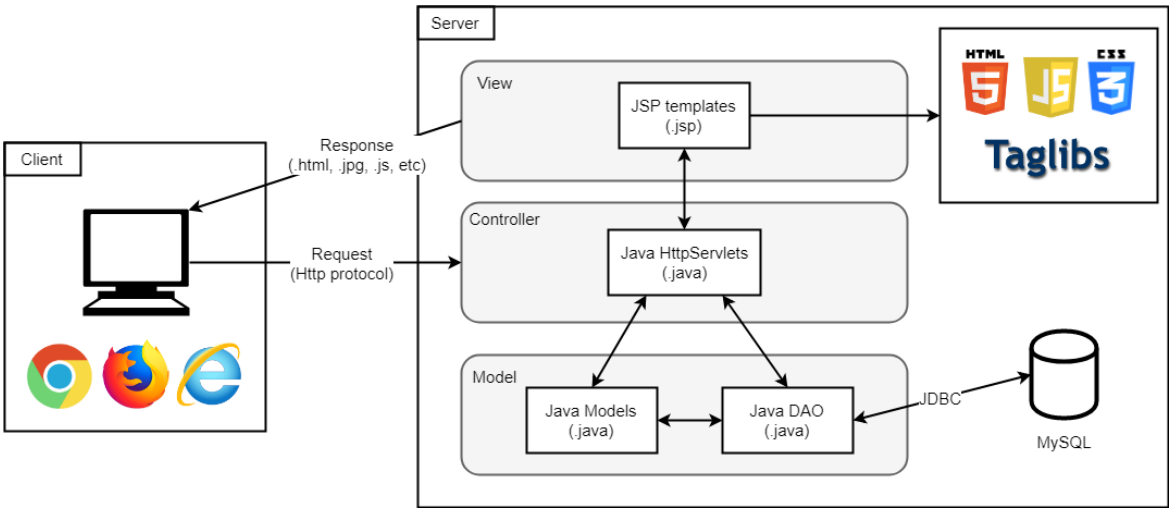
Keeping in mind the multiple challenges that the development of an SPL involves, which includes the use of several software languages, we decided to define a software architecture that used different software languages that communicated between each other. For this example, we decided:

- 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

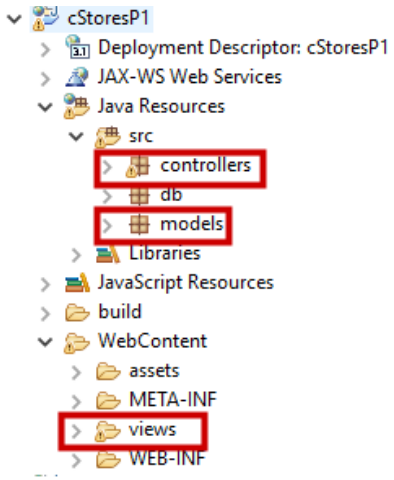


The server is divided into three layers:

- **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



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

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

concerns, fine-grained extensions, coarse-grained extensions, product customization, and 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 **¡Error! No se encuentra el origen de la referencia.** 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

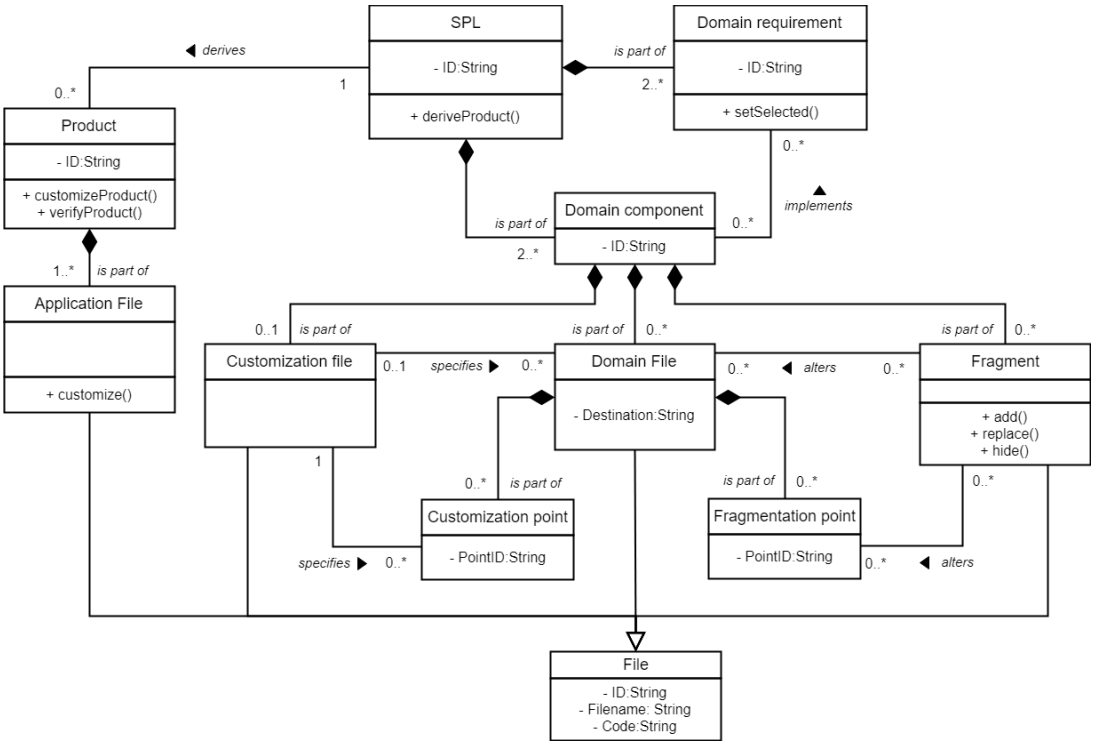
FragOP is a framework used to design, implement and reuse domain components in the context of an SPL. This framework is a mix of compositional and annotative approaches, which is based on the definition of six fundamental elements: (i) domain components, (ii) domain files, (iii) fragmentation points, (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)

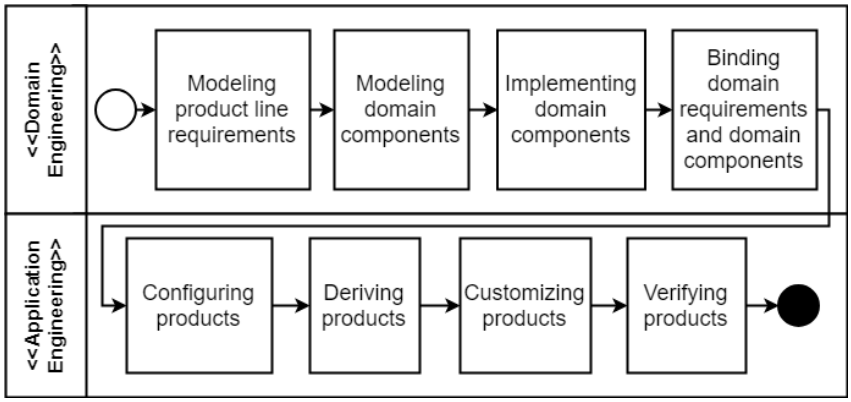


Each FragOP element is described in detail in Chapters 4 and 5.

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)



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, FragOP executes the fragments and modifies the product's file code, 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 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 SPL. FragOP is defined as a mix of compositional and annotative approaches and is based

[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/

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:

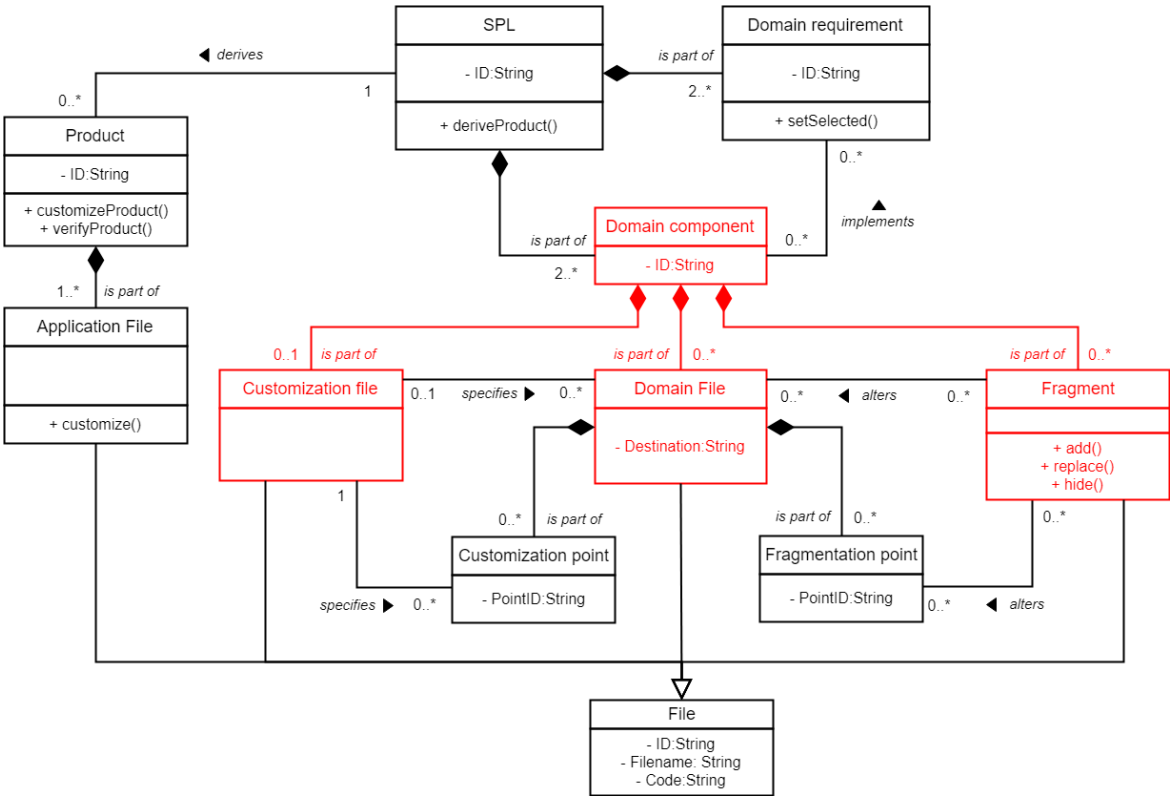
- Feature-oriented programming with AHEAD (Java 1.4), FeatureC++ (C++), and FeatureHouse (C, Java 1.5, JML, Haskell, XML, JavaCC).
- Aspect-oriented programming with AspectJ (Java).
- Delta-oriented modeling and programming with DeltaEcore and DeltaJ (Java).
- Annotation-based implementation with CIDE (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}`. Essentially, a domain component is a folder that stores 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

