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 Univesity of Uberlândia, Brazil
[2] Department of Computer Science, Federal Univesity 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 the use of feature-oriented programming is better 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 may 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 an evolving SPL. The quantitative data is collected from an SPL developed using three different variability mechanisms: FOP refinements, conditional compilation, and object-oriented design patterns. Our results suggest that FOP requires fewer changes in source code, yet a higher number of added modules, than the other techniques. It provides better support 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.

Keywords: Software product lines, feature-oriented programming, variability mechanisms.

1 Introduction

Software Product Lines (SPLs) [9] 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 foster reuse of mandatory and optional features in several SPL products. Optional features define points of variability and their role is to permit the instantiation of different products by enabling or disabling specific SPL features.

As in any software life cycle, changes are expected and must be accommodated [18]. When it comes to SPLs, these changes are even more frequent, since it is expected that a software product line evolves to attend new stakeholder requests [9].

Variability mechanisms are very important when considering SPL in permanent evolution. To be considered effective, they must guarantee the architecture stability and, at the same time, facilitate future changes. In order to ensure these requirements, the variability mechanisms should not degenerate modularity and should minimize changes. This can be reached through non-intrusive and self-contained changes that favor insertions and do not require deep modifications in existent components. The inefficacy of variability mechanisms to accommodate changes might lead to several undesirable consequences related to the product line stability, including invasive wide changes, significant ripple effects, artificial dependences between core and optional features, and unplugability of the optional code [20].

Our work targets to find out what variability mechanisms best adhere to mentioned SPL evolution practices. In this context, this paper presents a case study that evaluates comparatively three mechanisms for implementing variability on evolving product lines: conditional compilation (CC), object-oriented design patterns (DP) and feature-oriented programming (FOP). Our investigation focused on the evolution of five versions of a software product line (Section 3), called WebStore.

In Section 2, the implementation mechanisms used in the case study are presented. Section 3 presents the study setting, including the target SPL and the change scenarios. Section 4 analyzes changes which propagate through releases. In Section 5, the modularity of WebStore SPL is discussed. Section 6 presents some related work and points out directions for future work. Finally, Section 7 concludes this paper.

2 Variability Mechanisms for Software Product Lines

This section presents some concepts about the three techniques evaluated in the study: conditional compilation (CC), object-oriented design patterns (DP) and feature-oriented programming (FOP). Our main goal is to evaluate the composition mechanisms available in FOP using the other two variability techniques as baseline. We choose conditional compilation and design patterns because these are the state-of-the-practice options adopted in SPL industry [2, 21].

2.1 Conditional Compilation

The CC approach used in this work is a well-known technique for handling software variability [1, 2]. It has been used in programming languages like C for decades and it is also available in object-oriented languages such as C++ [19] and Java [2]. Basically, the 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. A directive `//#if defined(Paypal)` in line 5 indicates the beginning of code belonging to Paypal feature. The `#endif` directive 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. This value indicates

the presence of the feature in the 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

Object-oriented design patterns became widely used with the Gang of Four book [15]. Design patterns rely on object-oriented mechanisms, such as dynamic binding and polymorphism [8]. The example in Listings 2, 3 and 4 shows a class implemented using the Decorator design pattern. The purpose of this decoration is to add a behavior in a pluggable way. Line 7 in Listing 4 presents the method “init” which decorates the “init” method from the first code snippet. The decoration is made by dynamic binding mechanisms and the target class will hold a list containing both actions: “goToHome” and “goToPaypal”.

```
1  public class ControllerMapper
                      implements ControllerActionSelector {
2    protected Map actions = new HashMap();
3
4    public ControllerMapper() {
5      init();
6    }

7    public void addAction(...) {
8      actions.put(actionName, action);
9    }

10   public void init() {
11     addAction("goToHome", new GoToAction("home.jsp"));
12   }
13
14   public ControllerAction getAction(String actionName) {
15    return (ControllerAction) (actions.containsKey(actionName)
            ? actions.get(actionName) : null);
16   }
17 }
```

Listing 2. XXX.

```
1  abstract class ControllerDecorator
                 implements ControllerActionSelector {
2    protected IControllerActionSelector mapper;
3    protected Map controllerMap = new HashMap();
4
5    public ControllerDecorator(IControllerActionSelector m) {
6      this.mapper = m;
7      init();
8    }
9
10   public abstract void init();
11
12   public void addAction(String an, ControllerAction ca) {
13     controllerMap.put(an, ca);
14   }
15
16   public ControllerAction getAction(String an) {
17     return controllerMap.containsKey(an) ?
              controllerMap.get(an) : mapper.getAction(an));
18   }
19 }
```

Listing 3. XXX.

```
1   public class PaypalControllerMapperDecorator
                   extends ControllerMapperDecorator {
2
3    public PaypalControllerMapperDecorator(...) {
4      super(target);
5    }
6
7    public void init() {
8      addAction("goToPaypal", new GoToAction("paypal.jsp"));
9    }
10 }
```

Listing 4. XXX.

