On the Use of Feature-Oriented Programming for Evolving Software Product Lines – A Comparative Study

Gabriel Coutinho Sousa Ferreira[1], Felipe Nunes Gaia[1], Eduardo Figueiredo[2] and Marcelo de Almeida Maia[1]

[1]Federal University of Uberlândia, Brazil
[2] Department of Computer Science, Federal University of Minas Gerais, Brazil
{gabriel, felipegaia}@mestrado.ufu.br, figueiredo@dcc.ufmg.br, marcmaia@facom.ufu.br

Abstract. Feature-oriented programming (FOP) is a programming technique based on composition mechanisms, called refinements. It is often assumed that feature-oriented programming is more suitable than other variability mechanisms for implementing Software Product Lines (SPLs). However, there is no empirical evidence to support this claim. In fact, recent research work found out that some composition mechanisms might degenerate the SPL modularity and stability. However, there is no study investigating these properties focusing on the FOP composition mechanisms. This paper presents quantitative and qualitative analysis of how feature modularity and change propagation behave in the context of two evolving SPLs, namely WebStore and MobileMedia. Quantitative data have been collected from the SPLs developed in three different variability mechanisms: FOP refinements, conditional compilation, and object-oriented design patterns. Our results suggest that FOP requires few changes in source code and a balanced number of added modules, providing better support than other techniques for non-intrusive insertions. Therefore, it adheres closer to the Open-Closed principle. Additionally, FOP seems to be more effective tackling modularity degeneration, by avoiding feature tangling and scattering in source code, than conditional compilation and design patterns. These results are based not only on the variability mechanism itself, but also on careful SPL design. However, the aforementioned results are weaker when the design needs to cope with crosscutting and fine-grained features.

Keywords: Software product lines, Feature-oriented programming, Variability management, Design patterns, Conditional compilation.

1 Introduction

Software Product Lines (SPLs) [17] are known to enable large scale reuse across applications that share a similar domain. The potential benefits of SPLs are achieved through a software architecture designed to increase reuse of features in several SPL products. There are common features found on all products of the product line (known as mandatory features) and variable features that allow distinguishing between products in a product line (generally represented by optional or alternative

features). Variable features define points of variation and their role is to permit the instantiation of different products by enabling or disabling specific SPL functionality.

As in any software life cycle, changes in SPLs are expected and must be accommodated [30]. When it comes to SPLs, these changes have even more impact, since changes to attend new stakeholder requests [17], may affect several products. In an ideal scenario, the introduction of new features on an SPL should be conducted by inserting components that encapsulate new or enhanced features [11], minimizing ripple effects of changes.

Variability management is a key factor to be considered when evolving SPLs. Several mechanisms, whether annotative or compositional [34], support variability management. Examples of variability mechanisms are FOP refinements [12, 14], conditional compilation [2, 5], and object-oriented design patterns [27]. To be considered effective, these mechanisms must guarantee the SPL architecture stability and, at the same time, facilitate future changes. In order to ensure these requirements, the variability mechanisms should minimize changes and should not degenerate modularity. In other words, variability mechanisms should support non-intrusive and self-contained changes that favor insertions and do not require deep modifications in existent components. These requirements are related to the Open-Closed principle [42], which states that "software should be open for extension, but closed for modification". This principle can be achieved with mechanisms that add new artifacts to extend the system functionality, but minimize the amount of modifications in current code.

Our work targets to find out how variability mechanisms behave in terms of modularity and change propagation on specific SPL change scenarios. In this context, this paper presents two case studies that evaluates comparatively three mechanisms for implementing variability on evolving software product lines: conditional compilation (CC), object-oriented design patterns (DP) and feature-oriented programming (FOP). This investigation extends our preliminary work [22] and focuses on the evolution of two software product lines, called WebStore and MobileMedia (Section 3). We choose these SPLs because they were available to us and have been used in previous studies with similar purpose [16, 24]. Altogether, we considered five versions of WebStore SPL and seven versions of MobileMedia SPL.

In this study, we analyzed and compared the implementation of variability mechanisms to evolve two SPLs, using a pure FOP language (Jak) [14] and other two OO-based programming techniques. This work evaluated the compositional mechanisms available in FOP by using the other two variability techniques as baseline. The SPL implementation assessment was based on modularity and change propagation metrics recurrently used to quantify separation of concerns and change impacts [16, 18, 26, 47, 52]. Moreover, our study contributes to build up a body of knowledge that allows the comparison of AHEAD and other FOP or non-FOP approaches.

This paper extends the previous SBLP paper with two major contributions, as follows.

- A new case study using the MobileMedia SPL; our preliminary work relies only on the WebStore SPL. MobileMedia is larger than WebStore not only in terms of number of components but also with respect to the variety of change scenarios. Therefore, this new case study helped us to (i) increase the results

reliability, (ii) come up with new findings, and (iii) reduce threats to study validity.

- We also provide more detailed data analysis and a deeper discussion about the new findings. The analyses, that now considered data collected from both SPLs, reinforced the findings from the first case study and revealed several new ones. For instance, based on the MobileMedia case study, we observed that the SoC (*Separation of Concerns*) metrics tend to be less discriminative on larger systems.

Therefore, the novel contributions of this extended paper are threefold.

- The development of public benchmark data with 113,152 data points concerning four feature modularity metrics extracted from two SPLs implemented with three different variability mechanisms in 12 different versions.
- The qualitative and quantitative analysis framework for change propagation and feature modularity metrics that can be reused in further replications of this study.
- Discussion and observations based on the obtained data about the role and the singular applicability of each variability mechanism in the context of evolving software product lines.

The rest of this paper is organized as follows. In Section 2, the implementation mechanisms used in the case study are revisited. Section 3 presents the study setting, including the target SPLs and their respective change scenarios. Section 4 analyzes change measures through different releases. In Section 5, the modularity of WebStore and MobileMedia SPLs are quantitatively analyzed and discussed. Section 6 presents the threats to validity of this study. Section 7 presents related work. Finally, Section 8 concludes this paper.

2 Variability Mechanisms for Software Product Lines

This section revisits some concepts about the three techniques evaluated in this study: conditional compilation (CC), object-oriented design patterns (DP) and feature-oriented programming (FOP). We choose conditional compilation and design patterns because these are the state-of-the-practice options adopted in SPL industry [5, 42]. Although there are other approaches that could be used to represent the feature-oriented paradigm [43], we chose AHEAD because it has been widely studied [8, 10, 12, 14, 34].

2.1 Conditional Compilation (CC)

The CC approach used in this work is a well-known technique for handling software variability [2, 5]. It has been used in programming languages like C for decades and it is also available in object-oriented languages such as C++ [31]. Basically, preprocessor directives indicate pieces of code that should be compiled or

not based on the value of preprocessor variables. The pieces of code can be marked at granularity of a single line of code or to a whole file.

The code snippet in Listing 1 shows the use of conditional compilation mechanism by inserting the pre-processing directives. In this example, there are some directives that characterize the CC way of handling variability. The directive `//#if defined(Paypal)` in line 5, for instance, indicates the beginning of code belonging to the Paypal feature. The directive `#endif` in line 9 determines the end of code associated to this feature. The identifier `Paypal` used in the construction of these directives is associated with a Boolean value defined in a configuration file for each product of the line. This value indicates the presence of the feature in a product, and consequently, the inclusion of the bounded piece of code in the compiled product.

```
1  private ControllerAction selectPaymentMethod(...) {
2    if (paymentType.equals("Default")) {
3      paymentAction = new GoToAction("payment.jsp");
4    }
5    //#if defined(Paypal)
6    if (paymentType.equals("Paypal")) {
7      paymentAction = new GoToAction("paypal.jsp");
8    }
9    //#endif
10   return paymentAction;
11 }
```

Listing 1. Example of variability management with conditional compilation.

2.2 Object-Oriented Design Patterns (DP)

Object-oriented design patterns became widely used with the Gang of Four book [27]. Design patterns rely on object-oriented mechanisms, such as dynamic binding and polymorphism [15], to handle variability in SPLs. The example in Listings 2, 3 and 4 shows classes that implement the Decorator design pattern [27]. The purpose of this decoration is to provide an entry point to add a feature behavior in a pluggable way. This pattern was designed so that multiple decorators can be stacked on top of each other, each time adding new feature functionality to an overridden method. Optional features were mostly implemented with decorators, following the aforementioned stack method.

Both classes presented in Listings 2 and 3 implement the `Decorator` interface, which contains the `init` method declaration. Line 5 in Listing 4 presents the `init` method in the `PaypalControllerDecorator` class that decorates the `init` method of a concrete component (Listing 2). The decoration is supported by dynamic binding mechanism and the target class will contain both actions: `goToHome` and `goToPaypal`.

2.3 Feature-Oriented Programming (FOP)

Feature oriented programming (FOP) [45] is a paradigm for software modularization by considering features as a major abstraction. This work relies on

AHEAD [12, 14], which is an approach to support FOP based on step-wise refinements. The main idea behind AHEAD is that programs are constants and features are added to programs using refinement functions. We chose Jak (AHEAD) because it is a stable language and is widely studied in the literature related to feature-oriented programming [8, 10, 12, 14, 34]. The code snippets in Listings 5 and 6 show examples of a class and a class refinement used to implement variation points.