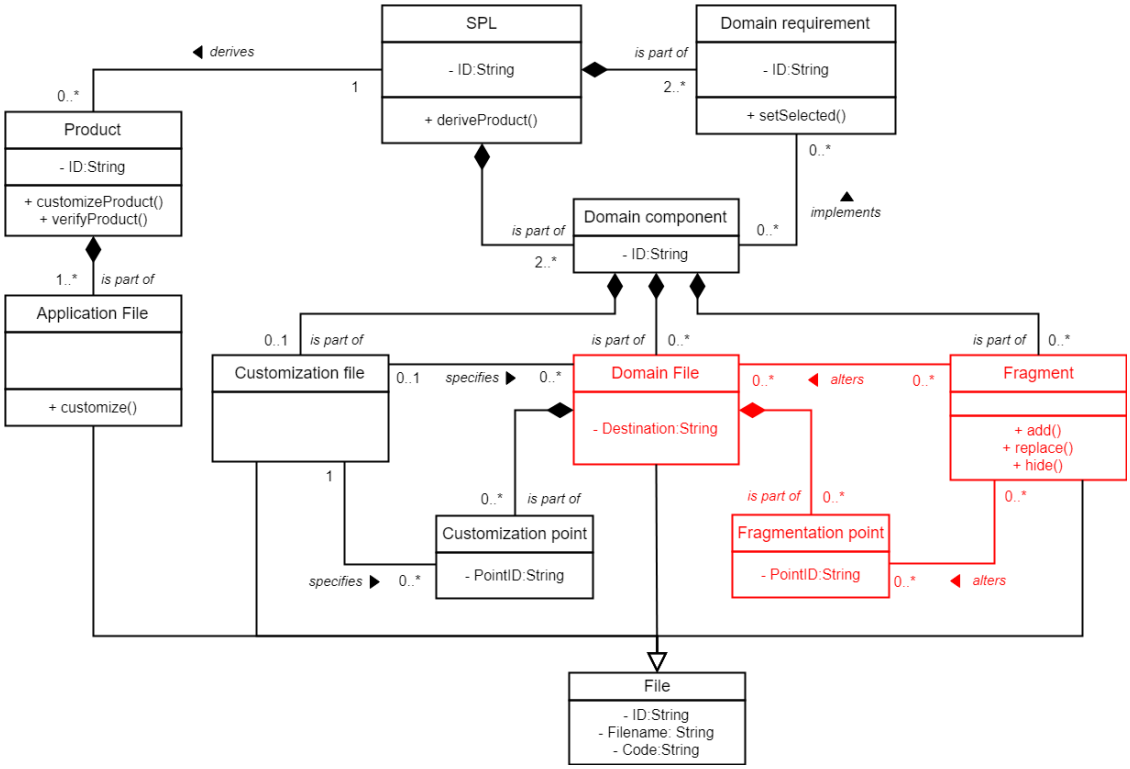
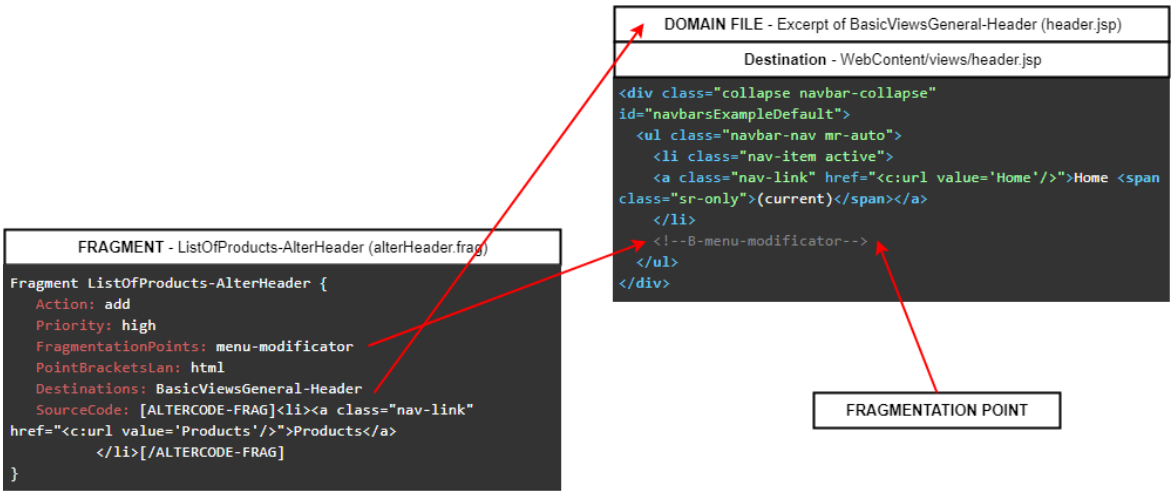


Figure 4-3: An example of the connection between a domain file, a fragment, and a fragmentation point



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).

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.

Listing 4-2: Fragmentation point shape

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

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]`.

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-->
```

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>
```

```
 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*>. Specifies the fragment priority (`high`, `medium` or `low`). Fragments with `high` priority are assembled before fragments with `medium` or `low` priority. 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 hold in the middle ground, with the fragment priority 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

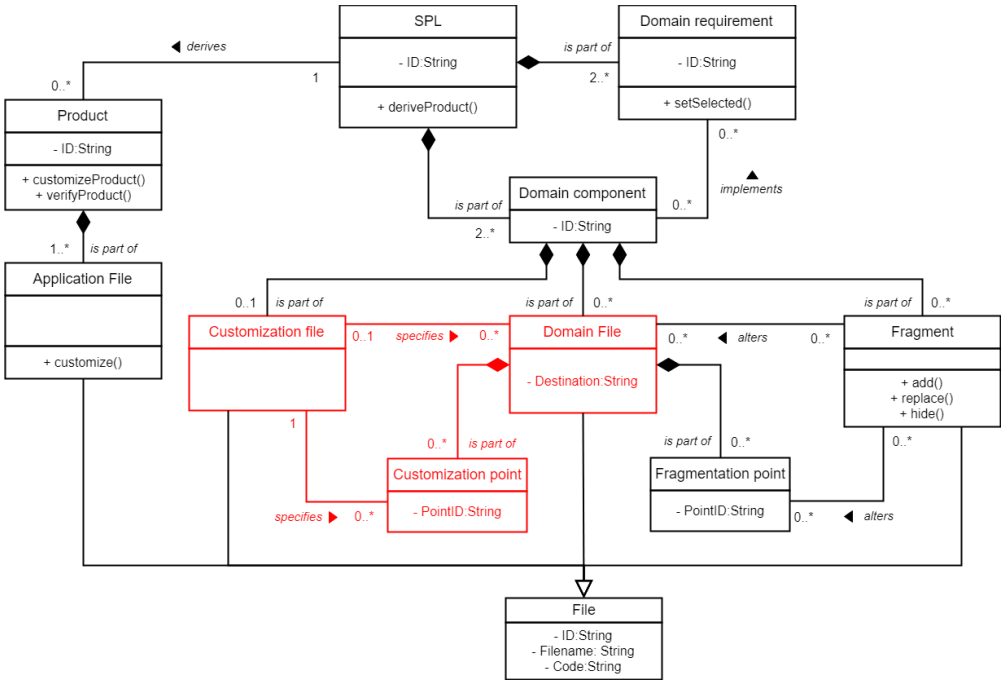
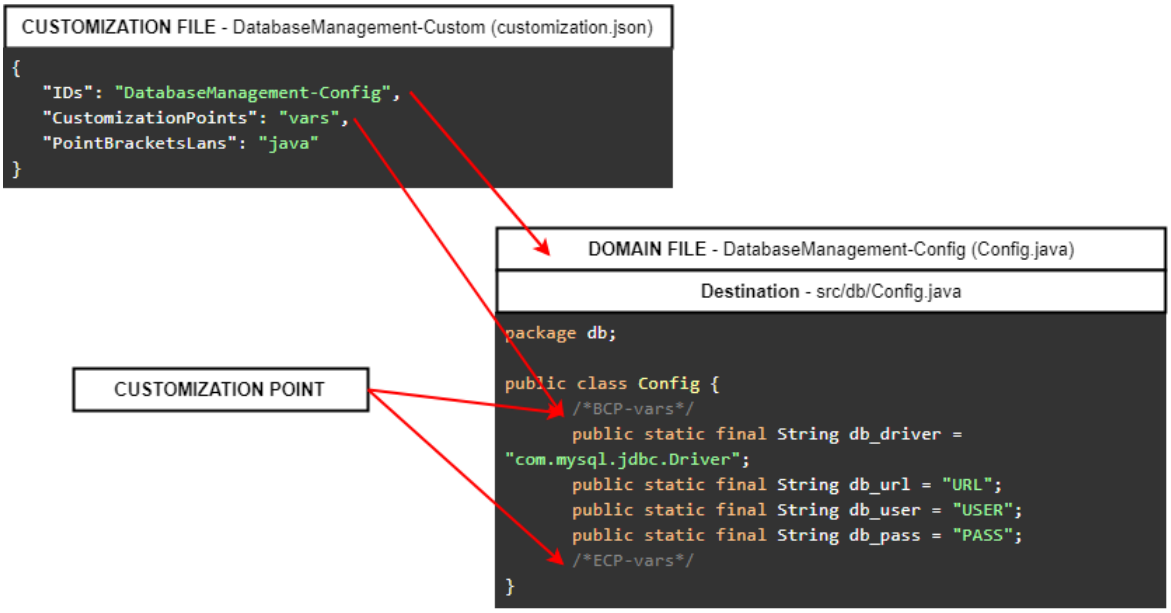


Figure 4-5: An example of the connection between a domain file, a customization file, and a customization point



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.

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).

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
```