2.3 Feature-Oriented Programming

Feature oriented programming (FOP) [24] is a paradigm for modularizing software, considering features as a major element. AHEAD is an approach to support FOP based on step-wise refinements [6, 7]. The main idea behind AHEAD is that programs are constants and features are added to programs using refinement functions. The code snippets in Listings 5 and 6 show examples of a class and a class refinement piece of code used to implement variation points.

```
1  public class ProcessCheckoutFormAction {
2     ...
3    private ControllerAction selectPayment ...) {
4      if (paymentType.equals("Default")) {
5        paymentAction = new GoToAction("payment.jsp");
6      }
7      return paymentAction;
8    }
9  }
```

Listing 5. XXX.

```
1  layer paypal;
2
3  refines class ProcessCheckoutFormAction {
4
5     private ControllerAction selectPayment(...) {
6       Super(ControllerAction, String).selectPayment(...);
7
8          if (paymentType.equals("Paypal")) {
9           paymentAction = new GoToAction("paypal.jsp");
10       }
11    }
12 }
```

Listing 6. XXX.

The example in Listing 5 shows an ordinary base class and Listing 6 presents the FOP class refinement. In line 1 of the second code snippet there is a clause that indicates the layer that encompasses the class refinements. The "paypal" identifier 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 refinement classes from other features included.

3 Steps and Study Settings

This study was divided in four phases: (1) design and implementation of SPL and change scenarios using the three techniques, (2) data collection from the source code, (3) metrics calculation and (4) qualitative and quantitative analysis of the results. In the first phase, two graduate students implemented the WebStore SPL using three variability mechanisms and the changes to evolve it. All code was marked according to each feature in the second phase. This step was done so that the developers of a version do not mark their own produced code. In the third phase, we collected changes propagation and the modularity metrics. Finally, the metrics were analyzed in the fourth phase. Section 3.1 presents the analyzed SLP and Section 3.2 discusses its change scenarios.

3.1 The Evolved WebStore SPL

The target SPL was developed to represent major features of an interactive web store. It was designed for academic purpose, but focusing on the key features available in real web store systems. We have also designed representative changes scenarios (similar in all studied implementations) that could exercise its 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 average size of the SPL implementations in terms of number of components and number of lines of source code (LOC). Classes, interfaces and class refinements were

counted as components. The average number of components varies from 25 (CC) to 50 (FOP). Although FOP requires more components to implement WebStore, the DP-based solution uses about 20% more lines of code than the other implementation options.

Table 1. Webstore SPL implemetations

	CC	FOP	DP
Number of Components (avg)	25	50	45
LOC (aprox.)	1150	1250	1400

Figure 1 presents a simplified view of the WebStore SPL feature model. 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).

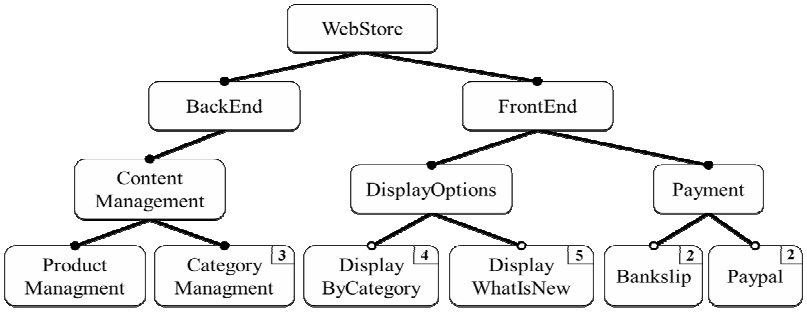

Figure 1. WebStore Feature Model

The WebStore versions that use FOP and CC mechanisms are very similar from the design point-of-view. The FOP versions were developed trying to maximize the division of features code, through base class and class refinements. In the CC versions, we developed and then marked all optional features in source code. These marks were made through `#ifdef` and `#endif` directives (Section 2.1).