The example in Listing 5 shows an ordinary base class that implements a default action for a *checkout form* and Listing 6 presents the respective FOP class refinement that considers *Paypal payment* in checkout. Line 1 of Listing 6 is a clause that indicates a layer of the class refinements. The `paypal` identifier in line 1 is used to compose the layers according to some pre-established order in the SPL configuration script. In general, the composition process of FOP is similar to the behavior of a pipeline. A base class is refined by one or more refinements in a certain order and the result is a class containing the source code of the base class and all class refinements from other features included. The creation of a product is specified in a configuration script that simply indicates the order of composition of layers.

```
1  public class ControllerMapper implements Decorator {
2    protected Map actions = new HashMap();

3    public ControllerMapper() {
4      init();
5    }
6    public void addAction(String an, ControllerAction ca) {
7      actions.put(an, ca);
8    }
9    public void init() {
10     addAction("goToHome", new GoToAction("home.jsp"));
11   }
12   public ControllerAction getAction(String an) {
13     return actions.containsKey(an) ? actions.get(an) : null;
14   }
15 }
```

Listing 2. Example of variability mechanism with the Decorator pattern (Concrete Component)

```
1  abstract class ControllerDecorator implements Decorator {
2    protected Decorator mapper;
3    protected Map controllerMap = new HashMap();
4    public ControllerDecorator(Decorator m) {
5      this.mapper = m;
6      init();
7    }
8    public abstract void init();
9    public void addAction(String an, ControllerAction ca) {
10     controllerMap.put(an, ca);
11   }
12   public ControllerAction getAction(String an) {
13     return controllerMap.containsKey(an) ?
               controllerMap.get(an) : mapper.getAction(an);
14   }
15 }
```

Listing 3. Example of variability mechanism with the Decorator pattern (Abstract Decorator)

```
1  public class PaypalControllerDecorator extends ControllerDecorator {
2    public PaypalControllerDecorator (Decorator m) {
3       super(m);
4    }
5    public void init() {
6      addAction("goToPaypal", new GoToAction("paypal.jsp"));
7    }
8  }
```

Listing 4. Example of variability mechanism with the Decorator pattern (Concrete Decorator)

```
1  public class ProcessCheckoutFormAction {
2    private ControllerAction selectPayment(...) {
3      if (paymentType.equals("Default")) {
4        paymentAction = new GoToAction("payment.jsp");
5      }
6      return paymentAction;
7    }
8  }
```

Listing 5. Example of variability mechanism with FOP (base class)

```
1  layer paypal;
2  refines class ProcessCheckoutFormAction {
3    private ControllerAction selectPayment(...) {
4      Super(ControllerAction, String).selectPayment(...);
5      if (paymentType.equals("Paypal")) {
6        paymentAction = new GoToAction("paypal.jsp");
7      }
8    }
9  }
```

Listing 6. Example of variability mechanism with FOP (class refinement)

3 Study Setting

This section describes the study based on the analysis of two evolving software product lines. One of these SPLs was constructed from scratch and the other was adapted and implemented in pure Java and AHEAD to complete the infrastructure setting. The study was developed to answer the research questions described in the sequel.

3.1 Research Questions

The following research questions were posed in order to better understand the impact of using feature-oriented programming in the SPL evolution:

RQ1) Does the use of FOP has smoother change propagation impact than CC and DP during the evolution of an SPL?

RQ2) Does the use of FOP provides more modular and stable design than CC and DP of the SPL features in evolution?

3.2 Infrastructure Setting

The independent variable of this study is the variability mechanism used to implement SPLs, namely, *Conditional Compilation* (CC), *Object-oriented Design Patterns* (DP) *and Feature-oriented programming* (FP). Two subject systems are used to analyze the behavior of the dependent variables: change propagation measures and modularity metrics. The study was organized in four phases: (1) construction of two subject SPLs with complete releases that correspond to their respective change scenarios using the three techniques aforementioned for each one, CC, DP and FOP, (2) manual feature assignment of all produced source code, (3) change propagation measurement [52] and modularity metrics calculation [47] and (4) quantitative and qualitative analysis of the results.

In the first phase, the first two authors implemented, from the scratch, all the source code of WebStore SPL. The FOP solution of WebStore was developed first and it contemplates the five releases already mentioned. The other solutions were implemented next, using the FOP solution as baseline. The other SPL, MobileMedia [24], was already used in previous studies. There is a full CC implementation of this SPL available and, thus, only DP and FOP solutions had to be implemented.

In the second phase, all code was manually assigned according to each SPL feature. The feature assignment task was performed using the Prune Dependency Rule proposed in [20]. The concrete result of this phase was text files, one for each source code file, where each line was marked with the corresponding feature. The feature assignment task was done so that the developers of a version do not mark their own produced code. We have considered only source code files in our analysis. Other files, such as makefiles and configuration scripts, generally represent a minor fraction of artifacts in maintenance activities. Thus, we have not considered them in our study.

In the third phase, change propagation measures [52] were collected and modularity metrics related to Separation of Concerns [47] were calculated. We have made all calculations using the metrics formulas by manually counting the feature lines. Finally, the results were analyzed in the fourth phase. The next sections present the analyzed SPLs, WebStore and MobileMedia, and discuss their change scenarios.

3.3 The Evolved WebStore SPL

The first target SPL was developed to represent major features of an interactive web store. It was developed for academic purpose, inspired by a sample application called Java Pet Store[1], focusing on the key features available in real web store systems. We decide to use WebStore because Java Pet Store is a public available application and it was used in a previous study with similar purpose [16]. We have also designed four change scenarios (the same for all studied techniques – CC, DP, and FOP) that could exercise the SPL evolution.

WebStore is an SPL for applications that manage products and their categories, show products catalog and control payments. Table 1 provides some measures about the size of the SPL implementation in terms of number of components, number of methods and number of lines of source code (LOC). Classes and class refinements

[1] http://www.oracle.com/technetwork/java/petstore1-3-1-02-139690.html

were accounted as components. The number of components varies from 23 (CC) to 47 (FOP).

Table 1. WebStore SPL implementation

	CC					FOP					DP				
	R.1	**R.2**	**R.3**	**R.4**	**R.5**	**R.1**	**R.2**	**R.3**	**R.4**	**R.5**	**R.1**	**R.2**	**R.3**	**R.4**	**R.5**
#Components	23	23	26	26	26	25	35	44	41	47	28	32	38	40	44
#Methods	138	139	165	164	167	150	170	200	198	208	142	147	175	177	182
LOC (aprox.)	885	900	1045	1052	1066	945	1077	1257	1244	1303	915	950	1107	1121	1149

Figure 1 presents a simplified view of the WebStore SPL feature model [13]. Examples of core features are CategoryManagement and ProductManagement. In addition, some optional features are DisplayByCategory and BankSlip. We use numbers in the top right-hand corner of a feature in Figure 1 to indicate in which release the feature was included (see Table 2).

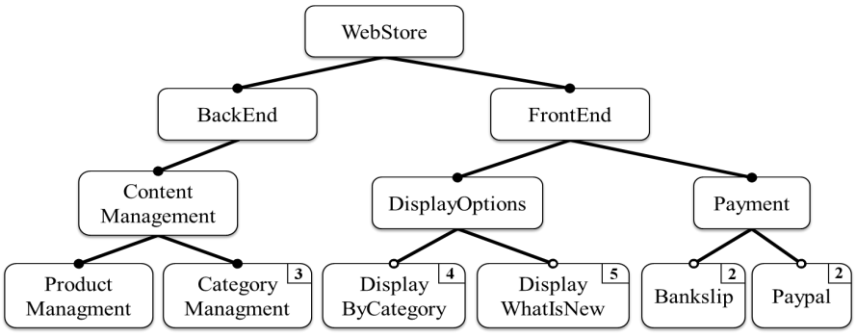

Figure 1. WebStore Basic Feature Model

The WebStore versions are very similar from the design point-of-view, even though they are implemented using three distinct variability mechanisms. In all versions the release R1 contains the core of target SPL. All subsequent releases were designed to incorporate the required changes in order to include the corresponding optional features and to transform optional features into mandatory. For instance, the version that uses FOP was developed first, trying to maximize the decomposition of the features. All components related to features that have not shared any piece of code were partitioned into one or more parts. This explains why release R1 in FOP contains more components than release R1 that uses CC. All subsequent scenarios were incorporated using insertions, modifications or removals of classes and class refinements.

In CC versions, scenarios were incorporated in the form of new classes and changes in existing classes. Only code of optional features was marked with CC directives, such as `#ifdef` and `#endif` (Section 2.1). On the other hand, the

WebStore version that uses object-oriented design patterns was implemented mainly based on two design patterns: Abstract Factory and Decorator [27]. Their roles are to mimic FOP mechanisms, in order to provide smooth feature code additions and different product instantiations.

3.4 Change Scenarios

As aforementioned, in the first phase of our investigation we designed and implemented a set of change scenarios. A total of four change scenarios were incorporated into WebStore, resulting in five releases. Table 2 summarizes changes made in each release. The scenarios comprised different types of changes involving mandatory and optional features. Table 2 also presents which types of change each release encompassed. The purpose of these changes is to exercise the implementation of optional and mandatory features to assess variability mechanisms properties in the context of software product line evolution.

Table 2. Summary of scenarios in WebStore

Release	Description	Type of Change	Extent of Change
R1	WebStore core		
R2	Two types of payment included (Paypal and BankSlip)	Inclusion of optional feature	No extensive modification because the features can be well localized.
R3	New feature included to manage category	Inclusion of optional feature	Required changes in components related to `Product` and insertions of new components related to `Category`.
R4	The management of category was changed to mandatory feature and new feature included to display products by category	Changing optional feature to mandatory and inclusion of optional feature	The inclusion of the new feature did not demand major modifications. Switching a feature from optional to mandatory required extensive removals in the DP.
R5	New feature included to display products by nearest day of inclusion	Inclusion of optional feature	Since this feature did not affect other functionalities, only minor changes and insertions were required.

In general, it's expected that evolution scenarios provide the increase of variability of the SPL. But in some cases this may not occur, as it did in release R4 of WebStore SPL. This kind of evolution was observed in other studies and has been classified as "New version of Infrastructure". In this case, this evolution scenario leads to a decrease of the functionality and this can be explained by the fact that some functionality have a tendency to move from the perimeter of a system towards the centre [48].

3.5 The Evolved MobileMedia SPL

The second target SPL was originally developed to serve as a benchmark for studies on aspect-oriented programming [24]. It was designed for academic purpose, but including diverse changes scenarios that could exercise its evolution.

