DYNA
ISSN: 0012-7353
Universidad Nacional de Colombia

Correa, Daniel; Mazo, Raúl; Giraldo-Goméz, Gloria Lucia
Fragment-oriented programming: a framework to design and implement software product line domain components
DYNA, vol. 85, no. 207, 2018, October-December, pp. 74-83
Universidad Nacional de Colombia

DOI: https://doi.org/10.15446/dyna.v85n207.71908

Available in: https://www.redalyc.org/articulo.oa?id=49658894010

UNIVERSIDAD NACIONAL DE COLOMBIA
SEDE MEDELLÍN
FACULTAD DE MINAS

DYNA
http://dyna.medellin.unal.edu.co/

Fragment-oriented programming: a framework to design and implement software product line domain components

Daniel Correa [a], Raúl Mazo [bc] & Gloria Lucia Giraldo-Goméz [a]

[a] *Facultad de Minas, Universidad Nacional de Colombia, Medellín, Colombia. dcorreab@unal.edu.co, glgiraldog@unal.edu.co*
[b] *CRI, Université Panthéon Sorbonne, Paris, France. raul.mazo@univ-paris1.fr*
[c] *GiDITIC, Universidad Eafit, Medellín, Colombia. rimazop@eafit.edu.co*

Received: April 26th, de 2018. Received in revised form: September 11th, 2018. Accepted: September 24th, 2018

Abstract
Software product lines facilitate the industrialization of software development. The main goal is to create a set of reusable software components for the rapid production of a software systems family. Many authors have proposed different approaches to design and implement the components of a product line. However, the construction and integration of these components continue to be a complex and time-consuming process. This paper introduces Fragment-oriented programming (FragOP), a framework to design and implement software product line domain components, and derive software products. FragOP is based on: (i) domain components, (ii) fragmentations points and (iii) fragments. FragOP was implemented in the VariaMos tool and using it we created a clothing stores software product line. We derived five different products, integrating automatically thousands of lines of code. On average, only three lines of code were manually modified; which provided preliminary evidence that using FragOP reduces manual intervention when integrating domain components.

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

Programación orientada a fragmentos: un marco para diseñar e implementar componentes de dominio de líneas de productos de software

Resumen
Las líneas de productos de software promueven la industrialización del desarrollo de software mediante la definición y ensamblaje de componentes de software. Actualmente existen diferentes propuestas para implementar estos componentes. Sin embargo, su construcción y ensamblaje continúa siendo un proceso complejo y que requiere mucho tiempo. Este artículo introduce la programación orientada a fragmentos (FragOP), la cual define un marco para implementar y ensamblar componentes de software. FragOP se basa en: (i) componentes de dominio, (ii) puntos de fragmentación y (iii) fragmentos. Utilizamos VariaMos y FragOP para crear una línea de productos de software, la cual contiene 20 componentes y miles de líneas de código. Se derivaron 5 productos y en promedio solo 3 líneas de código se modificaron manualmente para completar cada derivación; lo cual provee una evidencia preliminar de que la utilización de FragOP reduce la intervención manual en el proceso de integración de componentes de dominio.

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

1. Introduction

Software reuse has been practiced since programming began; its purpose is to improve software quality and productivity [1]. Software product lines (SPL) have become a successful, but challenging approach to software reuse [2]. A SPL is a collection of software-intensive systems sharing a common, managed set of characteristics that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

Software product line engineering (SPLE) has gained significant attention over the recent years. It is claimed that

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

SPLE provides a promising way to develop a large range of software intensive systems faster, better, and cheaper. SPLE considers two processes: (i) the domain engineering, which defines the commonalities and the variability of the SPL, and realizes the domain components; and (ii) the application engineering process, which derives the SPL applications from the domain artefacts [3].

A proper domain component development and management is crucial to take advantage of SPLE benefits. Currently, the domain component implementation and its subsequent integration (product derivation) continue to be a complex, time-consuming and expensive process [4].

1.1. Design and implementation of domain components

The design and implementation of domain components stage realizes the requirements identified during the domain analysis stage by constructing concrete domain reusable artefacts. These domain-specific artefacts are subsequently used throughout the product line (PL) to implement and improve the products. Domain components are thus critical to the successful implementation of the entire PL; and despite all the progress on this topic, there are still some research questions to study, for instance:

- How should domain reusable components be designed and implemented to guarantee their reuse at the application level?
- How to couple the components, so that the common and variable elements and their dependencies can be preserved during the implementation phase?
- How to reduce manual intervention when coupling components?
- How to deal with maintenance, evolution, and coupling of components developed in different software languages?

There are several approaches to design and implement components [5], such as: feature-oriented programming (FOP) [6], delta-oriented programming (DOP) [7], context-oriented programming (COP) [8], aspect-oriented programming (AOP) [9], service-oriented architecture (SOA) [10], CIDE [11], pure::variants [5], GenArch-P [12], and agents [13]; which are commonly grouped into two main approaches: annotative and compositional [5].

In **annotative approaches** such as pure::variants, CIDE and GenArch-P [5,11,12], developers simply introduce markers at the exact positions where a component should be extended [11]. Annotative approaches implement components with some form of explicit or implicit annotations in the components source code. The prototypical example is the use of #ifdef and #endif statements to surround the component code. In this approach, the code of all requirements is merged in a single code base, and annotations mark which code belongs to which requirement [5]. The use of annotations presents important advantages: (i) is easy to use, (ii) is already natively supported by many programming environments, and (iii) introduces markers at the exact positions where a component should be extended. However, it also presents some limitations: (i) the domain component files contain all source code variants, which increases the number of lines of codes; (ii) increases the number of relationships between the domain component file and other domain component files; and (iii) tends to make source code complex and therefore difficult to maintain and evolve [14].

