Evaluating Delta-Oriented Programming for Evolving Software Product Lines

João P. Diniz
Federal University of Minas Gerais
Belo Horizonte, Brazil
jpaulo@dcc.ufmg.br

Gustavo Vale
University of Passau
Passau, Germany
vale@fim.uni-passau.de

Felipe Gaia
Federal Institute of São Paulo
Hortolândia, Brazil
felipegaia@ifsp.edu.br

Eduardo Figueiredo
Federal University of Minas Gerais
Belo Horizonte, Brazil
figueiredo@dcc.ufmg.br

***Abstract*—Managing variability is a hard task for every technique that develops variability-rich systems, such as software product lines (SPL), especially in its evolution. Hence, to be effective a technique should provide stability and respect the Open-Closed principle. Among the techniques to develop SPLs, delta-oriented programming (DOP) seems to be promising given its flexibility. There are two strategies in DOP development: starting from a simple core and from a complex core. Simple core is the implementation of a minimum valid product. Complex core, on the other hand, can include many varying features. This study aims to evaluate the stability of delta-oriented SPLs in evolutionary scenarios. To do that, we develop, evolve, and compare 5 releases of an SPL using both strategies to develop DOP. Our evaluation focuses on size, change propagation, and modularity of all releases. The results show that DOP has means to develop stable SPLs. In addition, simple core is usually better than complex core in most cases.**

Keywords—Software Product Lines; Delta-Oriented Programming; Variability; Modularity; Metrics

I. Introduction

Variability-rich software systems, such as open platforms, self-adaptive systems, systems of systems, and software product lines (SPLs), face the challenge of managing variability. In the context of SPLs, variability can be expressed by the variation of features between the products. Features are defined as modules with consistent, well-defined, independent, and combinable functions [2].

To be effective, variability mechanisms should support non-intrusive and self-contained changes that favor insertions and do not require deep modifications in existing features. In other words, variability mechanisms should adhere to the Open-Closed principle [11] in the evolution of variability-rich software systems. The Open-Closed principle states that software should be open for extension, but closed for modification in existing features.

Several techniques that support variability management have been proposed to develop SPLs, such as conditional compilation (CC) [1], feature-oriented programming (FOP) [4], aspectual feature modules (AFM) [3], and delta-oriented programming (DOP) [13]. In special, DOP is one of the most recently proposed techniques - it is inspired by FOP - and claims to provide flexibility in variability management [15].

In DOP, a product line is represented by a core (delta) module and a set of (other) delta modules. To develop in DOP, two strategies can be used [13]: starting from a simple core and starting from a complex core. However, in both strategies, the core delta module must implement a valid product configuration. In the first one, just a reduced number of features are introduced to the core. Hence, other features are added by other delta modules. Unlike simple core, the complex core strategy starts with many features in the core module and some of these features are removed from this module depending of the product configuration.

Despite the fact that DOP has been compared to other techniques [14][15], we did not find any study evaluating both strategies (simple and complex core) in the context of SPL evolution. We just find one study comparing the size of SPLs in terms of number of delta modules and lines of code using both strategies [13]. To fill this gap, this study evaluates the evolution of one SPL implemented in DOP, named MobileMedia, with respect to its stability. Hence, we develop, measure, and compare five releases of the SPL implemented using both strategies (a total of 9 releases – the base and other four releases for each strategy). Our evaluation is based on size, change propagation, and modularity metrics of all developed releases at three different granularity levels: components, operations, and lines of code.

Our results suggest that DOP supports developing stable variability-rich software systems from the perspective of software product lines. Additionally, the simple core strategy presented better results when compared with complex core strategy in most of the analyzed scenarios. The contributions of this paper are mainly focused on developers that want to develop variability-rich systems using DOP, but they can also help in comparisons of other techniques to develop SPLs. Our results may also help the delta-oriented community, and DeltaJ, in further improvements.