MobileMedia [24] was developed based on a previous SPL, called MobilePhoto [53]. Table 3 provides some measures about the size of the SPL implementations in terms of number of components, number of methods and number of lines of source code (LOC). Classes and class refinements were accounted as components. LOC were accounted without considering blank lines. The average number of components varies from 22 (CC) to 141 (FOP). As occurred in WebStore, FOP requires more components to implement MobileMedia features. Moreover MobileMedia DP-based solution uses more lines of code than the FOP implementation, except in release 1. It is important to notice that DP-based solutions have a larger number of methods than other solutions. This can be explained by the fact that product configurations in DP-based solutions are done at runtime, using specific creational methods to permit variation point's instantiation. These methods are responsible to stack one or more feature decorator objects into a base object,

Table 3. MobileMedia SPL implementation

	CC							FOP							DP						
	R.1	R.2	R.3	R.4	R.5	R.6	R.7	R.1	R.2	R.3	R.4	R.5	R.6	R.7	R.1	R.2	R.3	R.4	R.5	R.6	R.7
#Comp.	22	23	23	28	35	44	49	54	63	73	86	106	127	141	34	49	55	74	86	108	135
#Meth.	113	132	135	153	191	227	267	143	177	191	216	285	331	368	132	191	209	275	337	417	518
LOC	971	1147	1214	1380	1852	2334	2926	1142	1356	1458	1629	2163	2498	2827	1064	1430	1544	1936	2440	2952	3682

Figure 2 presents a simplified view of the MobileMedia SPL feature model. Examples of core features are AlbumManagement and MediaManagement. In addition, some optional features are Favorite, Sorting, SMS Transfer and CopyMedia. Similar to Figure 1, numbers on the top right-hand corner of a feature in Figure 2, were used to indicate in which release the feature was included (see Table 4).

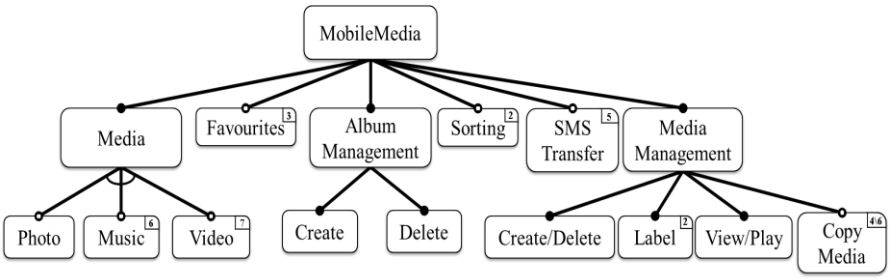

Figure 2. MobileMedia Basic Feature Model

3.6 Change Scenarios

Unlike WebStore, which was developed from scratch, we have a full CC implementation of MobileMedia available to us [24]. However, we had to design and implement the corresponding set of change scenarios in FOP and DP. Six change scenarios were considered in MobileMedia, resulting in seven releases. Table 4 summarizes changes of each release. The scenarios comprised different types of changes involving mandatory, optional and alternatives features. Table 4 also presents which types of change each release encompassed. The purpose of these changes is to exercise the implementation of optional, mandatory and alternative features to assess variability mechanisms properties in the context of software product line evolution.

Table 4. Summary of scenarios in MobileMedia

Release	Description	Type of Change	Extent of Change
R1	MobileMedia core.		
R2	New feature added to count the number of times a photo has been viewed and sorting photos by highest viewing frequency. New feature added to edit the photo's label.	Inclusion of optional and mandatory features	The feature `Sorting` required addition of new components and change components related to the use of this feature. For the feature `EditLabel`, a refactoring was conducted extracting a new `PhotoController` from the `BaseController`.
R3	New feature added to allow users to specify and view their favorite photos.	Inclusion of optional feature	The changes were narrowly localized.
R4	New feature added to allow users to keep multiple copies of photos.	Inclusion of optional feature	A major refactoring of `BaseController` was carried out producing four new specialized controllers.
R5	New feature added to send photo to other users by SMS.	Inclusion of optional feature	New controllers had to be included. New components related to SMS transfer had to be included. The `SMSTransfer` feature was designed as a specialization of the `CopyPhoto` feature.
R6	New feature added to play music. The photo management basic features were generalized to manage media and ViewPhoto was turned into an alternative feature.	Changing of one mandatory feature into two alternatives	A major refactoring of `PhotoController` and `PhotoListController` was carried out producing two new generic media controllers. New controllers related to music operations had to be included.
R7	New feature added to manage videos	Inclusion of alternative feature	New controllers related to video operations had to be included.

4 Change Propagation Analysis

This section presents a quantitative analysis to answer RQ1. In particular, we are interested to know how different variability mechanisms affect changes in software product line evolution.

Table 5. Summary of scenarios in MobileMedia

			WebStore Releases				Mobile Media Releases					
			R.2	R.3	R.4	R.5	R.2	R.3	R.4	R.5	R.6	R.7
Components	Added	CC	0	3	0	0	2	0	5	7	17	6
		FOP	4	6	2	4	10	10	23	21	74	14
		DP	10	9	8	6	15	6	20	13	79	29
	Removed	CC	0	0	0	0	1	0	0	0	8	1
		FOP	0	0	0	0	1	0	10	1	53	0
		DP	0	0	11	0	0	0	1	1	57	2
	Changed	CC	2	3	5	4	7	5	7	7	11	22
		FOP	1	1	0	0	10	6	23	10	28	13
		DP	4	4	4	1	13	11	29	11	11	27
Methods	Added	CC	1	26	0	3	22	3	37	38	103	47
		FOP	5	28	2	5	37	14	63	70	190	40
		DP	21	30	32	10	60	21	99	63	285	110
	Removed	CC	0	0	1	0	3	0	19	0	67	7
		FOP	0	0	0	0	3	0	38	1	144	3
		DP	1	0	34	0	1	3	33	1	205	9
	Changed	CC	2	2	6	2	9	7	10	7	26	30
		FOP	1	1	0	0	12	8	24	12	29	13
		DP	3	4	3	1	25	11	30	11	37	24
Lines of Code	Added	CC	15	148	7	14	197	67	538	478	1386	694
		FOP	35	160	14	28	243	102	490	551	1534	340
		DP	132	181	179	59	390	132	678	511	2189	820
	Removed	CC	0	3	0	0	21	0	372	6	904	102
		FOP	0	3	0	0	29	0	319	17	1199	11
		DP	0	1	192	0	24	18	286	7	1677	90
	Changed	CC	1	2	0	0	28	7	32	10	75	102
		FOP	1	2	0	0	21	10	83	8	62	19
		DP	9	2	3	0	45	13	85	12	75	46

4.1 Results

The quantitative analysis uses traditional measures of change impact [29, 52], considering different levels of granularity: components, methods, and lines of source code (Table 5). A general interpretation of these measures is that a lower number of modified and removed artifacts suggests a more stable solution, possibly supported by the variability mechanisms. In the case of additions of artifacts, we expect that it indicates the conformance with the Open-Closed principle. In this case, the lowest number of additions may suggest that the evolution is not being supported by non-intrusive extensions.

Figure 3. Additions in WebStore and MobileMedia

Figure 3 shows the relative values of added components, methods and lines of code in releases of both the WebStore SPL (left) and the MobileMedia SPL (right). In general, the CC mechanism presents lower number of added components and methods

in both subject systems compared to DP and FOP. This may be a result of how the insertions in CC have been carried out: by modifying existent components instead of creating new ones. The lower number of added components of CC is adherent with the practice for non-open-close systems that introduces changes directly in the existent components. The number of added components could be higher if, for example, we simply add conditional compilation directives around a method call that is declared by a new class. However, this solution, i.e. including more components artificially in a CC approach, would not be as usual as what programmers do in practice with annotative approaches, because we are considering that typically developers annotate *in loco* to introduce variations. Moreover, this alternative solution would artificially mislead the measures that are expected to represent the mechanisms provided by DP polymorphism and FOP extensions that enable the Open-Closed principle.

Figure 4. Modifications in WebStore and MobileMedia

Concerning MobileMedia, there is no sharp difference between the measures of the three mechanisms. The number of additions with DP is slighter greater than FOP that is slightly greater than CC. This behavior can be explained by the fact that product configurations in DP-based solutions are done at runtime, using specific creational classes and methods to permit variation point's implementation. For both SPL, there is a ratio of about two components using design patterns for each FOP refinement. In general, to implement a variation point in DP, it is necessary to implement decorator classes containing the additional behavior (similar to a FOP refinement) and another concerning the instantiation of the decorator classes.

On WebStore SPL, there is clearly higher number of components, methods and LOCs with DP than with FOP and CC. Since this LPS is smaller than MobileMedia SPL, i.e., it has fewer components, the presence of design pattern classes contribute to considerably increase the difference between the measures values. In release 4, this difference is even higher. This can be explained because the change of a feature from optional to mandatory caused several changes at design and architecture levels. These changes involved the removal of all classes responsible for implementing the optional feature and also the reinsertion of classes and methods to implement the new mandatory feature.

Figure 4 shows the relative values of changed components, methods and lines of code in all releases of both the WebStore SPL (left) and the MobileMedia SPL (right). The FOP mechanism has clearly a lower number of modified components and methods in the WebStore SPL compared to DP and CC. This was due to the simple nature of features implemented. In general, the number of components modifications in MobileMedia is in accordance with the variability mechanisms implementation. Both, DP and FOP have a greater number of components when compared to CC. Thus, it is expected that the number of components changes be proportional to the number of components. In release 4 of MobileMedia, the number of changed components is even lower in CC, because the respective versions in FOP and DP have been thoroughly refactored to support new features that would come in release 5. This can be verified in release 5 where changes were almost the same.

Figure 5 shows the relative values of removed components, methods and lines of code in releases 2 to 5 of both the WebStore SPL (left) and the MobileMedia SPL (right). In the WebStore SPL only in release 4 using DP had a significant difference, because the number of components, methods and lines removed were significant higher than in CC and FOP. This is because the feature change from optional to mandatory, resulting in removing the design pattern components that allowed enabling this feature. Considering release 4 in MobileMedia SPL, the number of removed components in FOP release was significantly higher than in DP and FOP. This can be explained because this version was restructured to better support the changes of release 5, where several class refinements needed to be removed to support this restructuring. This behavior was also observed in release 6 of MobileMedia, where the insertion of alternative features forced major restructuring.