The WebStore version that uses object-oriented design patterns was implemented mainly based in two design patterns: Decorator and Abstract Factory [15]. Their roles are to mimic the FOP mechanisms, namely easy feature code additions and different product instantiations, respectively.

3.2 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, which led to 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 the feature boundaries and, so, to assess the design stability of the product line.

Table 2. Summary of scenarios in WebStore

Release	Description	Type of Change
R1	WebStore core	
R2	Two types of payment included (Paypal and BankSlip)	Inclusion of optional feature
R3	New feature included to manage category	Inclusion of optional feature
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
R5	New feature included to display products by nearest day of inclusion	Inclusion of optional feature

4 Change Propagation Analysis

This section presents a quantitative analysis of how different variability mechanisms affect changes in software product line evolution. The quantitative analysis uses traditional measures of change impact [17, 26], considering different levels of granularity: components, methods, and lines of source code (Table 3). The lower the change impact measures the more stable the variability mechanisms of a solution are.

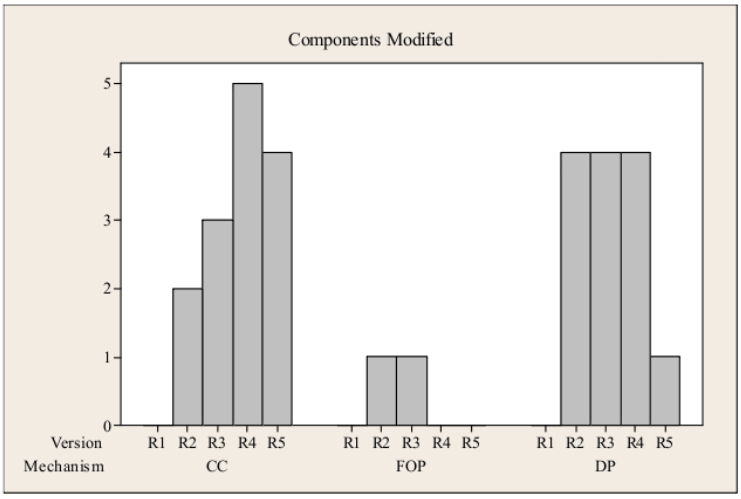

Figure 2. Components modified in the WebStore evolution

When analyzing changes impact measures, we noticed that the FOP implementation requires fewer modifications to components than the other solutions in all versions (Figure 2). The same behavior was also observed when methods were analyzed. During the entire evolution (release R1 excluded) of the target SPL using FOP mechanisms, just two components were modified and sixteen components added (in the majority through class refinements) as observed in Figures 2 and 3. This indicates that the FOP solution conforms more closely to the Open-Closed principle [21] which states that "software should be open for extension, but closed for modification".

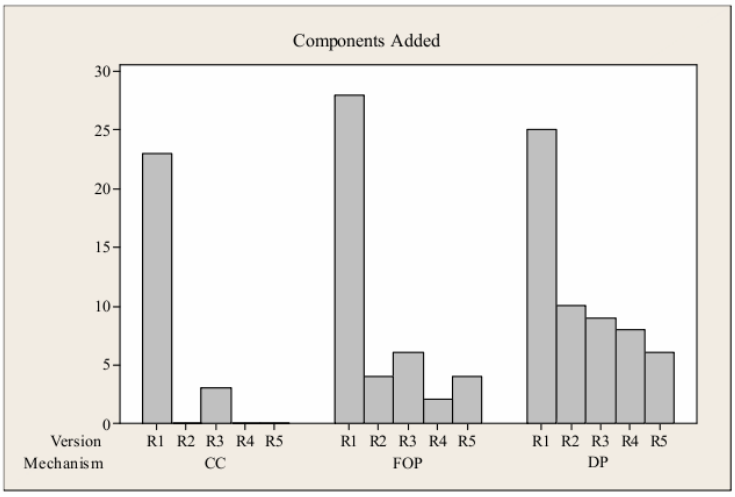

Figure 3. Components added in the WebStore evolution

Another point to be considered is the pattern established when additions are compared to modifications (Figures 2 and 3). The solution that uses Conditional Compilation mechanisms has fewer cases of additions of components and methods, but it is the one that requires more modifications over all releases. Excluding release R1 and taking the four subsequent change scenarios of the CC solution into account, 14 components were modified and just three components added. That is, unlike FOP, the CC implementation does not adhere to the Open-Closed principle at all. The variability solution implemented by DP sits between FOP and CC with respect to how close it adhere the Open-Closed principle. In fact, the DP implementation presents the worst results considering both the number of modifications and the number of additions to evolve the target SPL.