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", "..."],
   "PointBracketsLans": ["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.

PointBracketsLans: <*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",
   "PointBracketsLans": "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 component 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:

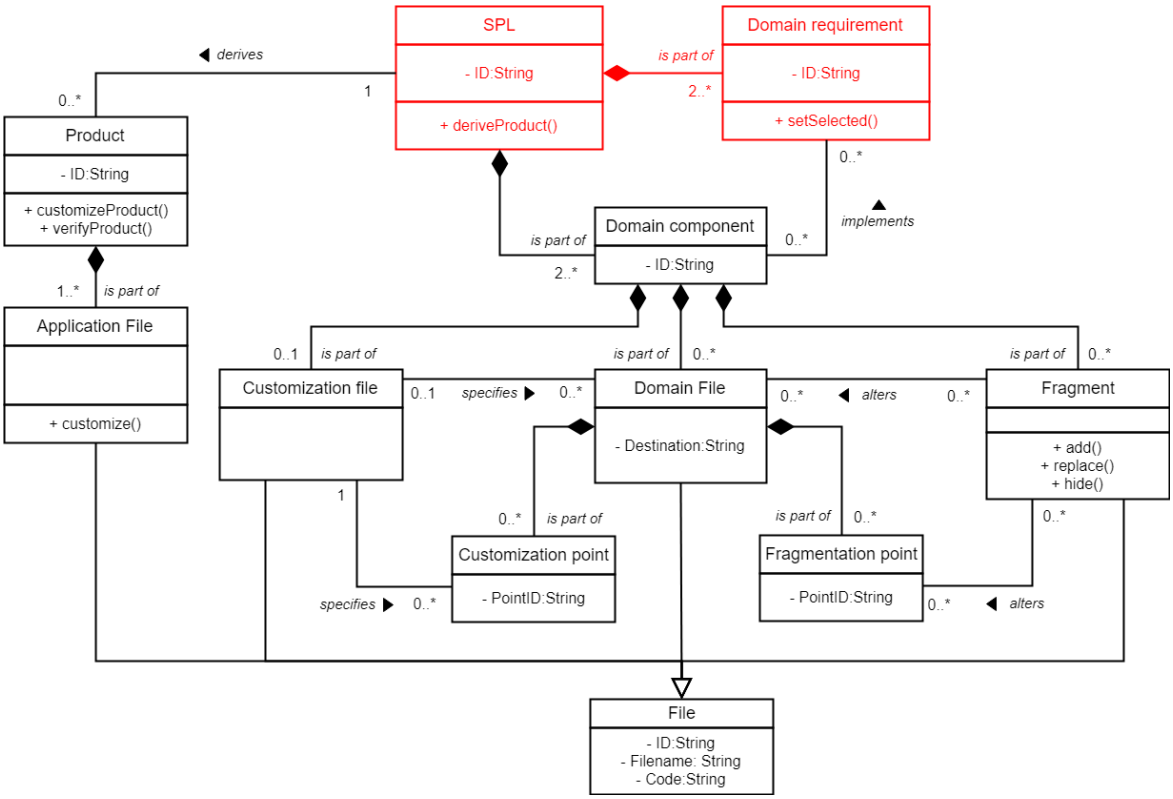
- **Requirements elicitation** consists of finding and identifying the relevant PL requirements. First, it is important to identify the different stakeholders (such as final users, employees, software developers, and executives) that could be interested in a product (relevant to the PL domain). Second, 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.

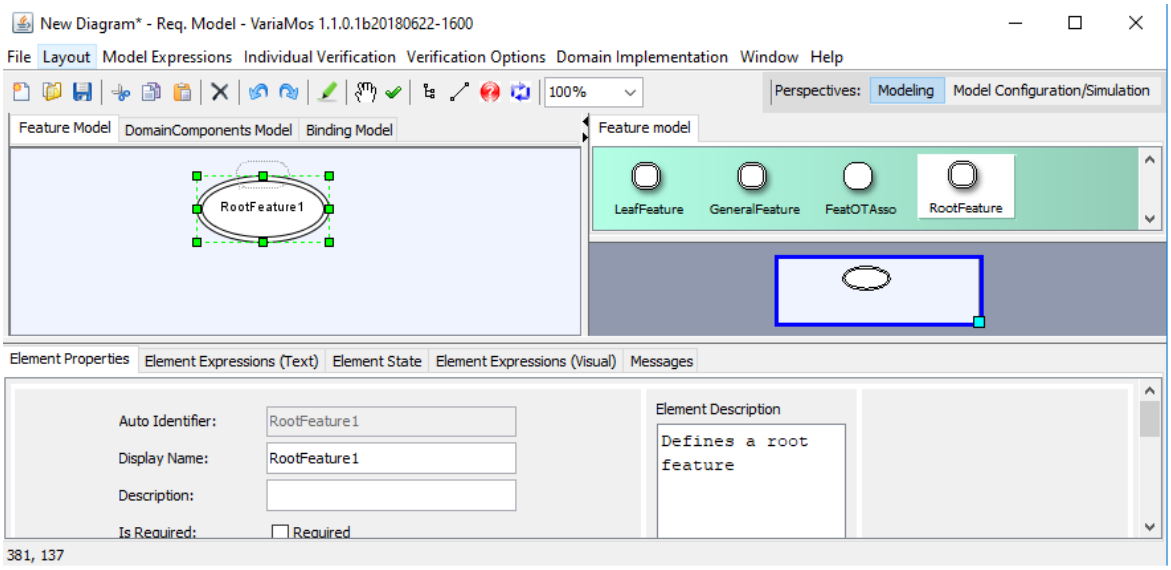
Figure 5-1: FragOP metamodel highlighting the concepts of SPL and domain requirement



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



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:

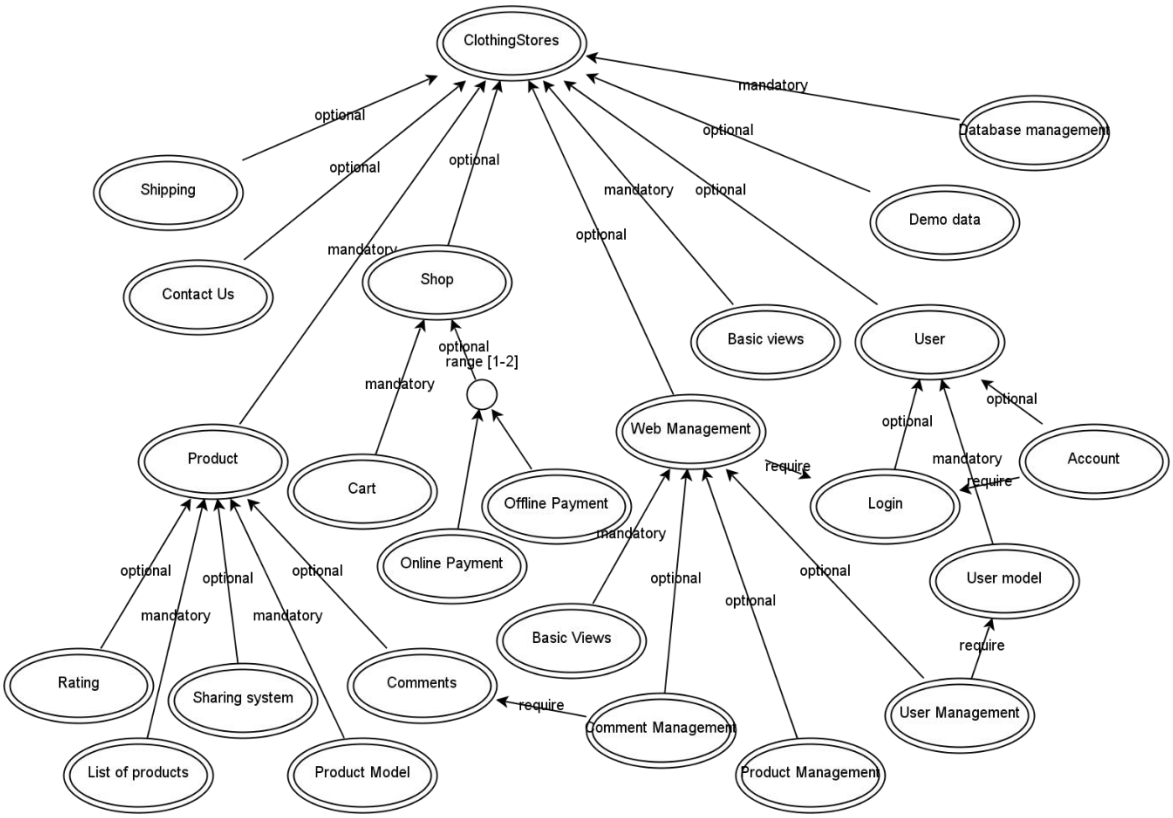
- **FeatureIDE with AHEAD (FOP) or FeatureHouse (FOP) or AspectJ (AOP).** After the creation of a FeatureIDE project, a model.xml file is automatically created. When the model.xml file is opened, a graphical editor is presented. There the SPL developer can edit the SPL feature model.

- **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)

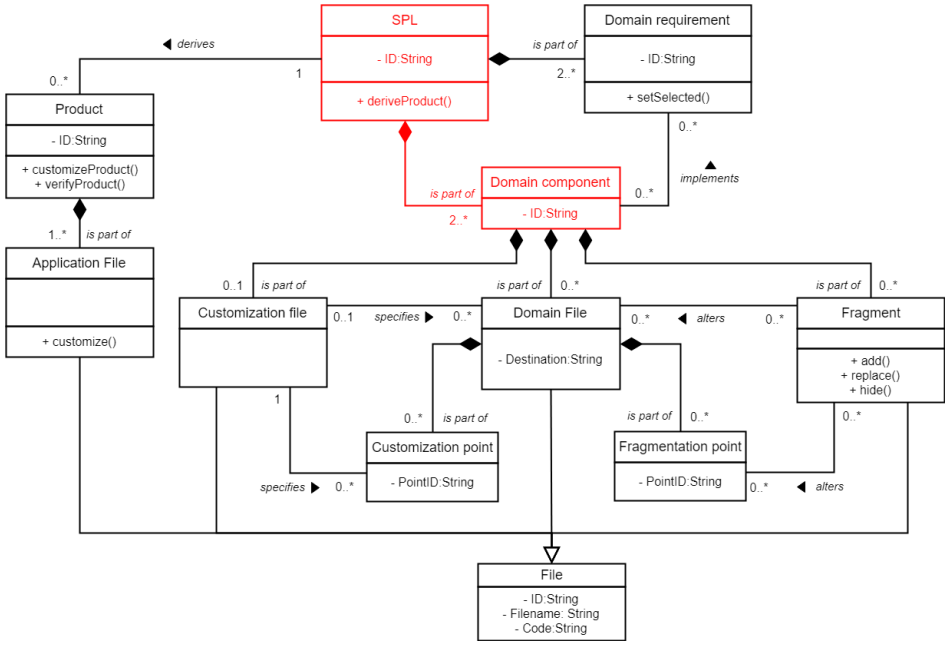


5.2 Modeling domain components

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

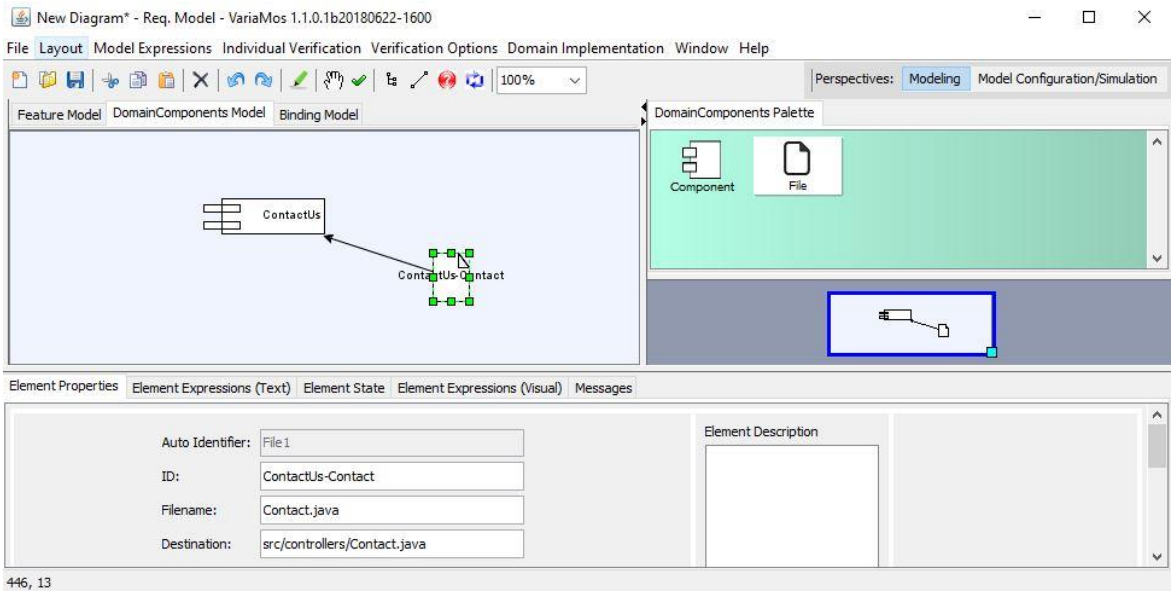
		Account-Img	user.png	WebContent/assets/img/user.png
		Account-AlterHeader	alterHeader.frag	
C13	Login	Login-Login	Login.java	src/controllers/Login.java
		Login-LoginForm	login_form.jsp	WebContent/views/login_form.jsp
		Login-AlterAccount	alterAccount.frag	
		Login-AlterAdmin	alterAdmin.frag	
		Login-AlterHeader	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**¡Error! No se encuentra el origen de la referencia.**), (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.

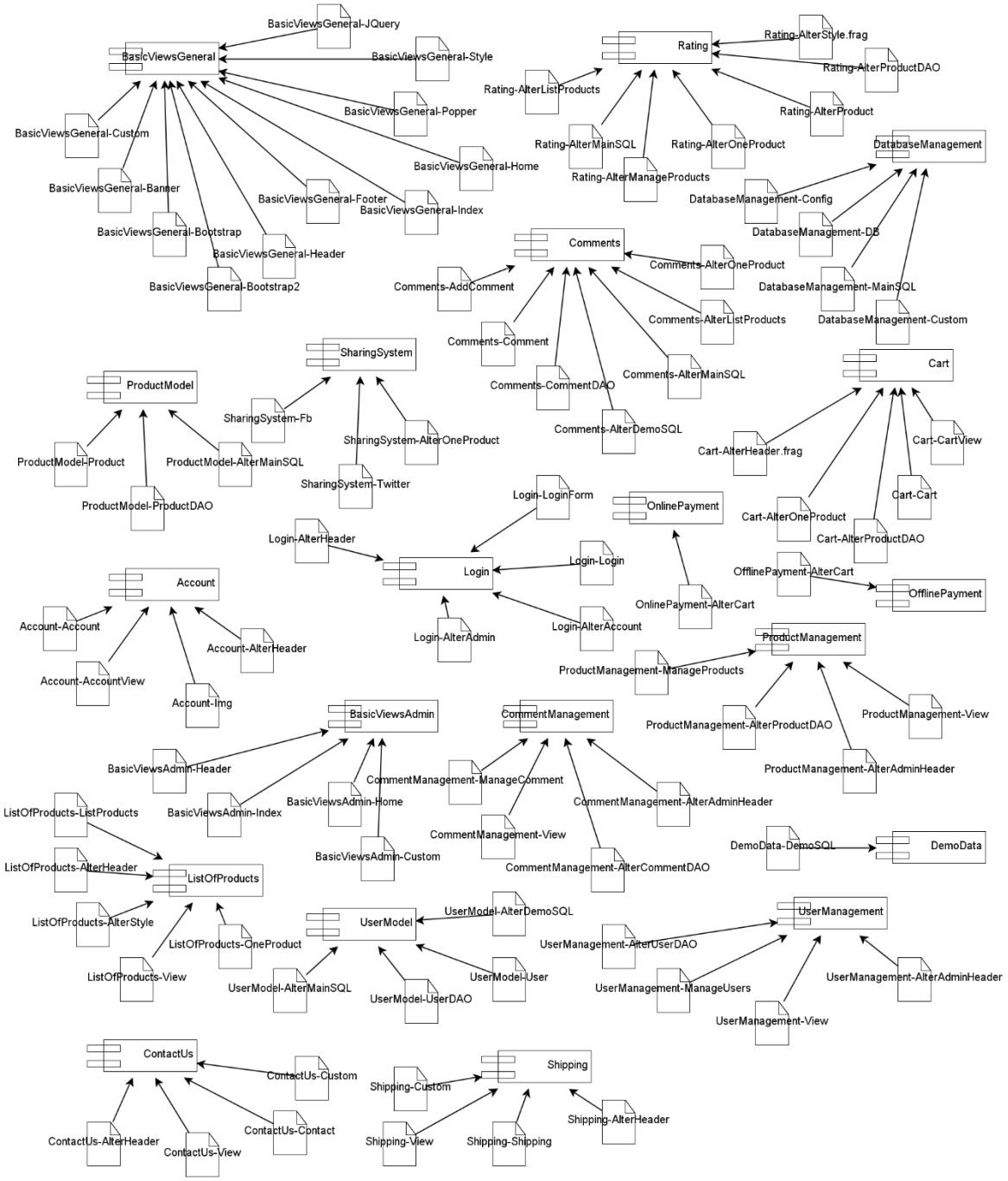
After developing the list of domain components and their files, the SPL developer should use VariaMos to create the domain component model. This activity requires the SPL developer to navigate to the “Domain component model” view (see Figure 5-5). The process consists of graphically representing the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