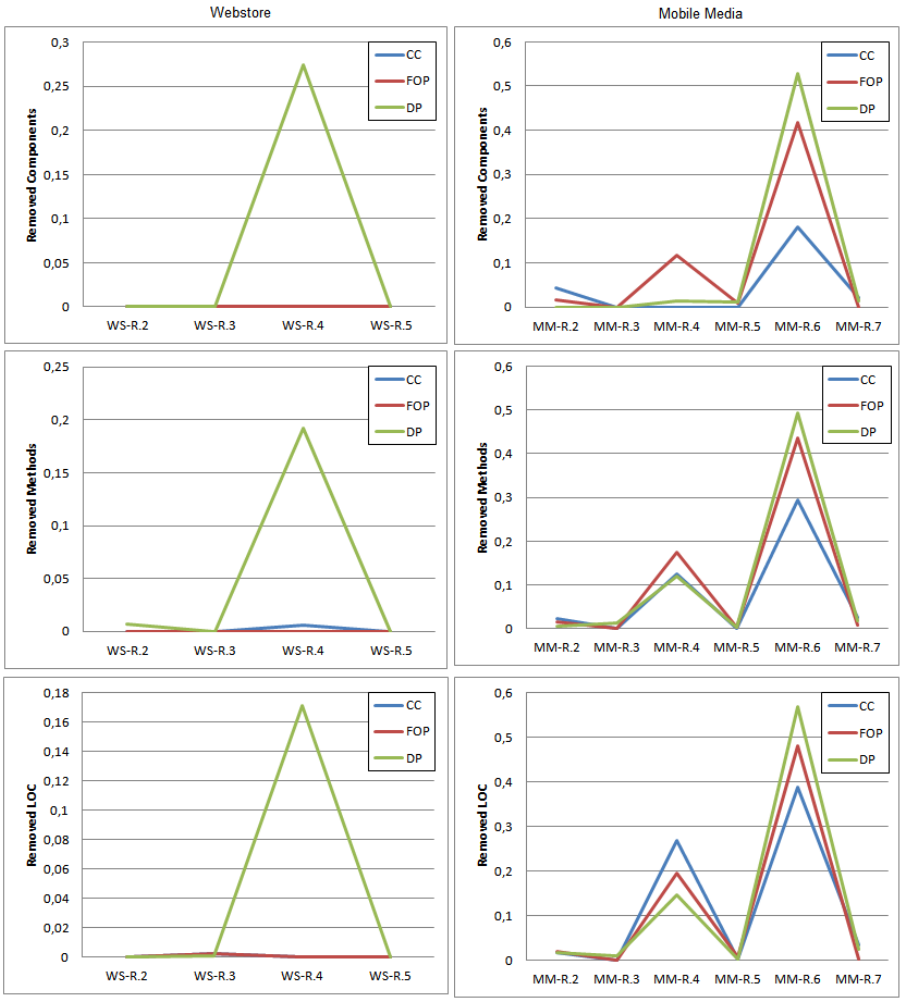

Figure 5. Removals in WebStore and MobileMedia

4.2 Discussion

Considering both systems and releases, the most significant difference noted in the change propagation is that CC releases have consistently lower number of added components than DP and FOP. Moreover, the results showed that FOP and DP strive to accommodate changes that require major features restructuring and usually demand a greater amount of component removals. Based on components insertions results, we suggest that CC does not adhere to the Open-Closed principle as FOP and DP adhere. Depending on how the additions were carried in CC, these values could be proportional to those presented by FOP and DP. However this would lead to a larger number of changes and removals in CC, breaking the compliance between the three

mechanisms. We could not observe a significant difference between FOP and DP mechanisms, because if in the WebStore, DP introduces more components than FOP, in MobileMedia, we have the inverse situation in three of four change scenarios.

5 Modularity Analysis

This section presents and discusses the results for the analysis of the stability of the SPLs design throughout the implemented changes. To support our analysis, we used a suite of metrics for quantifying feature modularity [47]. This suite measures the degree to which a single feature of the system maps to: (i) components (i.e. classes and class refinements) – based on the metric Concern Diffusion over Components (CDC), (ii) operations (i.e. methods) – based on the metric Concern Diffusion over Operations (CDO) and (iii) lines of code – based on the metrics Concern Diffusion over Lines of Code (CDLOC) and Lines of Concern Code (LOCC) [21]. We choose these metrics because they have been applied as benchmark in previous similar empirical studies concerning design modularity and stability [18, 23, 24, 47].

5.1 A Survey of Feature Modularity Metrics

The metrics presented in this section have a common characteristic that distinguishes them from traditional software metrics [23]. They capture information about the realization of features cutting across one or more components, i.e., these metrics are used for quantifying Separation of Concerns (SoC) [23, 47]. They can be applied to any kind of software artifact in either object-oriented or feature-oriented programs. Although these metrics were originally proposed to quantify concern properties, they can also be used to quantify features properties. The terms concern and feature are used without distinction in this study.

Sant'Anna et al. [47] defined three metrics that quantify scattering and tangling of features across a set of components, operations, and lines of code. The metrics Concern Diffusion over Components (CDC) and Concern Diffusion over Operations (CDO) quantify the degree of feature scattering at different levels of granularity – i.e., components and operations, respectively. The former counts the number of classes, interfaces and refinements that contribute to the implementation of a feature. The latter counts the number of methods and constructors realizing a feature. In addition to these two measures, the authors defined Concern Diffusion over Lines of Code (CDLOC) that computes the degree of feature tangling. For instance, given a certain feature F, this metric counts the number of "switches" between F and lines of code realizing other features [47]. A switch occurs when a code block realizing F is followed by a code block realizing another feature, and vice-versa. Besides Sant'Anna, other authors defined additional metrics to quantify properties of features. For instance, Eaddy and his colleagues [21] proposed a metric called Lines of Concern Code (LOCC). LOCC counts the total number of lines of code that contribute to the implementation of a feature. We adapted these metrics considering the ratio of the measured value to the total value on that release, for instance, CDC was calculated as the ratio of classes that contributes to the implementation of a feature to the total number of classes. In addition, our relative CDC represents the percentage of classes that are used to implement the feature. This relative metrics

enabled us to analyze together the set of metric values for all features. For all the employed metrics, a lower value implies a better result. Detailed discussions about the metrics appear elsewhere [21, 23, 26, 47].

5.2 Simple Analysis of the Modularity Metrics

This section presents and discusses the results for the metrics presented in Section 5.1. We analyzed 11 features from WebStore that include 4 optional and 7 mandatory features and 15 features from MobileMedia, 3 optional, 3 alternative and 9 mandatory. Optional and alternative features are the locus of variation in the SPLs and, therefore, they have to be well modularized. On the other hand, mandatory features also need to be investigated in order to assess the impact of changes on the core SPL architecture. From the analysis of the measures, interesting situations, discussed below, naturally emerged with respect to which type of modularization paradigm presents superior modularity and stability. The data was collected and organized in one sheet for each metric. For WebStore, each sheet has 4,442 lines, i.e., one line for each combination of feature, version, technique, and artifact. For MobileMedia, each sheet has 23,846 lines. Therefore, 113,152 points were measured in the whole study.

In this subsection, we present a simple analysis of the modularity metrics based on the metrics mean values for each version. Figure 6 presents CDC, CDO, CDLOC and LOCC mean values for each release of the WebStore SPL. The CDC mean values for FOP were consistently the lowest in all releases. The values for DP stayed in between FOP and CC. The CDLOC mean values for FOP were also consistently the lowest in all releases with stronger significant difference. However, for CDLOC, CC has presented lower values than DP, but the difference of the values tended to decrease in later releases, being almost the same in release 5. For CDO and LOCC there was no significant difference between releases or techniques.

Figure 7 presents CDC, CDO, CDLOC and LOCC mean values for each release of the MobileMedia SPL. The CDC values had similar behavior as those of WebStore. FOP values were consistently lower than DP values, which were consistently lower than CC values. For CDLOC mean values, differently from WebStore, there was no significant difference between FOP and DP, but CC was consistently greater than FOP and DP. Also, as occurred for WebStore, considering CDO and LOCC there was no significant difference between FOP, DP and CC. However, interestingly, for release 3, DP presented the lowest mean values of CDLOC, CDO and LOCC.