In **compositional approaches** such as FOP and AOP [5], components are implemented in the form of composable units. In FOP, the software assets are developed in terms of "feature modules", which can be seen as increments of product functionality. For example, in the context of object-oriented programming (OOP), a feature module can introduce new classes, or refine existing classes by adding fields and methods, or by overriding existing methods. The use of compositional approaches has important advantages: (i) locates code implementing one or several requirements in a dedicated file, container, or module [5], and (ii) there is no need to specify (annotate in advance) the location in which a component should be extended. However, it also presents important limitations: (i) According to Kästner et al. [11] "compositional approaches only introduce new code fragments in positions in which the order does not matter. Thus, it is possible to introduce new classes into the program or new methods into a class, but not new statements at a fixed position inside a method". (ii) Commonly, compositional approaches are attached to a particular host language. For example: AspectJ and DeltaJ are attached to Java, and FeatureC++ is attached to C++. However, software products are not only made up by one type of software language, but by several kinds of files, such as HTML, CSS, JSP, MySQL, and configuration files; which lead the developers to manually inject glue code in order to connect and modify those files during the derivation activity. Consequently, developing modules that can be applied to multiple languages appears as an important concern [15].

1.2. Contribution

The main contribution of this paper is a framework that we call fragment-oriented programming (FragOP). FragOP integrates in a new proposal, some advantages of the compositional and annotative approaches, and dismisses some negative effects of these approaches. The second contribution of this paper is the improvement of a modeling tool called VariaMos, which enables to carry out a SPL domain implementation and product derivation (supporting FragOP). The third contribution is the design and development of a preliminary evaluation in which a SPL is developed with the use of FragOP and VariaMos; thus, we developed a video following the complete process [16].

The remainder of this paper is structured as follows. Section 2 introduces FragOP and its implementation. Section 3 describes the first four activities of FragOP related to the domain engineering process. Section 4 describes the last two activities related to the application engineering process. Section 5 shows a preliminary evaluation of FragOP, which contains an example of a SPL and presents the preliminary evaluation results. Section 6 presents related work and a comparison among AOP, FOP, DOP, annotative approaches and FragOP. Finally, Section 7 summarizes the contributions and presents future research directions.

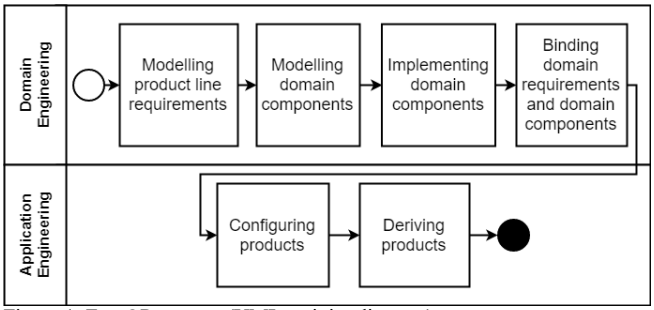

Figure 1. FragOP process (UML activity diagram).
Source: The authors.

2. Fragment-Oriented Programming (FragOP)

FragOP is a framework used to design and implement SPL domain components. It is a mix between compositional and annotative approaches, and is based on the definition of: (i) domain components, (ii) fragmentations points, which are annotations over the domain components code; and (iii) fragments, a new type of file which alters the domain components code. These three concepts are related to six activities that constitute the FragOP process (cf. Fig. 1): (1) Modelling PL requirements, (2) Modelling domain components, (3) Implementing domain components, (4) Binding domain requirements and domain components, (5) Configuring products, and (6) Deriving products.

To fully support the FragOP process we enhanced VariaMos [17]. VariaMos has been used in several SPL projects and approaches during recent years; this tool incorporates a language to represent and simulate families of systems and (self) adaptive systems [18]. We took advantage of some VariaMos capabilities such as: product line requirements modelling and product simulation; and we improved VariaMos with new capabilities to support the FragOP process: (1) domain components modelling, (2) the bind (or weave) the product line requirements model and the domain component model, (3) configure new products from the domain models, (4) derive the configured products, and (5) verify the domain models and the derived products. The activities of the FragOP process are explained and exemplified, within a simplified ClothingStores PL, in the next two sections.

3. Domain engineering process

3.1. Modelling product line requirements

In the first activity of the FragOP process, PL engineers should create the requirements model of the PL through, for instance, Feature Models (FMs) [19]. In a FM, a feature can be defined as a quality or a characteristic of a (software) system [5]. This activity is usually supported by a software tool which allows modelling product lines as presented in Fig. 2. This model corresponds to a clothing store PL called "ClothingStores"; which was designed as a very simple model. The main idea was to use this simple model as the base to explain in detail the FragOP process. A more complex model with more features and more complex relationships is

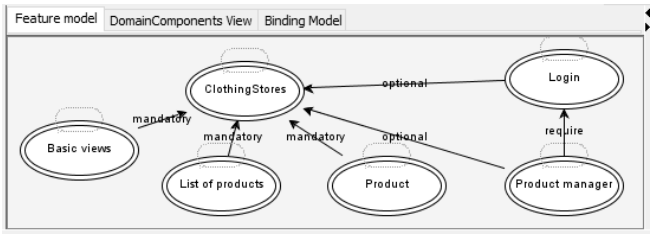

Figure 2. ClothingStores feature model (using VariaMos).
Source: The authors.