The remainder of this paper is organized as follows. Section II presents a background for this study. Section III presents the study settings, including the target SPL and the evolution scenarios. Section IV analyzes the developed releases and discusses the results quantitatively. Section V discusses qualitative results and lessons learned. Section VI presents threats to the validity of this study. Section VII discusses related work. Finally, Section VIII concludes this paper and points out directions for future work.

II. Managing Variability in DOP

This section revisits important definitions for understanding the rest of the work. Delta-Oriented programming (DOP) is a modular and flexible composition technique that can be used to manage variability needed in the context of variability-rich systems, such as software product lines [13]. This technique is based on modular units, called deltas. Similar to feature-oriented programming (FOP), a delta module can add new member into a class and an existing member can be modified. However, unlike FOP, DOP also allows the removal of a class, a class member (method or attribute) and even import statements [13].

There are two strategies to develop an SPL using DOP. These strategies are: *Simple Core* and *Complex Core*. In the simple core strategy, the core module contains an implementation of a minimum valid product and other modules add/remove varying features. Hence, in this strategy the core module take only the mandatory features and a minimal number of varying features, if applicable. For example, Listing 1 depicts code fragment of the simple core, implemented by the core module "cBase". In the complex core strategy, the core module may be implemented with any product for a valid feature configuration. That is, it is expected that a larger set of features will be included in complex core than in the simple core implementation. The strategies just define the starting point for generating all other products by combining delta modules.

Normally, the core module is the largest file of a DOP project in both strategies. The main difference is that, in order to provide a product configuration containing fewer varying features than that inside the core, source code has to be removed from it by another delta module. It is more common when using the complex core strategy because it accumulates more responsibilities. There is no defined rule for choosing varying features to be implemented in the complex core. One way to make this decision is by choosing features that are selected from the majority of the SPL clients.

DeltaJ is a Java-like language that supports DOP by organizing classes and interfaces in delta modules. It was proposed by the same authors of DOP and can be integrated to Eclipse IDE by plug-in. DeltaJ current version is 1.5, released in order to support Java 1.5 syntax [10].

In DeltaJ, a product generated by composition process is a Java code, respecting the customizations of the configured product variability points. For instance, Listings 1, 2, 3, and 4 present some DeltaJ code fragments extracted from four delta modules implemented for this work. Listing 1 and 3 present the Core module of simple and complex strategies. On the other hand, Listing 2 and 4 present the Sorting module. In the case of simple strategy, the Sorting module will be added if it is required in the product configuration. In the case of complex strategy it is part of the core module, hence, it will be removed if this Sorting module is not required by a product configuration. In both cases, the changes are applied on "cBase".

```
delta cBase {
...

adds {
 public class PhotoLScreen extends List {
   public static final Command add;
    add = new Command("Add", Command.ITEM, 1);
    public void initMenu() {
      this.addCommand(add);
     ...
    }
   }
  }
 }
```

Listing 1. Core module using simple core strategy

```
delta dSorting {
...

modifies PhotoLScreen {
  adds public static final Command sort;
  sort = new Command("Sort by Views",
                        Command.ITEM, 1);
   modifies initMenu() {
     original();
     this.addCommand(sort);
   }
  }
 }
```

Listing 2. Sorting module using simple core strategy

```
delta cBase {
...

adds {
 public class PhotoLScreen extends List {
   ...
   public static final Command add;
   public static final Command sort;
   add = new Command("Add", Command.ITEM, 1);
   sort = new Command("Sort by Views",
                       Command.ITEM, 1);

   public void initMenu() {
     this.addCommand(add);
     ...
     this.addOtherCommands();
   }

   private void addOtherCommands() {
     this.addCommand(sort);
   }
  }
 }
}
```

Listing 3: Core module using complex core strategy

```
delta dNotSorting {
...

  modifies PhotoLScreen {
    removes sort;

  modifies addOtherCommands() {
    return;
  }
 }
}
```

Listing 4: Sorting module using complex core strategy

III. Study Settings

This study aims to evaluate the stability of delta-oriented programming to evolve software product lines. We believe that it is important because, as in any software life cycle, changes are expected and must be accommodated during the SPL evolution. When changes come to SPLs, it is expected to have even more impact than single software systems, since changes may affect several products. Thereby, it is important that techniques to develop and evolve SPLs follow the Open-Closed principle and be stable.