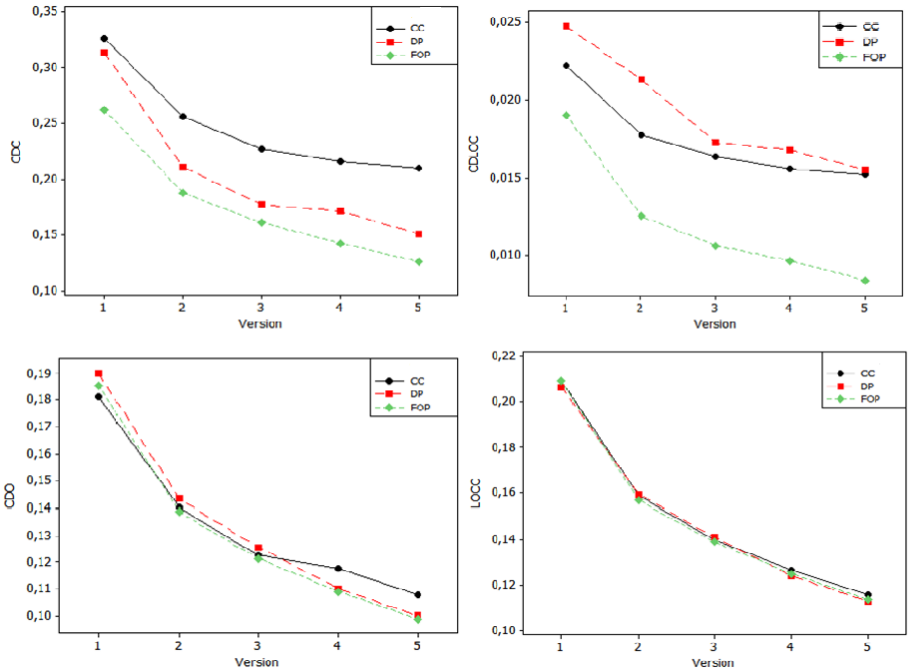

Figure 6. Metrics values through WebStore evolution

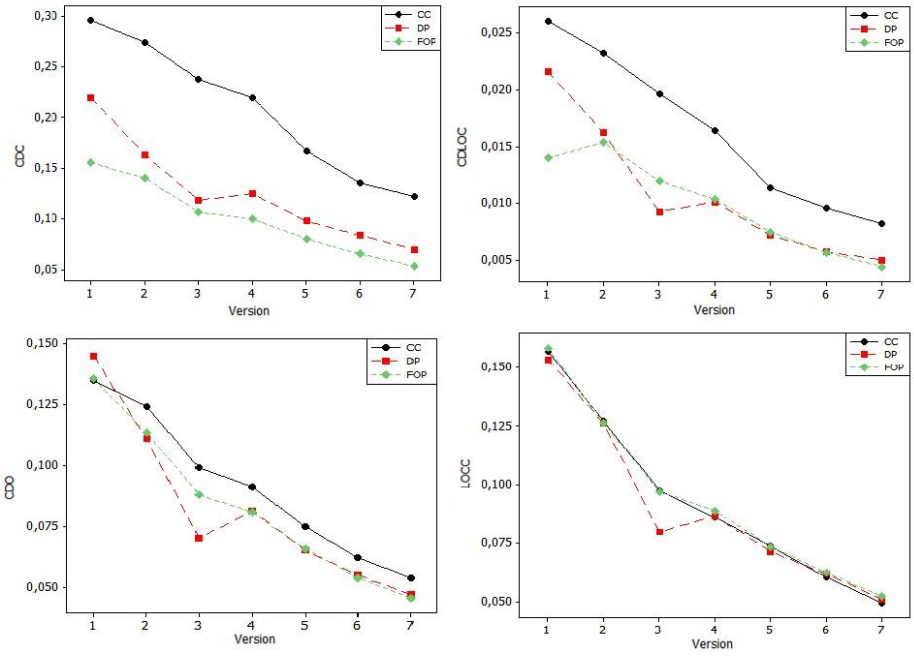

Figure 7. Metric values through MobileMedia evolution

5.3 Analysis of the Cumulative Distribution Function for the Modularity Metrics

In this subsection, we present more detailed analysis of the modularity metrics considering the dispersion of data. Our analysis is based on the empirical cumulative distribution functions of the data. The analyses were performed using Minitab 16©. The empirical cumulative distribution function (*ecdf*) can be used to evaluate the fit of a distribution to our data and to compare the different distributions of our sample. The stepped *ecdf* resembles a cumulative histogram without bars. The distribution that best fitted our data was 3-parameter Gamma. Our data definitely does not follow a normal distribution. Indeed, it does not follow a symmetric distribution. The data values are typically concentrated in smaller values. In order words, the median values for the metrics are generally smaller than the mean values.

The interpretation of the *ecdf* is done as follow: the higher is the area under the curve, the higher is frequency of lower values for the corresponding metrics. Considering that the lower are the values for feature modularity metrics, the better is the modularization, we consider that the best metrics curve is the one the presents the highest frequency of lower values.

Figures 8 and 9 show the empirical cumulative distribution function for the feature modularity metrics of WebStore and MobileMedia, respectively. One interesting point is that WebStore and MobileMedia, despite some differences, have presented an overall similar behavior, especially in CDC, CDO and LOCC. Concerning CDO, we can observe that FOP outperformed DP and CC in both systems, and DP outperformed CC. For CDLOC, we can observe that FOP clearly outperformed CC in both approaches, and clearly outperformed DP in WebStore. In MobileMedia, FOP just slightly outperformed DP. The fact is that DP had a performance similar to FOP in WebStore.

For CDO and LOCC, we could not see significant differences between the three approaches in both systems. Nonetheless, it is possible to see a slightly better performance for FOP in both systems.

In Figure 10, we can observe the tendency of the behavior of the metrics for each version of the system. We can see that, in general, the same global result previously presented can be observed in all versions. However, this version-based analysis shows that in the first versions, the CDC and CDLOC metrics have higher frequency of lower values for FOP. In general, we can observe that the higher is the version, the lower is the metrics values for all approaches and the lower is the difference between the approaches, but still discriminative in the case of CDC.

Figure 11 presents the same metric values from the feature point of view. We could see that independently from the used approaches, some features tend to produce a similar behavior. Some features have a remarkable worse behavior than all the others for all metrics, such as AlbumManagement (Black), PhotoManagement (Dashed Blue). They were followed by Base (Dashed Red), SMS Transfer (Dashed Green). These features are naturally complex. Concerning CDLOC, we can observe that besides the aforementioned features, all approaches had not good metric values for features Sorting (Blue Dashed-Dotted) and Favourites (Lilac Solid-Dotted).

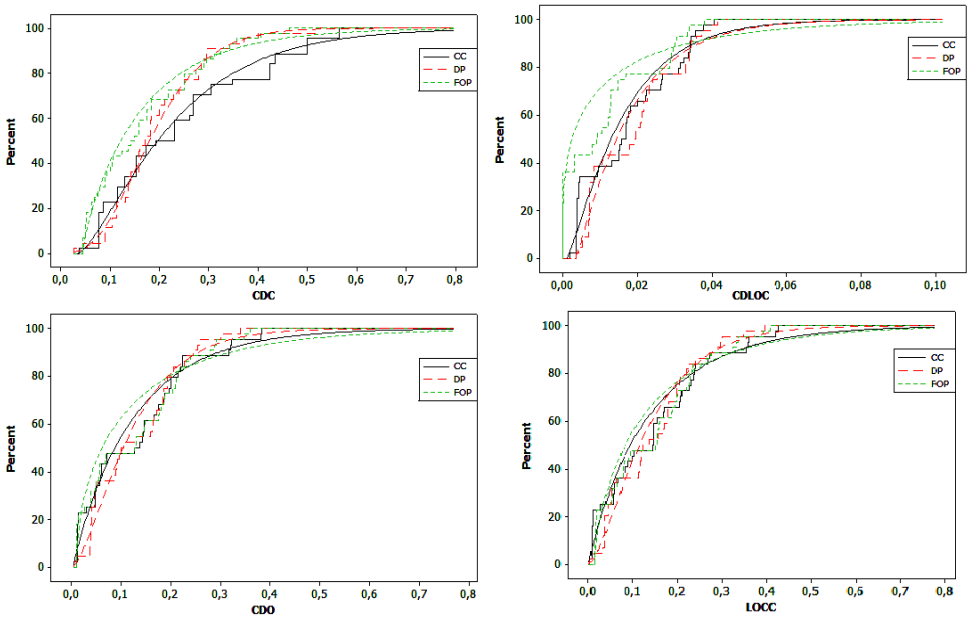

Figure 8. Empirical CDF for all versions of Webstore (3-parameter Gamma)

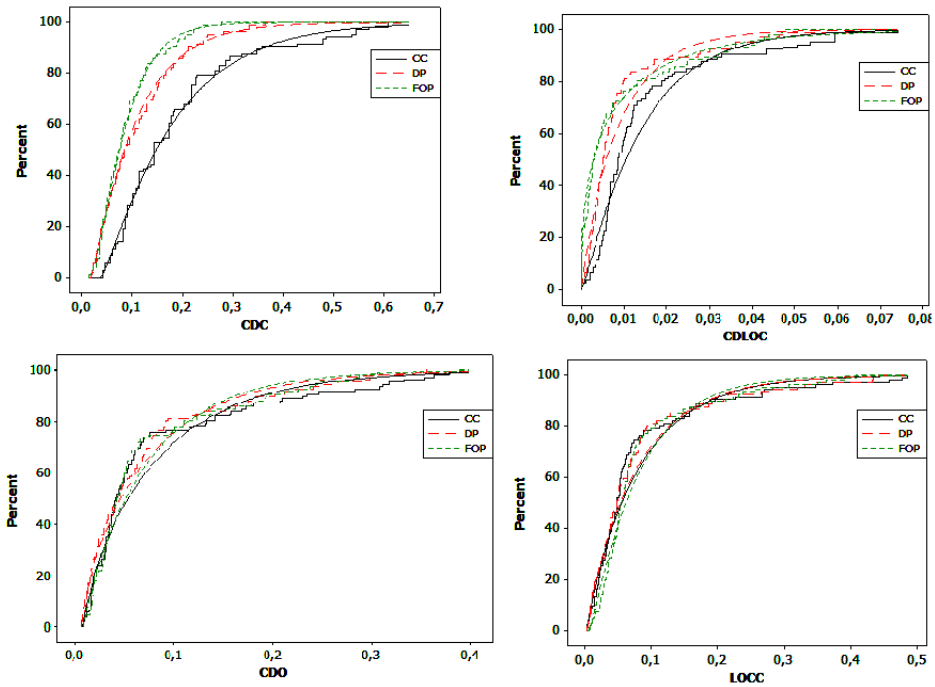

Figure 9. Empirical CDF for all versions of Mobile Media (3-parameter Gamma)