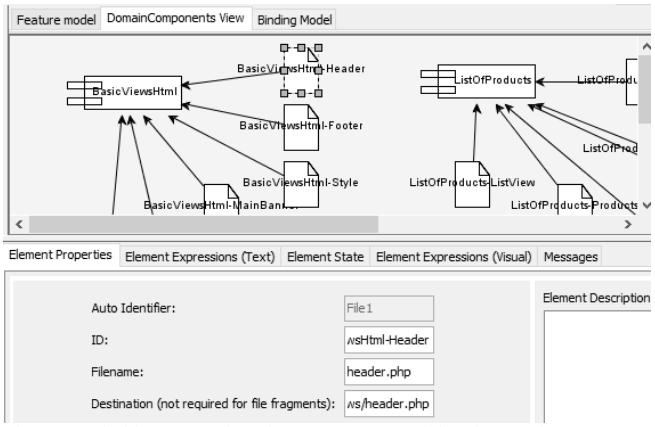

Figure 3. ClothingStores domain component model (using VariaMos).
Source: The authors.

presented later at Section 5 and used to preliminarily validate our approach. ClothingStores feature model contains the following features: ClothingStores refers to the root or name of the PL; Basic views refers to the basic views that any ClothingStores product must contain (e.g., headers, footers, sections and CSS styles); Product is a software service that stores product information and its operations; List of products represents a display service of all products of the store; Login represents a login service, and Product manager represents a product management service.

3.2. Modelling domain components

For the second activity, PL engineers should create a domain component model. This model provides an abstract view about how domain components are organized. The abstract view of each domain component can be operationalized with implementation files written in languages such as PHP, HTML and CSS as we did with the domain components of the ClothingStores PL. Fig. 3 shows an excerpt of the ClothingStores domain component model and its operationalization. Thus, the domain component model represents: (i) the domain components, (ii) their operationalization files, and (iii) information about the files (their identifiers, filenames and destinations in which will be derived).

3.3. Implementing domain components

In this activity, PL developers must create a domain component pool directory to store the corresponding files. The domain-independent structure presented in Fig. 4a.1 can

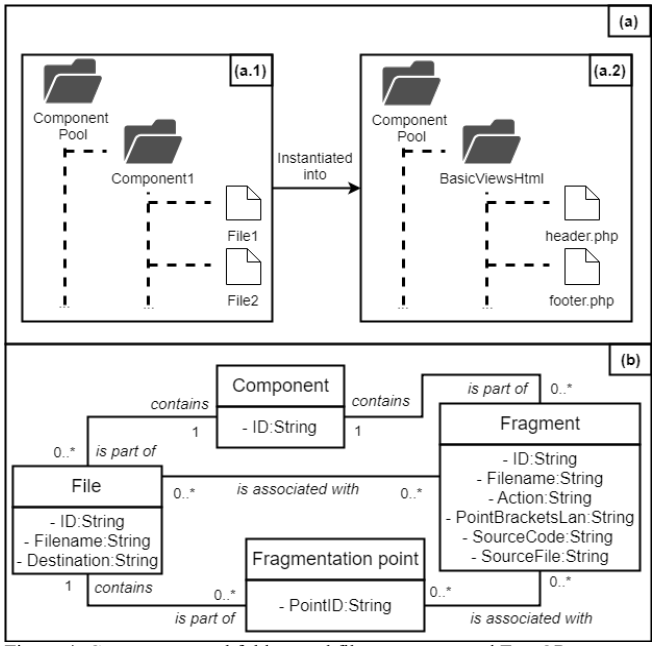

Figure 4. Component pool folders and files structure; and FragOP metamodel (UML class diagram).
Source: The authors.

```
<html><head><title><?php echo($title); ?>
</title></head><body>

<ul class="nav navbar-nav"><li><a href="<?php
echo base_url(); ?>index.php/Home/">
Home</a></li></ul>
```

```
<?php class ProductManager extends CI_controller
{
 public function __construct()
 {
    parent::__construct();
 }
}
```

Listing 1. BasicViewsHtml-Header (header.php) and ProductManager-Controller (ProductManager.php) component files.

be used as a template to create the domain-dependent folder structure (e.g., for a PL of Web applications), as presented in Fig. 4a.2. Fig. 4b shows the FragOP meta-model, which describes how domain component files and fragments are made up in detail, and how they are related to each other. The rest of this sub-section presents these concepts and shows how to implement each concept.

3.3.1. Implementation of files

Components are made up of files that represent, for instance, HTML, CSS, JavaScript, Java, and JSP files. Listing 1 shows an example of: (i) header.php file which is written in HTML and PHP and represents the header of an application. This code contains a menu, which corresponds to an unordered list with only one element (i.e., "Home") that is linked to the home section of the application. (ii) The ProductManager.php file which is written in PHP and represents a controller to manage the product information. In this case, it only contains a construct function.

3.3.2. Implementation of fragmentation points

The previous files could be refined with the addition of fragmentation points. A **fragmentation point** is an annotation (a very simple mark) that specifies a "point" in which a file can be modified. For example: other components such as Login or List of Products could require the modification of the BasicViewsHtml-Header file, specifically to add new elements to the previous menu. The Login component could also require the modification of the ProductManager-Controller replacing the construct function, with a new one that includes a call to the login class, to verify that only allowed users are using the ProductManager class. Listing 2 shows the fragmentation point shape.

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

Listing 2. 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 PHP, 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 the 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 later. 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 3 shows a fragmentation point example.

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

Listing 3. Fragmentation point shape example.

Based on the previous concepts and elements, the BasicViewsHtml-Header (header.php) and the ProductManager-Controller (ProductManager.php) files are refined as shown in Listing 4. As aforementioned, these two fragmentation points allow including, in the future, new header menu elements and replacing the construct function.

```
<html><head><title><?php echo($title); ?>
</title></head><body>
```

```
<ul class="nav navbar-nav"><li><a href="<?php
echo base_url(); ?>index.php/Home/">
Home</a></li>
<!--B-menu-modificator-->
</ul>
```

```
<?php class ProductManager extends CI_controller
{
<!--B-construct-modificator-->
 public function __construct()
 {
    parent::__construct();
 }
<!--E-construct-modificator-->
}
```

Listing 4. Refined BasicViewsHtml-Header (header.php) and ProductManager-Controller (ProductManager.php) component files.