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



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

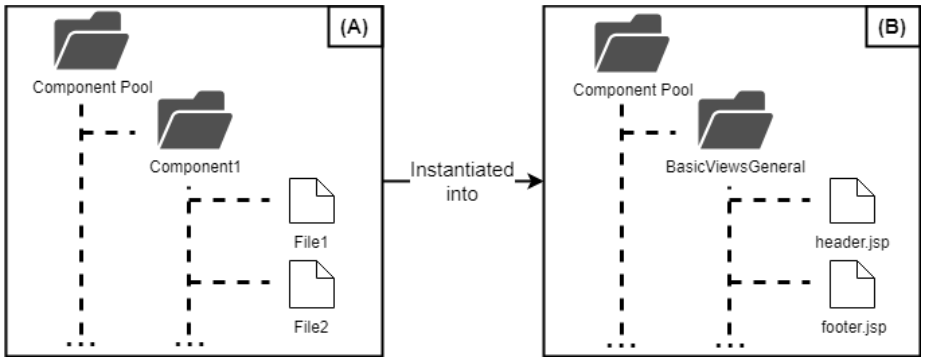
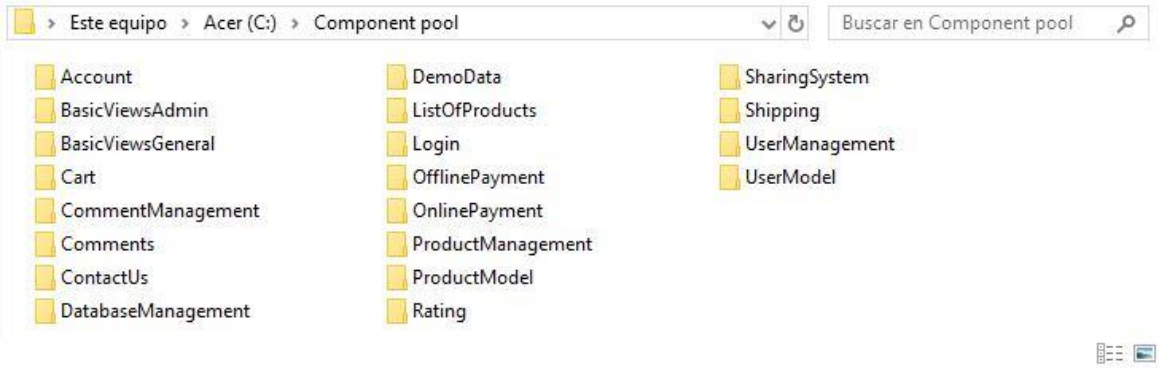


Figure 5-8 shows the ClothingStores component pool folder structure based on the 20 domain components defined in Table 5-1.

Figure 5-8: Clothing stores component pool folder

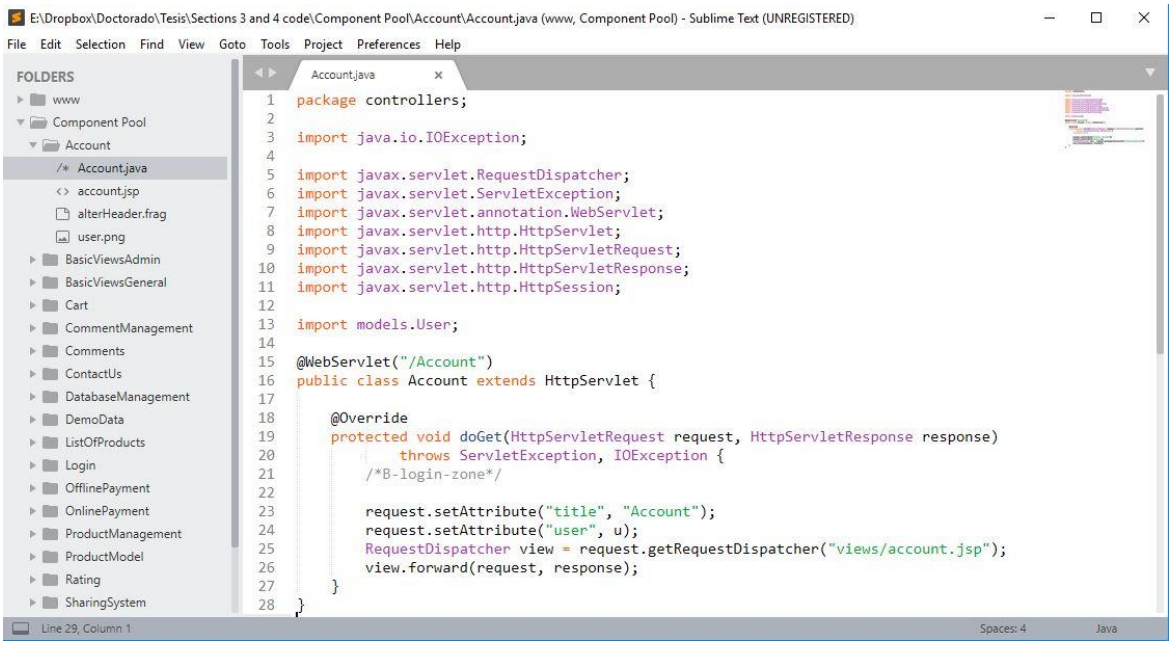


After the definition of the component pool, we recommend selecting a preferred IDE to develop the component file source code such as Sublime[3], IntelliJ[4], NetBeans[5], and Eclipse[6], among others. Figure 5-9 shows the example of Sublime being used to create the

[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



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).
- **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).

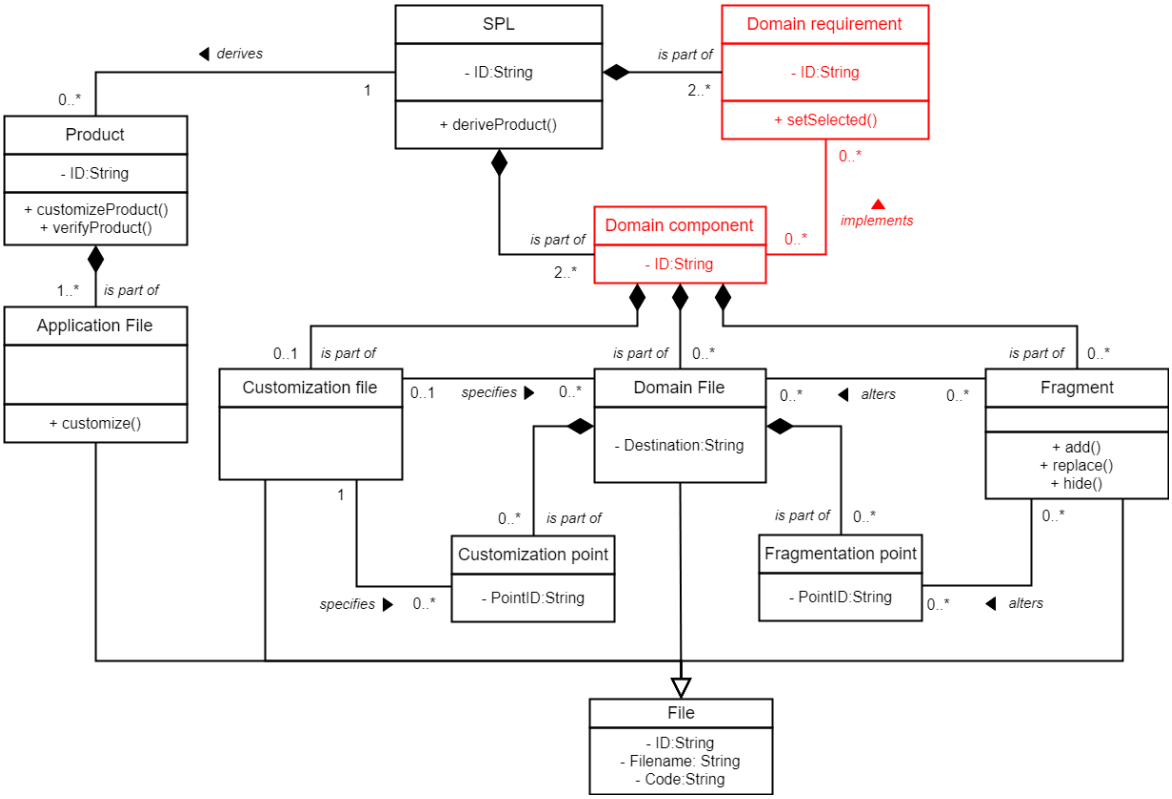
The reality is that most current approaches do not use graphical binding models, using instead a direct connection between the features and the components. For example:

- 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

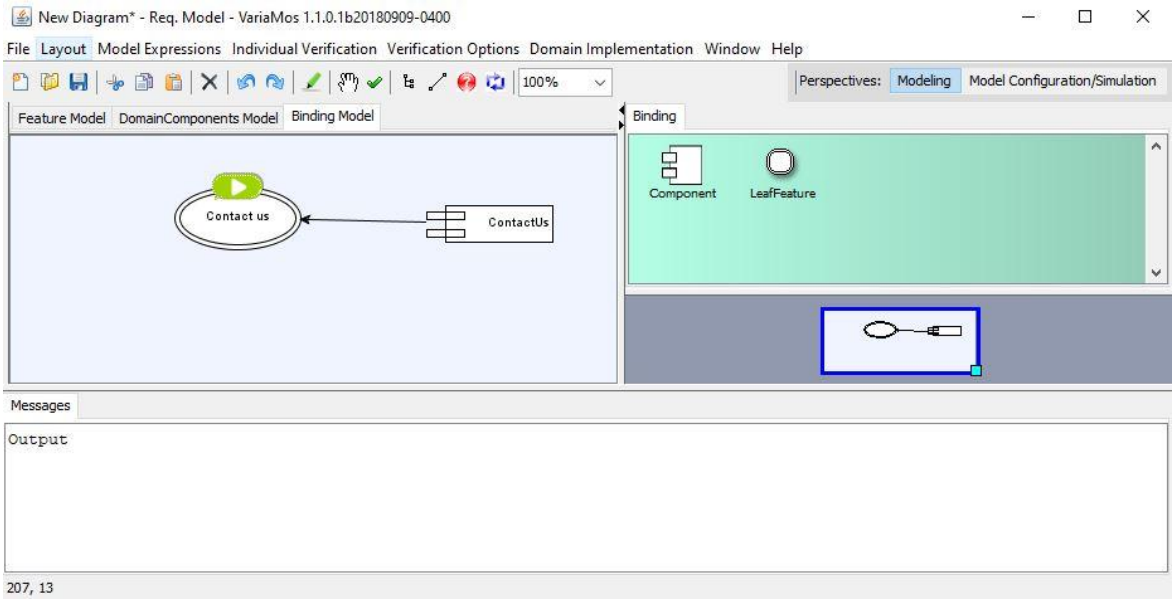


5.4.1 VariaMos support

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

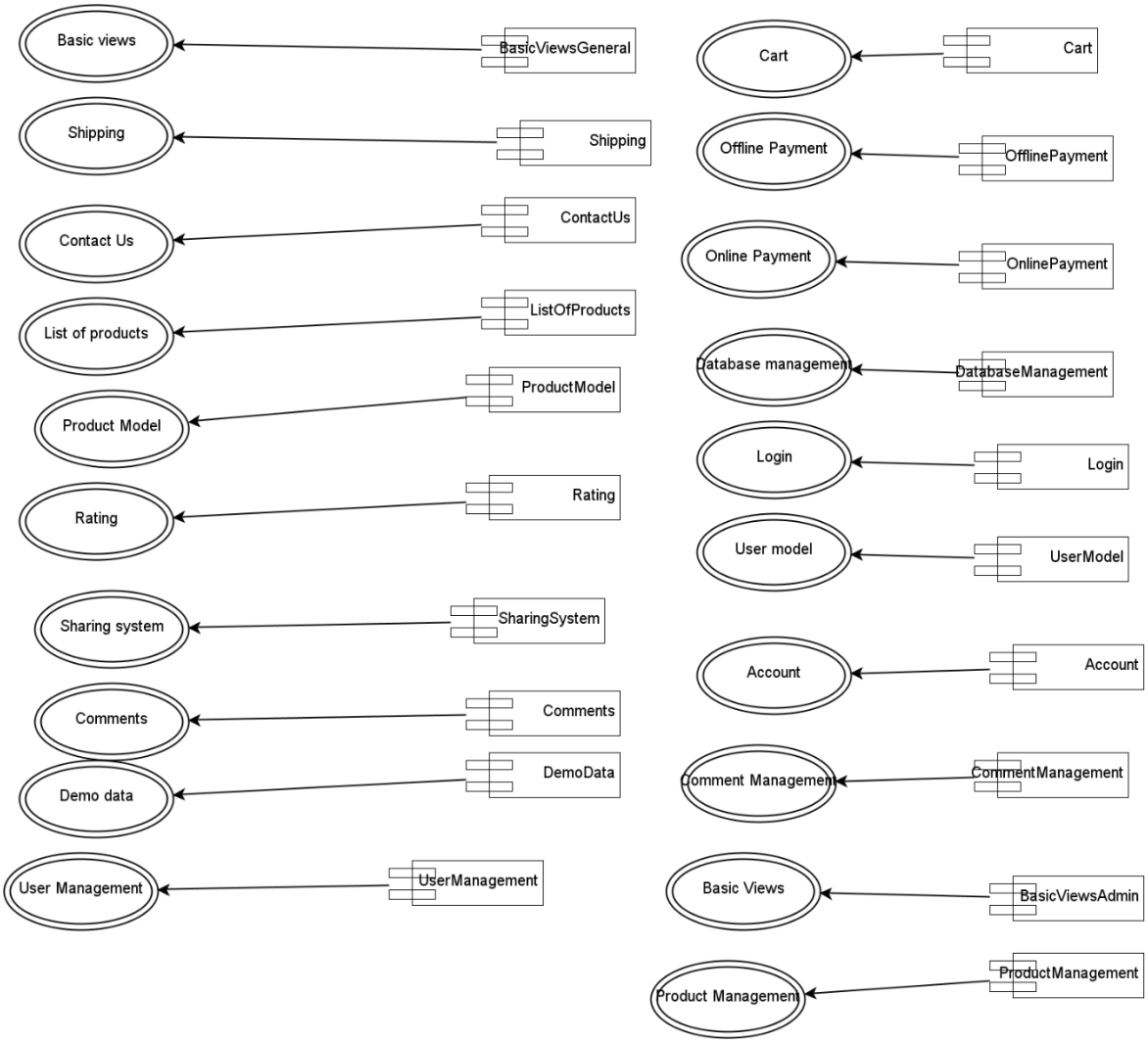


At present, VariaMos only allows a one-to-one binding relationship. Nevertheless, we plan to implement a constraint network (Lecoutre, 2009) to graphically represent more complex domain implementation relationships such as “Domain components C1 or C2, but not both, can be used to implement feature F”.

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).

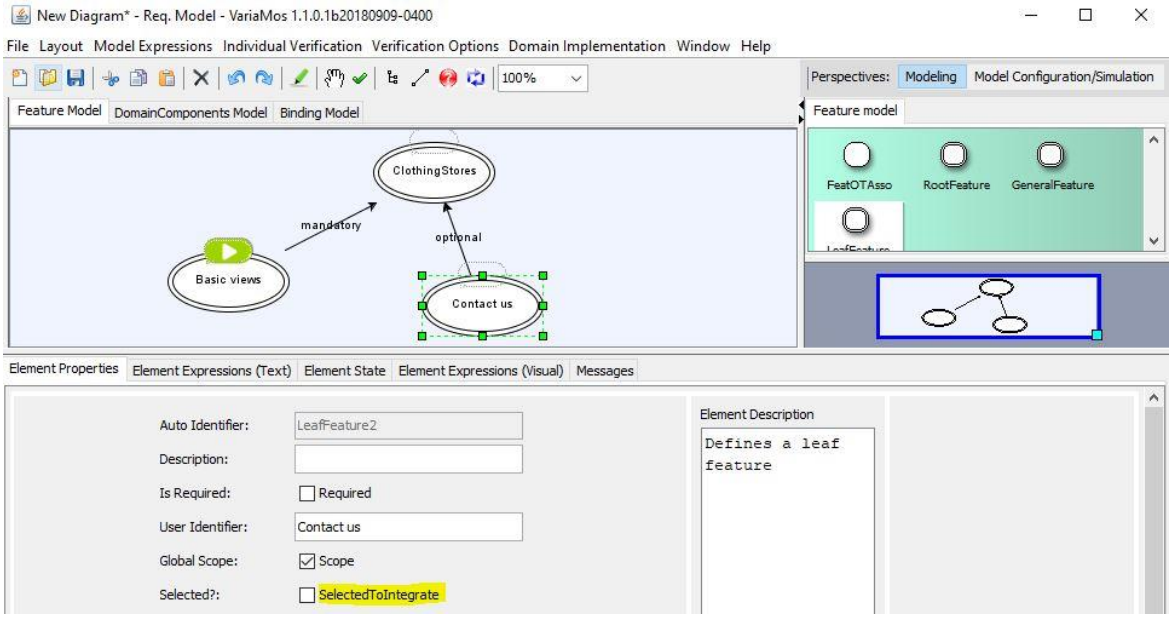
- **Selection approach** is an iterative process which consists of selecting the desired features that will be included in the final product. This approach starts with the selection of one desired feature `F1`, and the system looks for all possible configurations that include `F1`. The process continues 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