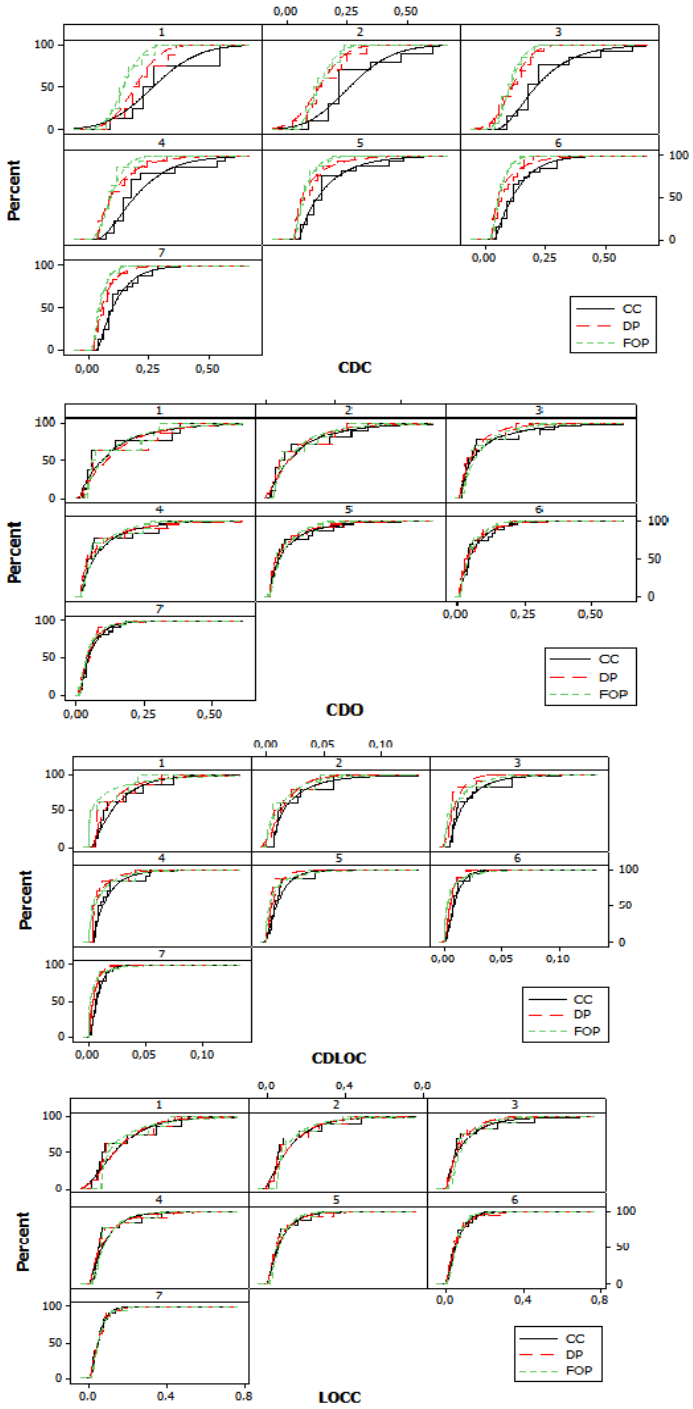

Figure 10. Empirical CDF per versions of MobileMedia (3-parameter Gamma)

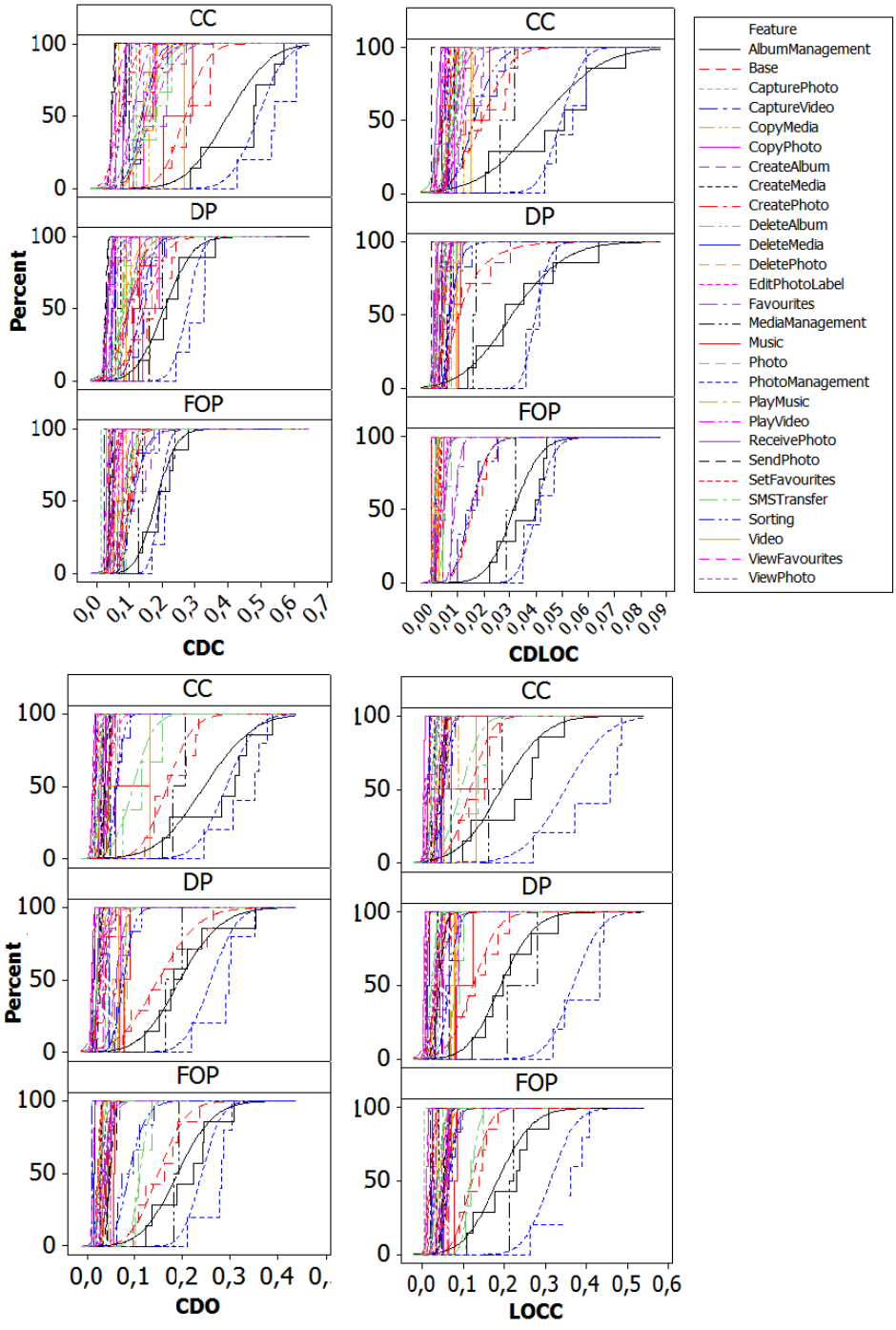

Figure 11. Empirical CDF per features of MobileMedia (3-parameter Gamma)

5.4 Discussion

FOP succeeds in features with no shared code. This situation was observed with six features of the MobileMedia SPL, namely, CreateAlbum, DeleteAlbum, CreatePhoto, DeletePhoto, EditPhotoLabel, and ViewPhoto. Some features with no shared code in WebStore SPL, namely, DisplayByCategory and DisplayWhatIsNew, produced similar results. The common characteristic of these features is that there is no source code sharing or overlapping, i.e., they do not share statements, methods or components with other features. The FOP solution presents lower values and superior modularity in terms of tangling (CDLOC) and scattering over components (CDC), which are supported by data in Figures 6 to 11. Figure 11, for instance, shows that the measured curves of these features are concentrated in lower values with FOP. The effectiveness of FOP mechanisms to modularize these features is due to the ability to move the code in charge of realizing the feature from large classes to a set of small cohesive class refinements. Conditional compilation lacks this ability because it has a somewhat intrusive effect on the code, due to the need of adding `#ifdef` and `#endif` clauses located at places where features crosscut. The results obtained from this quantitative analysis corroborate with the common knowledge about feature refinement mechanisms being more adequate to modularize features with no shared code. The analysis of the other scattering metrics (CDO and LOCC) did not follow the same trend of CDC, which can be explained with the fact that the granularity of the methods and lines of code is lower and the distribution of features occurs in a proportional fashion over all mechanisms. On the other hand, since the granularity of components is higher, the respective impact on modularity metrics is more observable.

When optional features are turned mandatory, DP removal may cause the SPL architecture destabilization. Another interesting situation that emerged in our analysis was the behavior of releases using the DP mechanisms on the transition from release 3 to release 4 of WebStore. For instance, while the FOP solution handles this particular situation without major issues, we observed the growth of the metrics in the DP implementation when an optional feature was turned mandatory, as observed in Figures 4 and 5. This problem can be explained by the fact that the implementation of an optional feature with DP requires a larger number of components compared to implement the same feature being mandatory. Therefore, developers have to carefully design flexible core architecture to allow the inclusion of mandatory features. If the patterns used to implement optional features are removed when the features become mandatory, then the architecture may degenerate and become unstable. An alternative solution would be keeping the features modularized in that patterns and make sure that the modules are always present in all products. However, this solution would not be fair to this specific change scenario since by turning an optional feature into mandatory, we should remove the components responsible for variation, i.e., the pattern implementation. If we keep pattern modules responsible for an obsolete variation point, it means that we are keeping needless code in the SPL, which could adversely affect future evolutions. For instance, the presence of these modules could turn program comprehension tasks more arduous. Moreover, keeping these DP would break the compliance between the SPL source code and feature model, since the SPL

source code would contain modules created to support the instantiation of an inexistent variation point.

Crosscutting features are problematic for all studied approaches. We could see from Figure 11 that the crosscutting features Sorting and Favourites were not well handled by the approaches as the majority of the other features. The reason is that the typical design to introduce these features intrinsically tangles and scatters their code. The code related to these features is highly tangled in some base components of MobileMedia, such as ImageData, MediaData, and MedialUtil. Due to this high coupling, these features are also scattered across the source code of other features. These components were minimally modularized and, thus, they are almost equally implemented with the three evaluated mechanisms. In these cases, the use of aspectual approaches would enhance modularity of these problematic features easing their code separation [7] [8].

Ratio-based analysis of metrics tends to be less discriminative in larger systems. The larger is the evaluated software version, the lower are the metrics ratios for all approaches and the lower is the observable difference between the approaches. Hence, we should consider that the size of the system can impact on the discriminative capability of the metrics to evaluate software modularity and stability. We performed our analysis based on the ratio of the measured values by the number of components. Since it is necessary to compare different mechanisms, we expect lower differences in metric values for larger systems due to the greater number of components. This situation occurs from the intrinsic nature of the studied metrics that evaluates scattering and tangling related to the whole system.