3.3.3. Implementation of fragments

A **fragment** is a special type of file which allows developers to specify code alterations to the components files. It is worth to note that these alterations are designed only to be carried out at the application level when components are being integrated to derive new products (described in Section 4.2), which guarantees the reusability of the domain components. In general, a fragment respects the shape presented in Listing 5 and explained thereafter.

```
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]
}
```

Listing 5. Fragment shape.

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

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

- *add* (i) allows injecting a piece of code over specific PointIDs or (ii) allows adding a file over specific destination paths.
- *replace* allows replacing a piece of code over specific PointIDs or (ii) allows replacing a file over specific destination paths.
- *hide* allows hiding a piece of code over specific PointIDs (the pieces of code are placed inside a comment block).

Priority: *<high || medium || low>*. Priority specifies the fragment priority (high, medium or low). Fragments with high priority are integrated before fragments with medium or low priority. This feature could be useful in the case that two or more different fragments inject code over the same fragmentation point. For example: two different fragments could inject code over the header menu (in order to include new menu options). Depending on each fragment priority, one code will be injected first and the another will be injected second (which allows to define a code integration order).

PointBracketsLan: *<language>* (Optional). Language specifies the comment bracket language in which the fragmentation points are defined. For example: PHP, HTML and 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 will be injected in several places.

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

- *FileIDs* represents the domain component files to be altered.
- *Paths* represents the locations to add or replace a file.

SourceFile: *<filename>* (Optional). Filename represents the file to be added or replaced.

SourceCode: *<code>* (Optional). Code contains the source code that will be injected.

```
Fragment ListProducts-AlterHeader {
 Action: add
 Priority: high
 PointBracketsLan: html
 FragmentationPoints: menu-modificator
 Destinations: BasicViewsHtml-Header
 SourceCode:[ALTERCODE-FRAG]<li><a href="<?php
echo base_url();
?>index.php/Prod/">Products</a></li>
[/ALTERCODE-FRAG]
}
```

```
Fragment Login-AlterProductManager {
 Action: replace
 Priority: high
 PointBracketsLan: php
 FragmentationPoints: construct-modificator
 Destinations: ProductManager-Controller
 SourceCode: [ALTERCODE-FRAG]public function
__construct(){ parent::__construct();
  HttpSession session = request.getSession();
  String admin = (String)
session.getAttribute("admin"); if(admin != "1"){
response.sendRedirect("Home");return;}
}[ALTERCODE-FRAG]
}
```

Listing 6. ListProducts-AlterHeader (alterHeader.frag) and Login-AlterProductManager (alterProductManager.frag) fragments source code.

For a better understanding about how fragments work, let us consider the following example. Listing 6 shows (i) the alterHeader.frag code which specifies that the BasicViewsHtml-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. And (ii) the alterProductManager.frag code specifies that the ProductManager-Controller file (Destinations) will be altered in the construct-modificator (FragmentationPoints) with a high priority. In this case, the fragment will replace (Action) the ProductManager construct function with a new construct function (SourceCode).

3.4. Binding domain requirements and domain components

The last domain engineering activity consists on developing a binding model between the requirements model

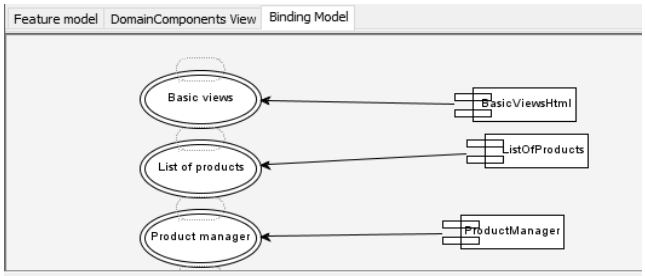

Figure 5. ClothingStores binding model (using VariaMos).
Source: The authors.

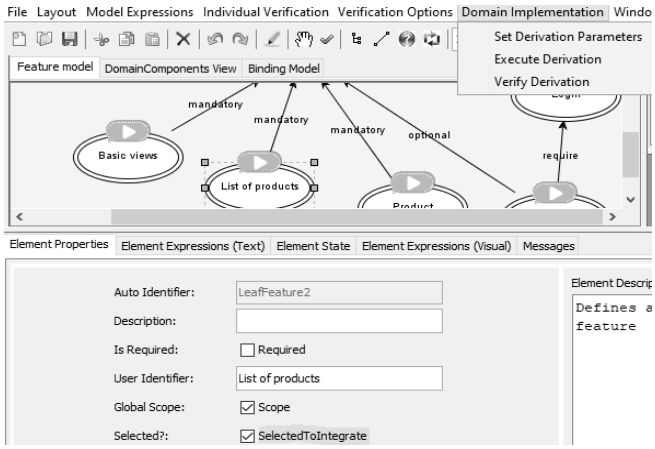

Figure 6. ClothingStores product configuration (using VariaMos).
Source: The authors.

and the implementation model. Fig. 5 shows a binding model between the FM and the domain component model. In this example, features are directly linked to components that operationalize them in a one-to-one relationship. This relationship goes from each domain component to the domain requirement (feature) it implements. To enhance this simple binding relationship, we plan to implement a constraints network [20] 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".