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

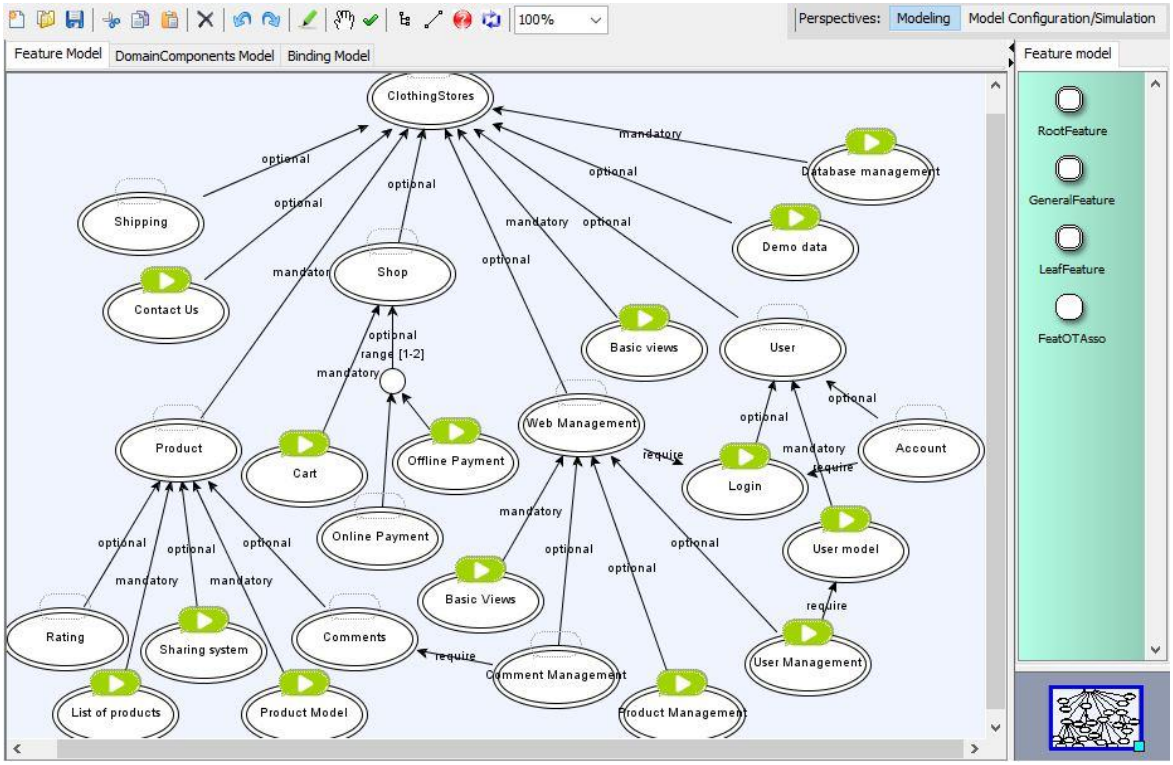
should open the configuration file and select the features that he/she wants to include in the current configuration.

- **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)

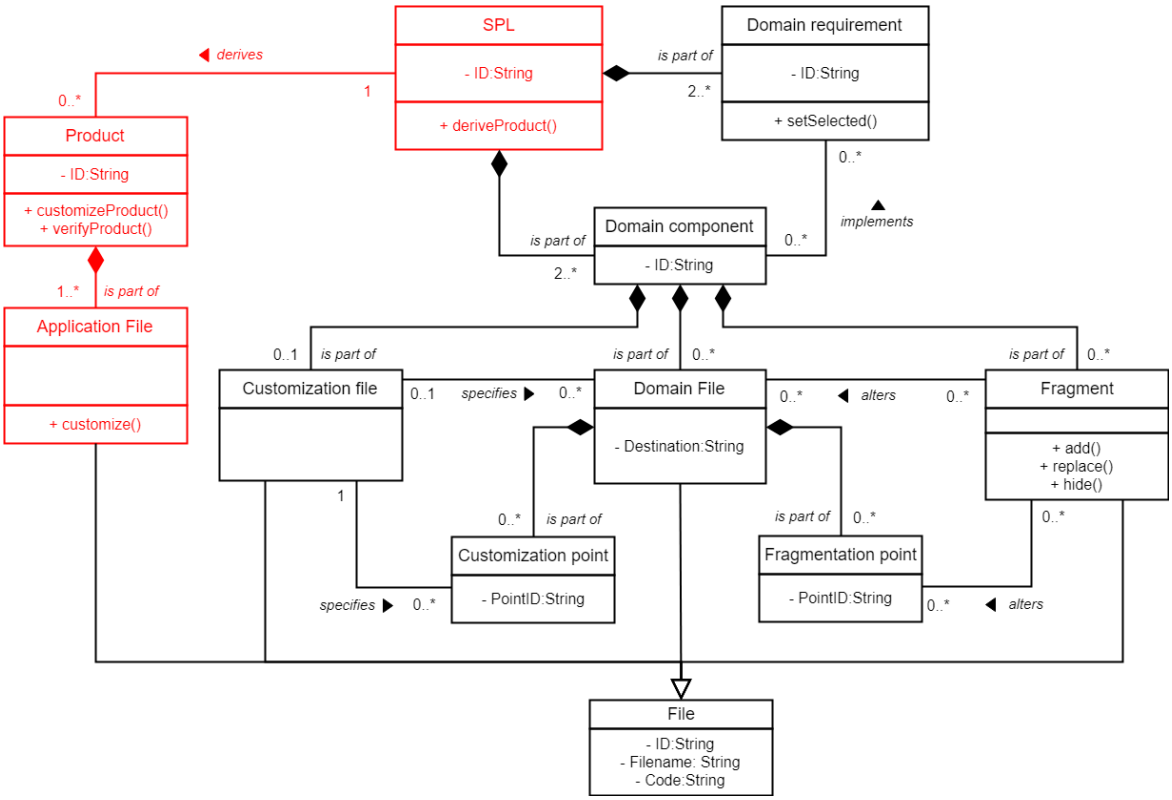


Based on the customer requirements, an SPL developer must deselect the `SelectedToIntegrate` option for each of the following features: (i) `shipping`, (ii) `rating`, (iii) `comments`, (iv) `comment management`, (v) `online payment`, and (vi) `account`. Figure 5-14 shows the final product configuration.

5.6 Deriving products

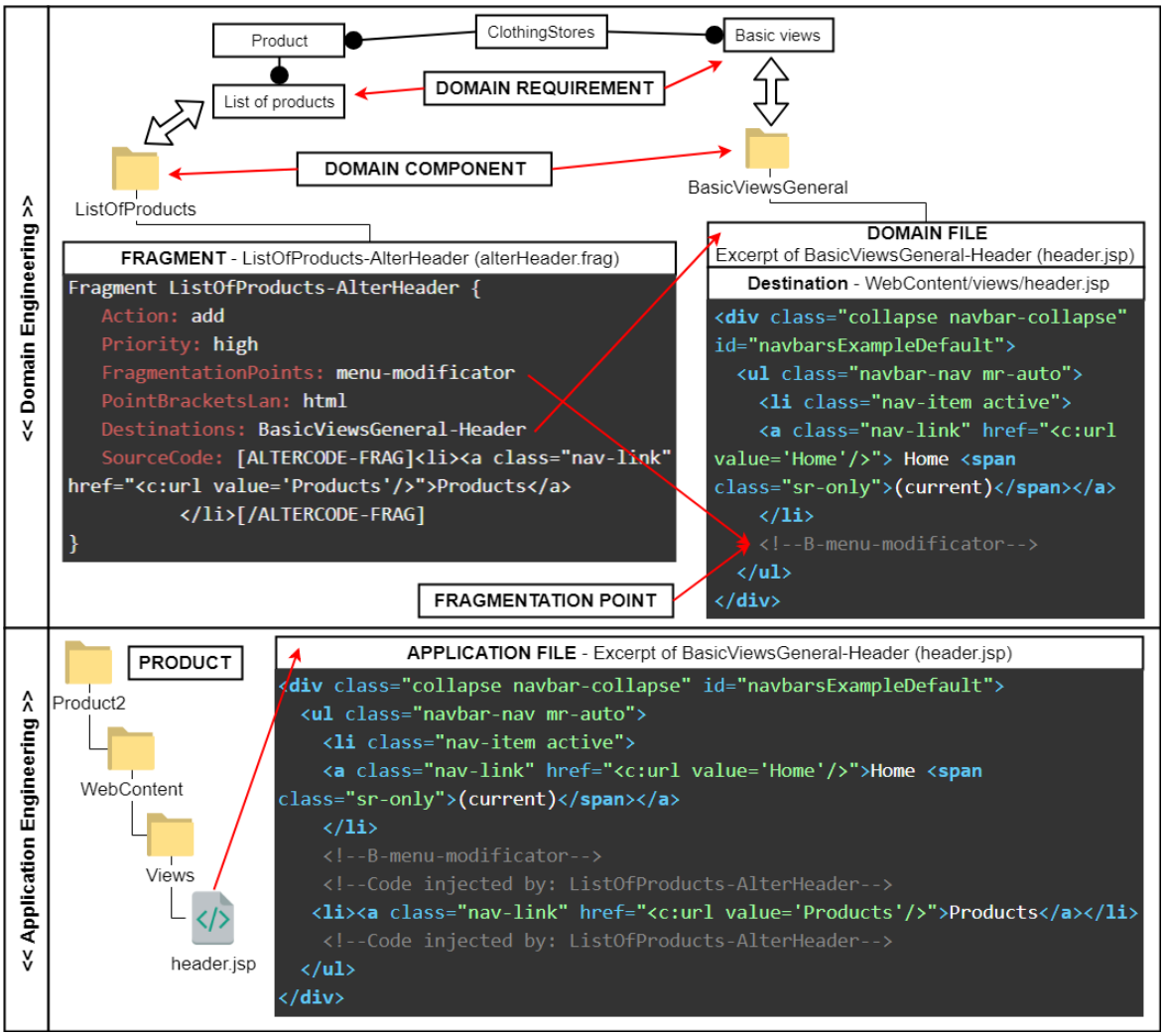
One of the key activities for an SPLE is the product derivation. Product derivation aims to create specific software products based on the assembling of the reusable domain components. Depending on the approach selected to build the domain components and the tool that supports the product derivation 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