On the use of a single variability mechanism to construct SPLs. In practice, developers do not necessarily use only a single mechanism to address all kinds of features during SPL construction. They often combine two or more variability mechanisms depending on the kind of feature, feature location and granularity, quantification level [6, 34, 46]. Recent research shows that there is no silver bullet when it comes to mechanisms that manage variability in SPLs [6, 46]. We would introduce more independent variables in the study, for example, with the use of hybrid approaches. However, there is still lack of data and study about the strength of individual mechanisms. For this reason, we decided to study the approaches individually to identify their unique characteristics. For example, annotative approaches, like CC, are well known to support fine-grained extensions on statements, parameters, and conditional expressions [31, 34]. On other hand, certain fine-grained features are very hard, if not impractical, to implement with FOP. All these points considered, the analysis of individual mechanisms showed that, in general, FOP refinements provide more benefits related to modularity and changes propagation when compared to CC and DP. In order to draw more specific conclusions about the mechanisms, such as to propose programming guidelines to optimize their use, it is necessary to analyze them in more studies considering different domains, changes scenarios, and types of features.

6 Threats to Validity

Even with the careful planning of the study, some factors should be considered in the evaluation of the results validity. We discuss the study validity with respect to its conclusion, internal, external, and construct validity [51].

Concerning the conclusion validity, since 60264 data points were collected, the reliability of the measurement process might be an issue. This issue was alleviated because the measurements were independently checked by one of the authors that had not collected the respective data. Moreover, analysis may have been affected by spurious evidence since, for instance, modularity metrics were indirectly used to answer RQ1. In this particular case, we could only draw plausible conclusions since a stronger data analysis could not been carried out with such indirect measurement.

Concerning the internal validity, most analyzed versions of the SPLs were constructed by the authors for the purpose of this study. Different design options might have produced different results. WebStore was inspired by a previous Java application, named PetStore [16], developed based on industry-strength technology, such as Java Server Pages (JSP) and Servlets. Additionally, its successive releases were discussed between the developers in order carefully developed to employ the most widely used of each implementation technique. All CC releases of MobileMedia were designed and implemented in previous studies [24]. Therefore, in this case we only adapted the available releases to conform to the DP and FOP designs.

Another issue with respect to internal validity is that the modularity metrics depends on how accurate was the mapping (assignment) of each concern to code elements. Fortunately, we observed in a previous study [25] that, apart from Concern Diffusion over Lines of Code (CDLOC), the mapping process does not significantly impact the modularity metrics used in this paper. Additionally, in order to mitigate this threat, we relied on concern mappings produced by the original developers. Whether the concern mapping was fully correct or not, it just reflects how these metrics would be used in practice.

Concerning the external validity, some other factors limit the generalization of the results:

- Although the SPLs were carefully designed to be as much general as possible, it should be considered that WebStore and MobileMedia are special purpose systems that may not represent all properties of real world systems. However, both PetStore (predecessor of WebStore) and MobileMedia were used in research studies with similar purposes of ours [16, 24].
- The evolution scenarios may also not represent the large space of possibilities in real-world SPL evolution scenarios. For instance, we have not investigated some intricate situations involving feature interaction that may appear in larger SPLs.
- Only the Java programming language and the AHEAD environment were considered in this study. Some of our results could be different if other languages and environments, such as CaesarJ [43], were used. For example, different languages may support different types of constructs and the measures could have some variation.
- Only modularity and change propagation metrics were considered helpful to point out the variability mechanisms benefits. However, they provide only a

limited view of these benefits, as they do not measure the real effort required to perform SPL changes. Similar limitation is observed in every study that relies on metrics.

Finally, concerning the construct validity, one issue is on how much support change propagation and modularity metrics offer to produce robust answers to our investigation. As a matter of fact, these proxy metrics offer a limited view on the design stability and modularity problems, i.e., they only permit us to draw indirect conclusions about SPL modularity and stability properties. The modularity metrics are mostly related to separation of concerns properties, which are insufficient to allow a complete analysis of the benefits of each variability mechanism with respect to SPL modularity. Change propagation measures were used to complement the modularity analysis. In fact, we have learned in this study that these two sets of metrics should not be analyzed in isolation. However, they have shown themselves to be more useful when analyzed in conjunction with the other used metrics.

7 Related Work

Several studies have investigated variability management on SPLs [3, 4, 11, 49]. Batory and others have reported an increased flexibility in changes and significant reduction in program complexity measured by number of methods, lines of code, and number of tokens per class [11]. Simplification in evolving SPL architecture has also been reported in [38, 44], as consequence of variability management. Other research work has also analyzed stability and reuse of SPLs [18, 24]. For instance, Figueiredo and his colleagues [24] performed an empirical study to assess modularity, change propagation, and feature dependency of two evolving SPLs. Their results suggest that AOP copes well with the separation of features with no shared code and does not succeed when mandatory features are the change focus. Their study focused on aspect-oriented programming (AOP) while, in this study, we analyzed variability mechanisms available in feature-oriented programming (FOP).

Apel and Batory [8] have proposed the Aspectual Mixin Layers [7] approach to allow the integration between aspects and FOP refinements. These authors have also used size metrics to quantify the number of components and lines of code in an SPL implementation. Similar to ours, their study can be seen as a step towards the proper use of composition mechanisms available in these languages. Their study, however, (i) did not consider a significant suite of software metrics, such as change propagation metrics, and (ii) did not address SPL evolution scenarios and stability.

Dantas and his colleagues [18] conducted an exploratory study to analyze the support of new modularization techniques to implement SPLs. Their study aimed at comparing the advantage and drawbacks of different advanced programming techniques in terms of SPL feature stability and reuse. These authors have compared essentially three different AOP implementations using two evolving software product lines: iBatis and MobileMedia. Moreover, they conducted their study considering two additional stability metrics - Refactoring of Modules (RoM) and Alterations in Code Elements (ACE). Their work suggests that CaesarJ [43], a hybrid AOP and FOP approach, provides better stability and reuse of SPL modules. With respect to modularity, their quantitative analysis, based on the same suite of SoC metrics, showed that compositional approaches enable further modular decomposition of the

SPL code. Our work also supports this finding and presents new ones for the other studied mechanisms in the context of SPL evolution, as discussed in Section 5.4.

Kästner and others [34] performed a study to compare other important properties to be assessed when dealing with variability mechanisms for SPL: feature traceability, ease of adoption and safety. Their study compared compositional and annotative approaches, showing that each one has strengths and weaknesses. Their study supports the synergistic use of both approaches for best results in expressiveness, granularity and type-safety. Other studies also analyzed granularity and type-safety of variability mechanisms in the context of SPL [9, 33]. These studies complement our analysis since they investigate different SPL quality properties.

Several studies focused on challenges in the software evolution field [28, 39, 41]. These works have in common the concern about measuring different artifacts through software evolution, which relies directly on the use of reliable software metrics. For instance, Greenwood and others [29] used a similar suite of metrics to assess the design stability of an evolving application. In general, there is a shared sense about software metrics on the engineering perspective: they are far from being mature and are constantly the focus of disagreements [1, 32, 40]. Different from our study, Greenwood's one did not target at assessing the impact of changes in the core and variable features of SPLs. Additionally, they used a different application as a case study, named Health Watcher.

8 Concluding Remarks and Future Work

The use of variability mechanisms to develop SPLs largely depends on our ability to empirically understand its positive and negative effects through design changes. Generally speaking, the development of an SPL has to provide means to anticipate changes. That is why incremental development has been largely adopted. This study evolved SPLs in order to assess the capabilities of FOP mechanisms to provide SPL modularity and stability in the presence of change requests. Such evaluation included two complementary analyses: change propagation and feature modularity.

Our main contributions in this work were the development of an open benchmark for the evaluation of evolving SPLs, a qualitative and quantitative data analysis framework and an extensive data analysis of collected metrics using the benchmark and the framework.

Some interesting results emerged from our analysis. First, the FOP design of the studied SPLs tends to be more stable than the other traditional widely-used approaches. This advantage of FOP is particularly true when a change targets optional features. Second, we observed that FOP class refinements adhere more closely the Open-Closed principle [42]. Furthermore, such mechanisms usually scale well for dependencies that do not involve shared code.

The results of Sections 4 and 5 indicate that conditional compilation (CC) may not be adequate when used in evolving SPLs when feature modularity is a major concern. For instance, the addition of new features using CC mechanisms usually causes the increase of feature tangling and scattering. These crosscutting features destabilize the SPL architecture and make it difficult to accommodate future changes.

The implementations using design patterns and FOP refinements also strive to accommodate changes that require major restructuring. They usually require a higher

number of components insertions during this kind of SPL evolution, when compared to CC. The results have shown that the removal of some design patterns makes the SPL architecture unstable when optional features are turned into mandatory. This kind of change negatively affects the SPL modularity properties (especially scattering).

This work has revealed evidences for developers and language designers that although FOP is well-suited for SPL implementation, it still has drawbacks that require the combination with other mechanisms or the design of constructions to handle fine-grained, crosscutting and type-safe issues, respectively.

For the future work, the study of different metrics and its relationship to other quality attributes in SPLs, such as robustness and reuse could be interesting. In addition, other modularity properties, such as coupling and cohesion, could be assessed to increase the comprehensiveness of the results presented.

Also, aspects can be used symbiotically with one of the studied variability mechanism to develop SPLs. These hybrid approaches would permit us to better understand how they behave in change scenarios, especially because we have pointed out the crosscutting features are issues that none of studied mechanisms could provide successful solution (Figure 11).

Finally, a key challenge on the developing of SPLs is to guarantee that only well-typed programs are generated. It is often hard, if not impractical, to type check all possible products, especially when the number of feature combinations grows exponentially with the number of features. The annotative and compositional approaches studied in this paper do not support modular type checking. However, there are solutions based on SAT solvers [19, 36, 50] and type-checking non-preprocessed code [9, 35, 37] proposed to help this problem. Thus, future studies should analyze the ability of each approach to deal with this problem and increase the breadth of our study.