4. Application engineering process

4.1. Configuring products

The product configuration activity consists on selecting the specific features that a specific product will contain based on the stakeholder requirements. For example, this process in VariaMos consists in clicking on the features and marking the option "SelectedToIntegrate" in the "Elements Properties" panel. Fig. 6 shows an example of a product configuration activity in VariaMos, where the green mark above the features indicates that they were selected to be part of the product being configured.

4.2. Deriving products

The production derivation activity consists in following three steps: (i) setting derivation parameters, (ii) executing

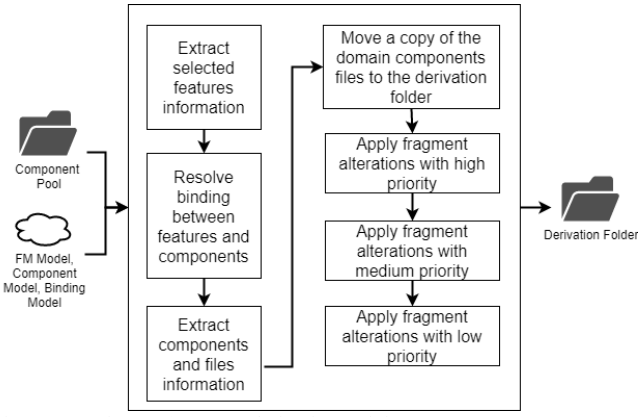

Figure 7. VariaMos (FragOP) derivation activity.
Source: The authors.

derivation and (iii) verifying derivation. Fig. 6 showed a menu with three options related to those steps. The **"Set Derivation Parameters"** option allows defining: (i) the "global assets folder path" which is the path where components and files are stored, and (ii) the "global integration folder path" which is the path where the integrated components of derived products are stored.

The **"Execute Derivation"** option allows deriving the software products based on an automated algorithm that follows a series of instructions as presented in Fig. 7. At the beginning, VariaMos takes the information from the component folder and the developed models. Then, based on these models, it resolves the binding relationships of the selected features to have the corresponding components and files. Following, VariaMos creates a copy of the components' files (from the domain component pool) and moves the copied files to the derivation folder. At the end, it applies the fragments' alterations over the copied components' files by priority order. The output is a derivation folder, which contains the integrated components and the final software product. The derivation algorithm also provides different alerts such as: invalid fragment definition, missing fields, invalid fragmentation point definition, invalid actions and invalid filenames and paths.

Continuing with the example, a component integration process of the components files defined in Listing 4 and the fragments defined in Listing 6, it generates new integrated files which are presented in Listing 7.

```
<html><head><title><?php echo($title); ?>
</title></head><body>

<ul class="nav navbar-nav"><li><a href="<?php
echo base_url(); ?>index.php/Home/">
Home</a></li>
<!--B-menu-modificator-->
<!--Code injected by: ListProducts-AlterHeader--
>
<li><a href="<?php echo
base_url();?>index.php/Prod/">Products
</a></li>
<!--Code injected by: ListProducts-AlterHeader--
>
</ul>
```

```
<?php class ProductManager extends CI_controller
{
<!--B-construct-modificator-->
<!--Code replaced by: Login-AlterProductManager-
->
  public function __construct()
  {
  parent::__construct();
  HttpSession session = request.getSession();
  String admin = (String)
session.getAttribute("admin");
  if(admin != "1"){
response.sendRedirect("Home");return;}
  }
<!--Code replaced by: Login-AlterProductManager-
->
<!--E-construct-modificator-->
}
```

Listing 7. Source code of the resulting header.php and ProductManager integrated components.

Finally, **"Verify Derivation"** allows finding grammar errors over the derived files. As we have shown, FragOP permits managing different component' files developed in different software languages; therefore, it allows injecting multiple pieces of code over the components' files. Based on that, it is critical to verify that the pieces of code are properly injected, and the resulting files contain valid grammar elements. To this aim, VariaMos uses ANother Tool for Language Recognition (ANTLR) [21] which is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions. VariaMos implemented ANTLR 4.7.1 and uses a series of parsers and lexers for languages such as: PHP, Java, CSS, MySQL, among others. Based on the derived component file extension, VariaMos analyses the grammar of each file and generates alerts if errors are found.

5. Preliminary evaluation of FragOP

In order to evaluate FragOP in practice, we have implemented a SPL in VariaMos. Based on the ClothingStores SPL presented in Section 3.1 we created a more complex SPL called "cStores" which includes new features such as: shop, cart, web management, sharing system, login, database management, offline payment and comments among others. In cStores, the components were built with Java, JSP, JavaScript, HTML, CSS and MySQL. A total of 25 features and their relationships were designed (as shown in Fig. 8.a), 20 components were built which included 80 files (46 domain component files and 36 fragments) containing more than 2000 lines of code (as shown in Fig. 8.b). We developed the corresponding FM, component model and binding model; these models, including the domain components code, the derived products, and even a video which shows the process the derive one product can be found online at a GitHub repository [16].

At the end of this implementation we derived five products. The results of our preliminary evaluation are summarized in Table 1 with the: (i) name of the product, (ii) quantity of the selected leaf features, (iii) linked files between the selected features and the corresponding component files, (iv) fragment lines of code automatically injected, (v) lines of code manually modified to finalize the product derivation, and (vi) time in seconds required to carry out the product derivation.

Table 1.
Results of derived products with VariaMos.

Name	Leaf features selected	Linked files	Fragment LOC injected	Manually LOC modified	Time to derive (Sec)
P1	5	22	31	3	0.04386
P2	8	35	83	3	0.05396
P3	13	55	193	3	0.08725
P4	15	65	348	3	0.13434
P5	20	80	437	3	0.18426

Source: The authors.