Besides requiring many components to be added and modified, both DP and CC solutions also need a large amount of methods additions as presented in Table 3. This behavior can be justified by the rigid structure these variability mechanisms impose. This structure consists primarily of classes that implement the Abstract Factory and

Decorator design patterns. These patterns aim at guaranteeing different product instantiations and inserting new behavior (feature) to a class in a less intrusive way.

Table 3. Measures of change propagation in WebStore

			Releases			
			R.2	**R.3**	**R.4**	**R.5**
Components	Added	CC	0	3	0	0
		FOP	4	6	2	4
		DP	10	9	8	6
	Removed	CC	0	0	0	0
		FOP	0	0	0	0
		DP	0	0	11	0
	Changed	CC	2	3	5	4
		FOP	1	1	0	0
		DP	4	4	4	1
Methods	Added	CC	1	26	0	3
		FOP	5	28	2	5
		DP	21	30	32	10
	Removed	CC	0	0	1	0
		FOP	0	0	0	0
		DP	1	0	34	0
	Changed	CC	2	2	6	2
		FOP	1	1	0	0
		DP	3	4	3	1
Lines of Code	Added	CC	15	148	7	14
		FOP	35	160	14	28
		DP	132	181	179	59
	Removed	CC	0	3	0	0
		FOP	0	3	0	0
		DP	0	1	192	0
	Changed	CC	1	2	0	0
		FOP	1	1	1	1
		DP	9	2	3	0

5 Modularity Analysis

This section presents and discusses the results for the analysis of the stability of the WebStore SPL throughout the implemented changes. To support our analysis, we used a suite of metrics for quantifying feature modularity [25]. 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), (iii) attributes – based on the metric Number Of Concern Attributes (NOCA) [14] and (iv) lines of code – based on the metrics Concern Diffusion over Lines of Code (CDLOC) and Number Of Lines of Concern Code (LOCC) [11]. We choose these metrics because they have been used and validated in several previous empirical studies [10, 12, 13, 25].

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 [13]. They capture information about the realization of features cutting across one or more components. 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, the terms concern and feature are used without distinction in this study. Therefore, we use these metrics to quantify properties of features in the WebStore SPL.

Metrics for Feature Diffusion. Sant'Anna et al. [25] 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 and interfaces that contributes 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 [25]. A switch occurs when a code block realizing F is followed by a code block realizing another feature, and vice-versa.

Other Metrics: Besides Sant'Anna, other authors defined additional metrics to quantify properties of features. For instance, Eaddy and his colleagues [11] 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. Figueiredo et al. [14] defined a different metric called Number of Concern Attributes (NOCA). This metric counts the number of attributes realizing a feature in the software system. For all the employed metrics, a lower value implies a better result. Detailed discussions about the metrics appear elsewhere [11, 13, 14, 25].

5.2 Feature Modularity

This section presents and discusses the measurement results for the metrics presented in Section 5.1. We analyzed 11 features from WebStore which include 4 optional and 7 mandatory features. They were selected because optional features are the locus of

variation in the SPLs and, therefore, they have to be well modularized. On the other hand, mandatory features need to be investigated in order to assess the impact of changes on the core SPL architecture. From the analysis of the measures, two interesting situations, discussed below, naturally emerged with respect to which type of modularization paradigm presents superior modularity and stability.

FOP succeeds in features with no shared code. This situation was observed with three optional features (Bankslip, Paypal, and DisplayWhatIsNew). For these features the FOP solution presents lower values and superior stability in terms of tangling (CDLOC) and scattering over components (CDC) (Figures 4 and 5). The effectiveness of FOP mechanisms to localize this kind of features is due to the ability to permit refinements of a base behavior to be composed in a plug-and-play fashion. This is done using insertions of small class refinements. Conditional compilation lacks this ability because it has a somewhat intrusive effect on the code, due to the need to add the #ifdef /#endif clauses located at places where features crosscut. The results of the other metrics (CDO, NOCA and LOCC) followed the same trend of the CDC metric.