Acknowledgments

This work was partially supported by FAPEMIG, grant CEX-APQ-02932-10 and CEX-APQ-2086-11 and CNPq grant 475519/2012-4. This work was partially supported by CAPES and CNPq scholarships. We would like to thank the reviewer's comments that helped to improve the quality of this work.

References

1. Abran, A., Sellami, A., Suryn, W. Metrology.: Measurement and Metrics in Software Engineering. In Proceedings of the 9th International Software Metrics Symposium (Metrics), pp. 2—11. (2003).
2. Adams, B., De Meuter, W., Tromp, H., Hassan, A. E.: Can we Refactor Conditional Compilation into Aspects? In 8th ACM International Conference on Aspect-oriented Software Development (AOSD), pp. 243--254. ACM, Virginia, New York (2009)
3. Adler, C. Optional Composition - A Solution to the Optional Feature Problem? MSc Dissertation, University of Magdeburg, Germany. (2011).
4. Ali Babar, M., Chen, L., Shull, F. Managing Variability in Software Product Lines, IEEE Software,.27, pp. 89—91. (2010)

5. Alves, V., Neto, A. C., Soares, S., Santos, G., Calheiros, F., Nepomuceno, V., Pires, D., Leal, J., Borba., P.: From Conditional Compilation to Aspects: A Case Study in Software Product Lines Migration. In First Workshop on Aspect-Oriented Product Line Engineering (AOPLE), Portland, USA. (2006)
6. Anastasopoulos, M.: Implementing Product Line Variabilities. In Proceedings of the 2001 Symposium on Software Reusability, pp. 109—117. ACM (2001)
7. Apel, S. et al.: Aspectual Mixin Layers: Aspects and Features in Concert. In Proceedings of the 28th International Conference on Software Engineering, pp. 122--131, Shanghai, China. (2006)
8. Apel, S., Batory, D.: When to Use Features and Aspects? A Case Study. In Proceedings of the 5th International Conference on Generative Programming and Component Engineering, pp. 59--68. Portland, Oregon (2006)
9. Apel, S., Kästner, C., Größlinger, A. and Lengauer, C.: Type Safety for Feature-Oriented Product Lines. Automated Software Engineering, pp. 251--300. (2010)
10. Apel, S., Leich, T., Saake, G.: Aspectual feature modules. IEEE Transactions on Software Engineering., Volume 34, pp.162--180. (2008)
11. Batory, D., Johnson, C., MacDonald, B., Heeder, D. V.: Achieving Extensibility through Product-Lines and Domain-Specific Languages: A Case Study, ACM Transactions on Software Engineering and Methodology, Volume 11, pp. 191--214. (2002)
12. Batory, D., Sarvela, J., Rauschmayer.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering, Volume 30, Issue 6, pp. 355--371. (2004)
13. Batory, D.: Feature models, Grammars, and Propositional Formulas, In Proceedings of the 9th International Software Product Line Conference (SPLC), pp. 7--20. (2005)
14. Batory, D.: Feature-Oriented Programming and the AHEAD Tool Suite. In Proceedings of the 26th International Conference on Software Engineering, ICSE'04, pp. 702--703. IEEE Computer Society, Washington, (2004)
15. Cardelli, L., Wegner, P.: On understanding Types, Data Abstraction, and Polymorphism. Computing Surveys, 17 (4): pp. 471--522. (1985)
16. Castor Filho, F., Cacho, N., Figueiredo, E., Maranhao, R., Garcia, A., Rubira, C.: Exceptions and Aspects: The Devil is in the Details. In Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), pp. 152--162. Portland, USA, (2006).
17. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, (2002)
18. Dantas, F., Garcia, A.: Software Reuse versus Stability: Evaluating Advanced Programming Techniques. In: 23th Brazilian Symposium on Software Engineering, SBES'10, pp. 40--49, Salvador, Bahia, Brazil, (2010)
19. Delaware, B., Cook,W., Batory, D.: Fitting the Pieces Together: A Machine-checked Model of Safe Composition. In Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pp. 243—252, ACM, New York, NY, USA, (2009)
20. Eaddy, M., Aho, A., Murphy, G.C.: "Identifying, Assigning, and Quantifying Crosscutting Concerns". In Proceedings of the 1st International Workshop on Assessment of Contemporary Modularization Techniques, pp. 2, (2007)
21. Eaddy, M.: An Empirical Assessment of the Crosscutting Concern Problem. Ph.D. Dissertation. Columbia University. (2008).
22. Ferreira, G., Gaia, F., Figueiredo, E., and Maia, M.: On the Use of Feature-Oriented Programming for Evolving Software Product Lines: A Comparative Study. In

Proceedings of the 15th Brazilian Symposium on Programming Languages (SBLP), pp. 29--30. Sao Paulo, Brazil, (2011)
23. Figueiredo, E. et al.: On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In Proceedings of European Conference on Software. Maintenance and Reengineering, pp. 183--192, Athens (2008)
24. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., and Dantas, F.: Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: 30th International Conference on Software Engineering, pp. 261--270. ACM, New York (2008)
25. Figueiredo, E., Garcia, A., Maia, M., Ferreira, G., Nunes, C., Whittle, J. On the Impact of Crosscutting Concern Projection on Code Measurement. In Proceedings of the Int'l Conference on Aspect-Oriented Software Development (AOSD), (2011)
26. Figueiredo, E., Sant'Anna, C., Garcia, A. and Lucena, C.: Applying and Evaluating Concern-Sensitive Design Heuristics. In: Proceedings of the 23rd Brazilian Symposium on Software Engineering (SBES), pp. 83--93, Fortaleza, Brazil (2009)
27. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison Wesley (1995)
28. Godfrey, M., German, D.: The Past, Present, and Future of Software Evolution, In Frontiers of Software Maintenance, pp. 129--138 (2008)
29. Greenwood, P. et al.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In Proceedings of the 21st European conference on Object-Oriented Programming (ECOOP), pp. 176—200. Berlin (2007)
30. Grubb, P., Takang, A. A.: Software Maintenance: Concepts and Practice. World Scientific Publishing Company, New Jersey (2003)
31. Hu, Y., Merlo, E., Dagenais, M. and Lague, B.: C/C++ Conditional Compilation Analysis Using Symbolic Execution. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM), pp. 196, (2000)
32. Jones, C.: Software metrics: Good, Bad and Missing, Computer, Volume 27, Issue 9, pp. 98--100 (1994)
33. Kästner, C., Apel, S. and Kuhlemann, M. Granularity in Software Product Lines. In Proceedings of the 30th ICSE'08, pp. 311--320, New York, NY, USA, ACM (2008)
34. Kästner, C., Apel, S., Integrating Compositional and Annotative Approaches for Product Line Engineering. In: Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE). pp. 35--40. (2008)
35. Kästner, C., Apel, S., Thüm, T., Saake, G.: Type checking Annotation-based Product Lines. Transactions on Software Engineering and Methodology, Volume 21, Issue 3, Article 14 (2012).
36. Kästner, C., Apel, S.: Type-Checking Software Product Lines - A Formal Approach. In Proceedings of the International Conference on Automated Software Engineering (ASE), pages 258–267. IEEE CS, (2008).
37. Kenner, A., Kästner, C., Haase, S., Leich, T.:. TypeChef: Toward Type Checking #ifdef Variability in C. In Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, ACM, New York, NY, USA, pp. 25--32. (2010)
38. Lee, K., Kang, K. C., Koh, E., Chae, W., Bokyoung, K., Choi, B. W. Domain-oriented Engineering of Elevator Control Software: A Product Line Practice, in: Proceedings of the First Conference on Software Product Lines: Experience and Research Directions, pp. 3--22. Kluwer Academic Publishers, (2000)

39. Maletic, J., Kagdi, H. Expressiveness and Effectiveness of Program Comprehension: Thoughts on Future Research Directions, In Frontiers of Software Maintenance, pp. 31-- 37. (2008).
40. Mayer, T., Hall, T.: A Critical Analysis of Current OO Design Metrics, Software Quality Control, Volume 8, pp. 97—110, (1999) .
41. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfield, R., Jazayeri, M. Challenges in Software Evolution, in: IWPSE'05 : Proceedings of the Eighth International Workshop on Principles of Software Evolution, IEEE Computer Society, pp. 13--22, (2005)
42. Meyer, B.: Object-Oriented Software Construction, 1st ed. Prentice-Hall, Englewood Cliffs (1988)
43. Mezini, M., Ostermann, K. Conquering Aspects with Caesar. In 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA, (2003)
44. Pettersson, U., Jarzabek, S. Industrial Experience with Building a Web Portal Product Line Using a Lightweight, Reactive Approach. In Proceedings of the 10th European Software Engineering Conference, pp. 326–335. ACM, (2005)
45. Prehofer, C. Feature-oriented Programming: A Fresh Look at Objects. ECOOP 1997: pp. 419–443. (1997)
46. Ribeiro, M., Borba, P.: Improving Guidance when Restructuring Variabilities in Software Product Lines. In Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR), pp. 79--88, Kaiserslautern, (2009)
47. Sant'Anna, C. et al.: On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In Brazilian Symposium on Software Engineering (SBES), pp. 19--34 (2003)
48. Svahnberg, M., Bosch. J.: Evolution in Software Product Lines: Two cases. Journal of Software Maintenance, Volume 11, Issue 6, pp. 391--422, New York, NY, USA. (1999)
49. Svahnberg, M., Gurp, J.v., Bosch, J.: A Taxonomy of Variability Realization Techniques, Software--Practice and Experience, Volume 35, Issue 8, pp. 705--754. (2005)
50. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe Composition of Product Lines. In Proceedings of the 6th International Conference on Generative Programming and Component Engineering, GPCE, ACM, New York, NY, USA, pp. 95--104 (2007)
51. Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., Wesslen, A.: Experimentation in Software Engineering. Springer. (2012)
52. Yau, S. S. and Collofello, J. S.: Design Stability Measures for Software Maintenance. IEEE Transactions on Software Engineering, Volume 11, Issue 9, p. 849--856, (1985)
53. Young, T.: Using AspectJ to Build a Software Product Line for Mobile Devices. MSc Dissertation, University of British Columbia, (2005)