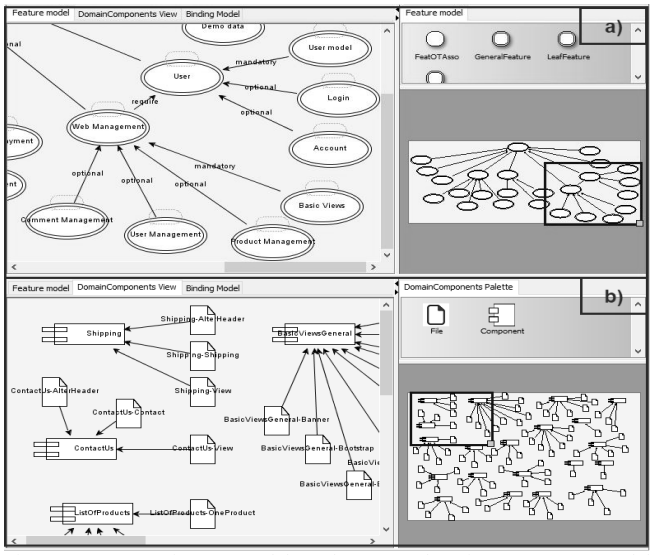

Figure 8. cStores feature model; and cStores domain component model (using VariaMos).
Source: The authors.

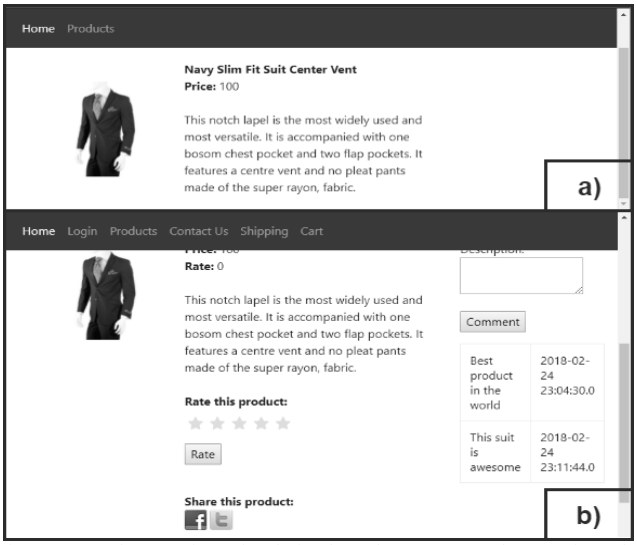

Figure 9. Product section of a derived product (P1); Product section of a derived product (P5).
Source: The authors.

The results show that only three lines of code were manually modified (in the database management configuration file to specify the database URL, name and

password) to complete each product derivation. Fig. 9 shows the P1 and P5 products running over a web browser; Fig. 9.a shows the P1 "product section" which contains a very basic configuration where the final user is able to read the product information; and Fig. 9.b shows the P5 "product section" which contains a complete configuration where the final user is able to rate, share, comment and add the product to the cart.

The view layer of each one of the product sections was represented with a file located at WebContent/views/oneproduct.jsp. This file contained 104 lines of code in the derived product P5 (including marks which show what component was responsible for each alteration) [22], and only 31 lines of code in the derived product P1 [23]. This shows that FragOP only injected the required code based on the product configuration needs.

The product derivation with FragOP was simple: we followed the steps defined in Section 4.2 as well as the following technical steps: we (i) created a "Dynamic web project" in Eclipse, (ii) added two libraries (JSTL and JDBC), (iii) created a database and imported a couple of auto-generated SQL files (which were located over the "derivation folder" by the "demo data" component), (iv) moved all the content from the "derivation folder" to the Eclipse project root folder, (v) modified the database configuration file, and (vi) executed the web project.

It is worth noting that these results present some improvements compared to other approaches: (i) versus compositional approaches, which are commonly attached to a host language, the P5 included a total of 437 LOC automatically injected (223 were related to Java files and 214 to other languages). Deriving P5 with a Java compositional approach allows to automatically inject 223 LOC as a maximum; the other 214 LOC should be manually injected. (ii) versus annotative approaches, the results could vary depending on the annotative approach language support; however, as mentioned before, annotative approaches inject code variations inside the domain components, which is not the case in FragOP.

Table 2.
Comparison of different approaches part A

	AOP	**DOP**	**FOP**
Approach support	Compositional	Compositional	Compositional
Granularity	Altering the static structure of components by introducing new attributes and operations	Introducing, modifying, removing or extending methods in existing classes	Introducing or extending methods in existing classes
Domain implem. mechanism	Component files, and Aspects	Delta modules	Feature modules
Comp. units' separation	Physically separated	Physically separated	Physically separated
Language independence	Usually depending on a particular host language	Depends on a particular host language	Depends on a particular host language

Source: The authors based on [7,2].

Table 3.
Comparison of different approaches part B

	Annotative	FragOP
Approach support	Annotative	Compositional and Annotative
Granularity	Introducing markers at the exact positions where a component should be extended	Introducing fragmentation points at the exact positions where a component should be extended
Domain implem. mechanism	Component files with annotations	Component files with frag. points, and fragments
Comp. units' separation	Usually physically integrated	Physically separated with frag. points
Language independence	Language-independent	Supporting multiple language

Source: The authors based on [7,2].

6. Related work

FOP, DOP, AOP and annotative approaches have been developed to design and implement domain components. **FOP** allows developing software assets in terms of feature modules [6], which can be seen as increments of product functionality. For example, in the context of OOP, a feature module can introduce new classes or refine existing ones by adding fields and methods, or by overriding existing methods. **DOP** allows the assets to be defined in terms of delta modules, which can also be seen as increments of product functionality. DOP is an extension of FOP, which is also a compositional approach. Delta modules generalize feature modules by allowing the removal of functionality [7]. **AOP** allows the assets to be defined in terms of aspects. An aspect encapsulates a cross-cutting feature into a modular unit. AOP has been used to implement SPLs by the composition of aspects, through mechanisms such as pointcut and advice [9] with the aim of making crosscutting features more modular and evolutionary. **Annotative approaches** allow the developers to add "annotations" to the assets at arbitrary levels of granularity. In annotative approaches, developers simply introduce markers at the exact positions where an asset should be extended [11].