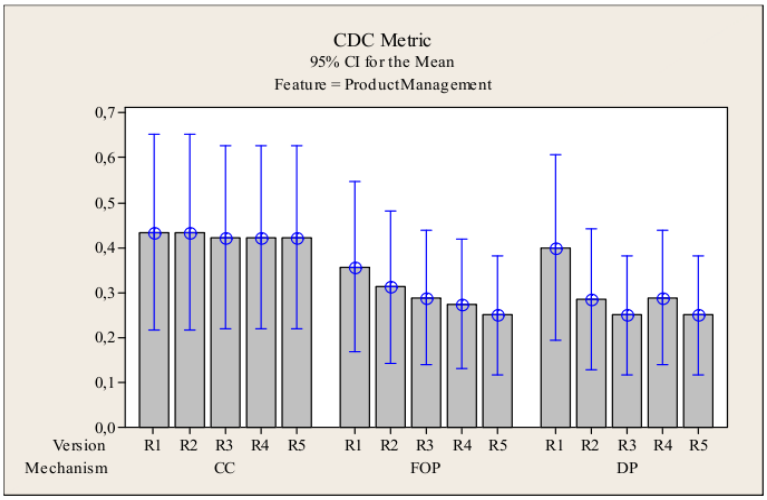

Figure 4. CDC metric values through WebStore evolution

The elimination of DP makes the SPL architecture unstable when optional features are turned mandatory. 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. 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. This situation results in increased scattering and tangling of the feature as observed in Figures 4 and 5. This problem can be explained by the fact that the implementation of an optional feature requires a larger number of components compared to implement the same feature being mandatory. Therefore, developers have to carefully design a flexible core

architecture to allow the inclusion of additional mandatory features. If the patterns used to implement optional features are removed when the features become mandatory, then the architecture may degenerate and becomes unstable.

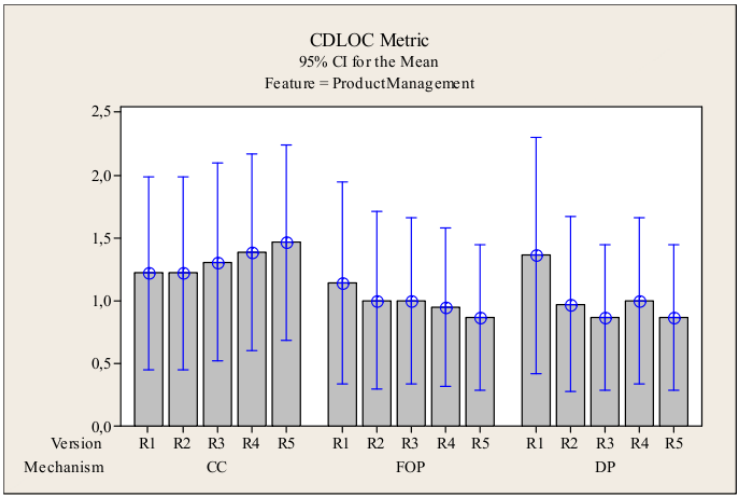

Figure 5. CDLOC metric values through WebStore evolution

6 Related Work and Study Constraints

Recent research work has analyzed stability and reuse of SPLs [10, 12]. Figueiredo et al. [12] performed an empirical study to assess modularity, change propagation, and feature dependency of two evolving SPLs. Their study focused on aspect-oriented programming while we analyzed variability mechanisms available in feature-oriented programming in this study. Dantas and his colleagues [10] conducted an exploratory study to analyze the support of new modularization techniques to implement SPLs. However, their study aimed at comparing the advantage and drawbacks of different techniques in terms of stability and reuse. Although Dantas also used a FOP language, named CaesarJ [23], we focused on different goals and on a different language: AHEAD [6]. Other studies also analyzed the variability management of SPLs and benefits of using FOP in software reuse [5, 22].

Apel and Batory [3] have proposed the Aspectual Mixin Layers [4] approach to allow the integration between aspects and refinements. These authors have also used size metrics to quantify the number of components and lines of code in an SPL implementation. Their study, however, did not consider a significant suite of software metrics and did not address SPL evolution and stability. In other work Greenwood et al. [17] used similar suites of metrics to ours to assess the design stability of an evolving application. However, they did not target at assessing the impact of changes in the core and variable features of SPLs.

In this study, we analyzed and compare development of variability mechanisms to evolve SPLs, using FOP and two OO-based programming techniques. The evaluation was based on the modularity and stability metrics used to quantify cohesion, coupling,

and other properties of features [16]. Due to the exploratory nature of our study, we understand that it has some constraints and the results may not repeat in other contexts. For instance, the WebStore SPL was built for academic purpose. Therefore, it is a small software application in terms of the number of features and lines of code compared to industry-strength SPLs. However, to minimize this issue, we carefully designed and implemented WebStore to incorporate characteristics and changes similar to what is present in practice.