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

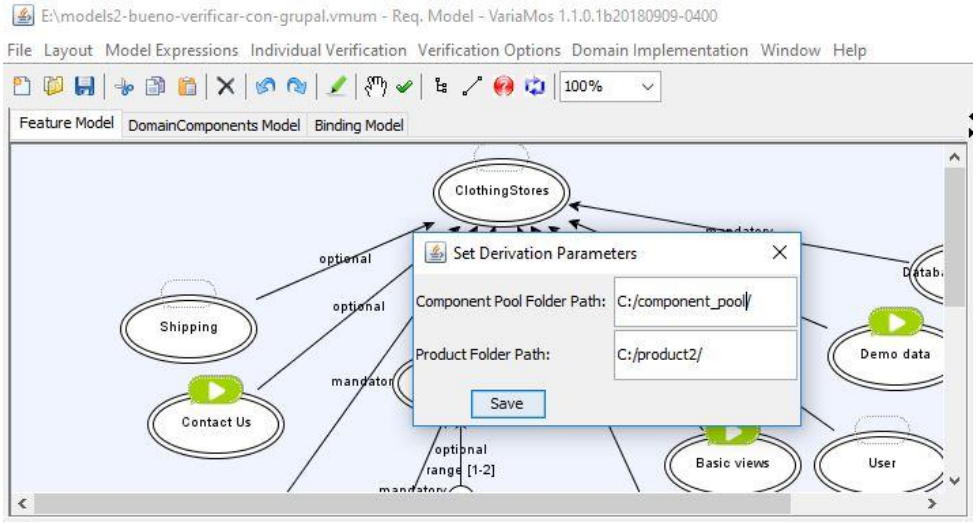
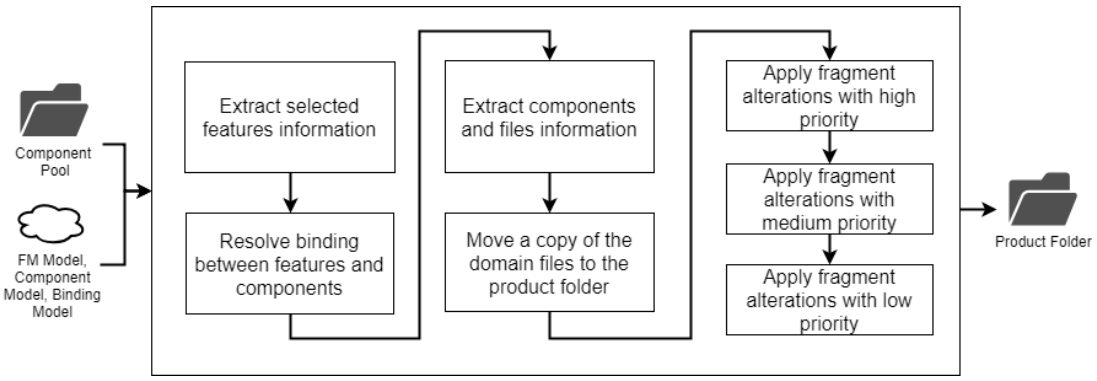


Figure 5-18: VariaMos product derivation activity



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.

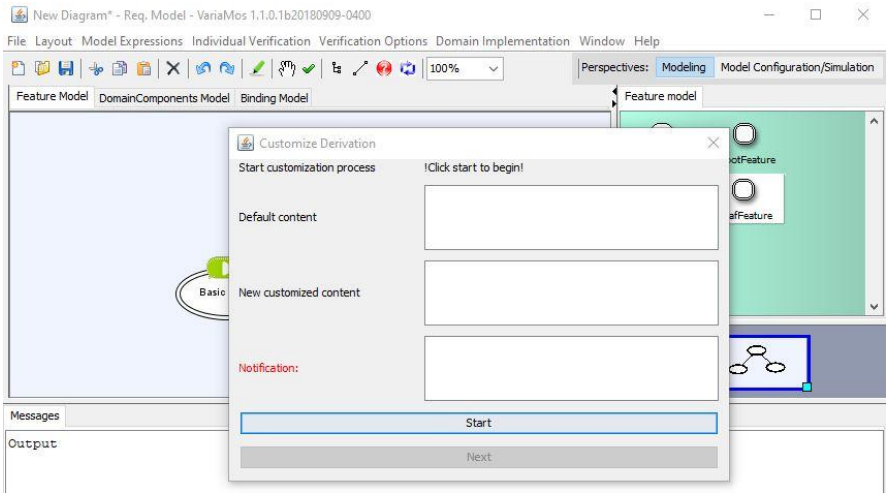
5.7.1 VariaMos support

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.

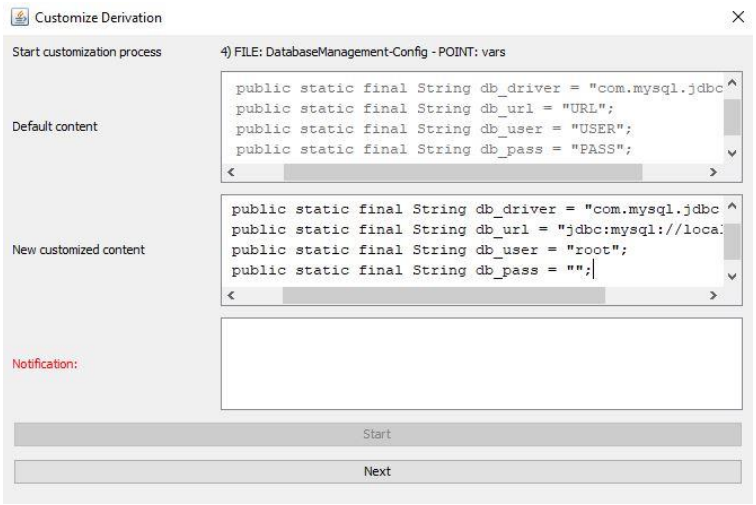
Figure 5-19: VariaMos product customization activity



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)



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 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 grammar, 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 verification activity work?

The product 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 verification support allows us to answer RQ3.

ClothingStores SPL

To explore the workings of the product 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)

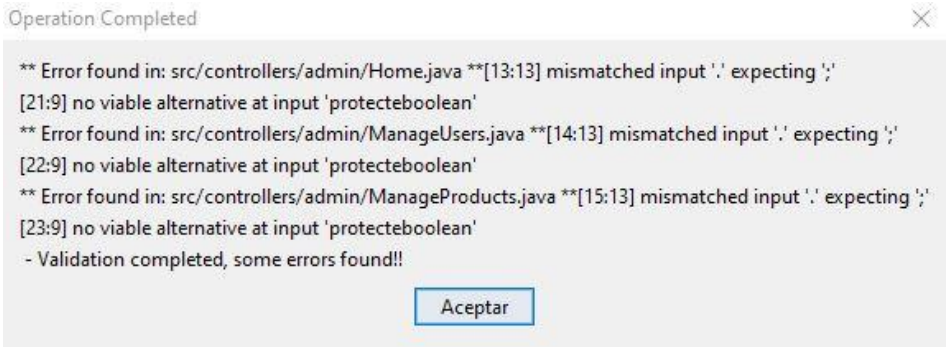
```
Fragment Login-AlterAdmin {
   Action: replace
   Priority: high
   FragmentationPoints: validation-function, validation-function, validation-
function, validation-function
   PointBracketsLan: java
   Destinations:     BasicViewsAdmin-Home,     CommentManagement-ManageComment,
ProductManagement-ManageProducts, 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)



It is important to note that VariaMos product 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 **¡Error! No se encuentra el origen de la referencia.** 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 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.