There are other studies in this area; for instance, CaesarJ [24] proposes a combination of feature modules and aspects to extend FOP with means to modularize cross-cutting concerns; Kästner & Apel [25] compare both groups of approaches and propose an integration of both approaches; and Apel & Hutchins [26] propose gDeep as a possible unifying foundation of languages for FOP.

Tables 2 and 3 summarize the comparison of the previous approaches and FragOP based on five different perspectives: **Approach support** refers to the approach classification. **Granularity** is closely related to the approach expressiveness. Very coarse-grained approaches only assemble files in a directory, while fine-grained approaches allow modifications on the level of methods, statements, parameters or even expressions. Annotative approaches are more fine-grained than compositional ones. **Domain implementation mechanism** refers to the way in which each approach realizes the domain implementation. **Component units' separation** refers to the way in which components are developed. Compositional approaches implement

components as distinct (physically separated) code units. Annotative approaches commonly use #ifdef and #endif statements to surround the component code; but including the code variants and relationship inside the domain components code. FragOP introduces fragmentation points in the components code, but uses separated fragments that contain the code to be injected. **Language independence** refers to the applicability of each approach to be used independently from the language. Annotative approaches are line-based or character-based, and therefore language-independent. Compositional approaches are usually dependent on a particular host language. As shown, FragOP could be used for implementing domain components that contain multiple languages.

7. Conclusions and future work

This paper presents FragOP, a framework used to design and implement SPL domain components; which is a mix between a compositional and an annotative approach. FragOP takes advantage of the main benefits of each approach and tries to dismiss the disadvantages of each one. FragOP is based on the definition of (i) domain components, (ii) fragmentations points and (iii) fragments. We enhanced the VariaMos software tool to support the FragOP process and to carry out a preliminary evaluation of the new approach. In particular, we (i) designed and implemented clothing stores SPL and (ii) used FragOP to derive five products. The results showed that only three LOC were manually modified in order to complete the product derivation. We also showed that FragOP could be used to develop domain components in languages such as Java, PHP, HTML, SQL, CSS, and JSP.

In the short term we plan to (i) improve FragOP and its VariaMos implementation to support complex binding relationships; (ii) include the product customization process inside the FragOP approach; (iii) increase the number of programming languages supported by FragOP with the use of ANTLR; and (iv) support other variability models such as Orthogonal Variability Model (OVM). In addition, we think that further studies about how to deal with dynamic composition and dynamic binding are important research directions that will make FragOP suitable to be used in the context of dynamic product lines and domain-derived self-adaptive systems.

From an experimentation point of view, we find valuable and therefore we invite our colleagues and we ourselves account (i) to develop rigorous experiments to validate the FragOP benefits; (ii) to compare the different approaches to design and implement the domain components [27,28]; and (iii) to develop more software product lines (e.g., at industrial level) with the FragOP approach in order to provide valuable evidence about the benefits and limitations of FragOP.

Another research topic that is not addressed in this paper is the downstream of economic benefits behind the use of FragOP in industry. For example, one could raise the question how much can software companies really benefit with the use of FragOP in their projects? How much does it cost to implement FragOP? These complex issues have yet to be investigated. Finally, how to improve the fragment quality and how to detect the errors before the product derivation activity remains as an important research area.

References

[1] Frakes, W.B. and Kang, K., Software reuse research: Status and future. IEEE transactions on Software Engineering, 31(7), pp. 529-536, 2005. DOI: 10.1109/TSE.2005.85

[2] Laguna, M.A. and Marqués, J.M., UML support for designing software product lines: The package merge mechanism. J. of Universal Computer Science, 16(17), pp. 2313-2332, 2010. DOI: 10.3217/jucs-016-17-2313

[3] Botterweck, G., Lee, K. and Thiel, S., Automating product derivation in software product line engineering, In: Software Engineering, pp. 177-182, 2009.

[4] Azanza, M., Díaz, O. and Trujillo, S., Software Factories: Describing the Assembly Process. In ICSP, pp. 126-137, 2010. DOI: 10.1007/978-3-642-14347-2_12

[5] Apel, S., Batory, D., Kästner, C. and Saake, G., Feature-oriented software product lines: concepts and implementation. Springer Science & Business Media, 2013. DOI: 10.1007/978-3-642-37521-7

[6] Prehofer, C., Feature-oriented programming: A fresh look at objects, Proceedings of the Europ. Conf. Object-Oriented Programming, pp. 419-443, 1997. DOI: 10.1007/BFb0053389

[7] Schaefer, I., Bettini, L., Bono, V., Damiani, F. and Tanzarella, N., Delta-oriented programming of software product lines, In: SPLC. LNCS, 6287, pp. 77-91, 2010. DOI: 10.1007/978-3-642-15579-6_6

[8] Salvaneschi, G., Ghezzi, C. and Pradella, M., Context-oriented programming: A software engineering perspective. J. of Systems and Software, 85(8), pp. 1801-1817, 2012. DOI: 10.1016/j.jss.2012.03.024

[9] Tizzei, L.P., Rubira, C.M. and Lee, J., An aspect-based feature model for architecting component product lines, In: SEAA, pp. 85-92, 2012. DOI: 10.1109/SEAA.2012.64