Of course, not all kinds of changes have been incorporated during the SPL evolution. Aware of this, we intend to perform a follow up of this study considering another SPL (larger and with more features). The use of AHEAD could also be pointed out as a constraint in our comparative study, since it is not the only existing FOP language. However, we chose AHEAD because it is stable and widely-used FOP language [6, 7]. Furthermore, it presents some unique characteristics not available in other FOP languages. Finally, our study contributes to build up a body of knowledge that allows the comparison of AHEAD and other FOP or non-FOP languages.

7 Conclusion

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 an SPL 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: feature modularity and change propagation.

Some interesting results emerged from our analysis. First, the FOP design of the studied SPL 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 (Section 5.1). Second, we observed that FOP class refinements adhere more closely the Open-Closed principle [21]. Furthermore, such mechanisms usually scale well for dependencies that do not involve shared code and facilitate multiple different product instantiations.

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 implementation that uses design patterns is the most rigid one to accommodate changes, and consequently, the one that requires more components insertions and modifications during the SPL evolution. As previously discussed (Section 5.1), the results also showed 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).

Acknowledgments

This work was partially supported by FAPEMIG, grant APQ-02932-10.

8 References

1. 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 '09, pp. 243—254. ACM, Virginia, New York (2009)
2. 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)
3. Apel, S., Batory, D.: When to Use Features and Aspects? A Case Study. In: GPCE, Portland, Oregon (2006)
4. Apel, S. et al. Aspectual Mixin Layers: Aspects and Features in Concert. Proceedings of ICSE'06, Shanghai, China (2006)
5. Apel, S., Leich, T., Saake, G.: Aspectual feature modules. IEEE Trans. Softw. Eng., 34:162–180. (2008)
6. Batory, D.: Feature-Oriented Programming and the AHEAD tool suite. In: 26th International Conference on Software Engineering, ICSE'04, pp. 702—703. IEEE Computer Society, Washington (2004)
7. Batory, D., Sarvela, J., Rauschmayer.: Scaling step-wise refinement. IEEE Transactions on Software Engineering, 30(6):355–371 (2004)
8. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. Computing Surveys, 17 (4): 471–522 (1985)
9. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
10. Dantas, F., Garcia, A.: Software Reuse versus Stability: Evaluating Advanced Programming Techniques. In: 23th Brazilian Symposium on Software Engineering, SBES'10 (2010)
11. Eaddy, M. et al. Do Crosscutting Concerns Cause Defects?, IEEE Trans. on Software Engineering (TSE), vol. 34, 497-515 (2008)
12. 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, ICSE '08, pp. 261--270. ACM, New York (2008)
13. Figueiredo, E. et al. On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. Proc. of European Conf. on Soft. Maint. and Reeng. (CSMR). Athens (2008)
14. Figueiredo, E., Sant'Anna, C., Garcia, A. and Lucena, C.: Applying and Evaluating Concern-Sensitive Design Heuristics. In: 23rd Brazilian Symposium on Software Engineering (SBES). Fortaleza, Brazil (2009)
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison Wesley (1995)
16. Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., and von Staa, A. (2005). Modularizing design patterns with aspects: a quantitative study. In Proceedings of the 4th international conference on Aspect-oriented software development, AOSD'05, pages 3–14, New York, NY, USA. ACM. (2005)

17. Greenwood, P. et al.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: ECOOP, Berlin (2007)
18. Grubb, P., Takang, A. A.: Software Maintenance: Concepts and Practice. World Scientific Publishing Company, New Jersey (2003)
19. 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), 2000.
20. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features using AspectJ. In: International SPL Conference (2007)
21. Meyer, B.: Object-Oriented Software Construction, 1st ed. Prentice-Hall, Englewood Cliffs (1988)
22. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: 12th ACM SIG-SOFT twelfth international symposium on Foundations of software engineering, SIGSOFT'04/FSE-12, pages 127--136, New York, NY, USA. ACM. (2004)
23. Mezini, M., Ostermann, K. Conquering Aspects with Caesar. In 2nd International Conference on Aspect-Oriented Software Development (AOSD). 2003. Boston, USA.
24. Prehofer, C. Feature-oriented programming: A fresh look at objects. ECOOP 1997: p 419–443. (1997)
25. 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)
26. Yau, S. S. and Collofello, J. S. Design Stability Measures for Software Maintenance. IEEE Transactions on Software Engineering, 11(9), p. 849-856, 1985.