[10] Alzahmi, S., Matar, M.A. and Mizouni, R., A practical tool for automating service oriented software product lines derivation, Proceedings of the 8th Int. Symposium on Service Oriented System Engineering (SOSE), pp. 90-97, 2014. DOI: 10.1109/SOSE.2014.16

[11] Kästner, C., Apel, S. and Kuhlemann, M., Granularity in software product lines, Procedings of the 30th Int. Conf. on Software Engineering (ICSE), pp. 311-320, 2008. DOI: 10.1145/1368088.1368131

[12] Aleixo, F.A., Kulesza, U. and Junior, E.A.O., Modeling variabilities from software process lines with compositional and annotative techniques: A quantitative study. Proceedings of the Int. Conf. on Product Focused Software Process Improvement, pp. 153-168, Springer, Berlin, 2013. DOI: 10.1007/978-3-642-39259-7_14

[13] Jordan, H.R., Russell, S.E., O'Hare, G.M. and Collier, R.W., Reuse by inheritance in agent programming languages. In: Intelligent Distributed Computing V, Studies in Computational Intelligence, pp. 279-289, Springer, Berlin, Heidelberg, 2011. DOI: 10.1007/978-3-642-24013-3_30

[14] Le, D.M., Lee, H., Kang, K.C. and Keun, L., Validating consistency between a feature model and its implementation. In: ICSR, pp. 1-16, 2013. DOI: 10.1007/978-3-642-38977-1_1

[15] Kästner, C., Apel, S. and Ostermann, K., The road to feature modularity?, Proceedings of the 15th Int. Software Product Line Conference, 2, pp. 5, 2011. DOI: 10.1145/2019136.2019142

[16] FragOP, GitHub [Online]. [date of reference: April 26th of 2018]. Available at: https://github.com/danielgara/FragOP

[17] VariaMos - Families of systems & SAS modeling tool [Online]. [date of reference: September 9th of 2018]. Available at: https://variamos.com/home/

[18] Mazo, R., Muñoz-Fernández, J.C., Rincón, L., Salinesi, C. and Tamura, G., VariaMos: an extensible tool for engineering (dynamic) product lines, Proceedings of the 19th Int. Conf. on Software Product Line, pp. 374-379, 2015. DOI: 10.1145/2791060.2791103

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

[20] Lecoutre, C., Constraint Networks. Wiley-IEEE Press, 2009. DOI: 10.1002/9780470611821
[21] Parr, T., The definitive ANTLR 4 reference. Pragmatic Bookshelf, 2013.
[22] Oneproduct.jsp file P5 – FragOP, GitHub [Online]. [date of reference: September 13th of 2018]. Available at: https://github.com/danielgara/FragOP/blob/master/cstoresp5/integrated/WebContent/views/oneproduct.jsp
[23] Oneproduct.jsp file P1 – FragOP, GitHub [Online]. [date of reference: September 13th of 2018]. Available at: https://github.com/danielgara/FragOP/blob/master/cstoresp1/integrated/WebContent/views/oneproduct.jsp
[24] Mezini, M. and Ostermann, K., Variability management with feature-oriented programming and aspects. In: ACM SIGSOFT Software Engineering Notes, 29(6), pp. 127-136, 2004. DOI: 10.1145/1041685.1029915
[25] Kästner, C. and Apel, S., Integrating compositional and annotative approaches for product line engineering, Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering, pp. 35-40, 2008.
[26] Apel, S. and Hutchins, D., A calculus for uniform feature composition. ACM Transactions on Programming Languages and Systems (TOPLAS), 32(5), pp. 19, 2010. DOI: 10.1145/1745312.1745316
[27] Rincón, L. and Mazo, R., Análisis y diseño de componentes reutilizables de dominio. In: Guía para la adopción industrial de líneas de productos de software, pp. 259-306, Editorial Eafit, Medellín, Colombia, 2018. ISBN 978-958-720-506-0
[28] Correa, D. and Mazo, R., Implementación de componentes reutilizables de dominio. In: Guía para la adopción industrial de líneas de productos de software, pp. 307-368, Editorial Eafit, Medellín, Colombia, 2018. ISBN 978-958-720-506-0

D. Correa, received the BSc. Eng in Systems and Informatics Engineering in 2012, and the MSc. degree in Systems and Informatics Engineering in 2015 all of them from the Universidad Nacional de Colombia. Medellin, Colombia. He is currently a PhD candidate in Systems and Informatics Engineering for the Universidad Nacional de Colombia; and he is an auxiliary professor in the Computing and Decision Sciences Department, Universidad Nacional de Colombia. His research interests include: software product lines, software engineering and software frameworks.
ORCID: 0000-0001-5767-2447

R. Mazo, is associate professor at Paris1 Panthéon Sorbonne University from September 2012, and visiting professor at Eafit University (Colombia) from September 2016. He received a BSc. in Computer Science Engineering in 2005 from the University of Antioquia (Medellin, Colombia), a MSc. of Science degree in Information Systems in 2008 and a PhD. degree in Computer Science in 2011, both from the Panthéon Sorbonne University. His research and teaching topics include: software engineering, and (dynamic) product line engineering.
ORCID: 0000-0003-0629-1542

G.L. Giraldo-Gómez, is associate professor in the Computing and Decision Sciences Department. Universidad Nacional de Colombia, located in Medellín. She is BSc. in System Engineer of the Antioquia University in Medellín, Colombia and the same university she obtained a title as Sp. in Electronic Sciences and Informatics. She is MSc. in theory and database engineering at Paris I Pantheon Sorbonne University, France. She is PhD. in Computer Science at Paris-Sud 11 University, Orsay, France. Her research areas are the software engineering, requirements of software engineering, software product lines, ontologies and information systems.
ORCID: 0000-0001-7487-2707