To achieve our goal, we followed three steps: (i) define some research questions, (ii) select and evolve a system (SPL), and, (iii) select some metrics to measure stability and follow a measurement process. These steps are detailed in the next sub-sections.

A. Research Questions

As we just mentioned, a technique to develop SPLs should address the Open-Closed principle and be stable. Thereby, we decided to evaluate the evolution of DOP in terms of size, change propagation, and modularity. This decision results in the four following research questions.

RQ1: *Does the simple core strategy generate a small version than a complex core SPL?*

RQ2: *Does the simple core strategy have smoother change propagation impact than the complex core one?*

RQ3: *Is simple core more modular than the complex core strategy?*

RQ4: *Is it easier to evolve DOP software product lines using simple core than complex core?*

With the answer of the first three research questions, we expected to have quantitative data of stability for a software product line developed using delta-oriented programming. Furthermore, with our lessons learned in this study, we believe that we are able to answer our forth research question giving not just a quantitative perspective, but also a qualitative one.

B. Target SPL

In this study, we focus on an SPL named MobileMedia. MobileMedia is an SPL that provides products for media management in mobile devices [8]. We selected this SPL due to two main reasons. First, it has been investigated in the literature [7][8][9]. Second, MobileMedia has several variability points related to heterogeneous mobile platforms and many varying features. In fact, it encompasses different degrees of complexity and different levels of scalability.

Figure 1 presents a simplified view of the MobileMedia SPL feature model used in this study. Feature models are used to represent the mandatory and variable features in SPLs [6]. Examples of mandatory features are Album Management, Create Album, and Create Photo. On the other hand, examples of varying features are Sorting, Favourites, and SMS Transfer. The numbers between parenthesis indicate the release each feature was included. For instance, the Edit Photo Label feature was included in release 2.

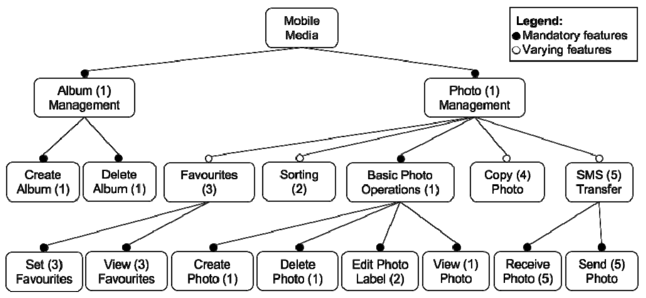

Figure 1. MobileMedia Feature Model.

This study considers five successive releases of MobileMedia that we develop using the last version of DeltaJ (1.5) and both strategies to develop DOP (Section II). Table I summarizes changes made in each release. The releases comprised different types of changes involving mandatory and varying features. The first column of Table I shows the release number (R1 to R5). The second and third columns briefly explain which types of changes each release encompassed. The purpose of these changes is to exercise the implementation of mandatory and varying features to assess variability mechanisms properties in the context of the evolving SPL.

TABLE I. MobileMedia Evolutionary Scenarios

	Description	Type of Change
R1	MobileMedia core	None
R2	Features added to count photo visualizations and sorting, and editing photo label.	New varying and core features
R3	Feature added to set favorite photos.	New varying features
R4	Feature added to copy photos to albums.	New varying features
R5	Feature added to send and receive photo by SMS	New varying features

Regarding the flow of development, we first implemented the 5 releases of MobileMedia using the simple core strategy. In this case, the core module is composed of a minimum valid product configuration. In other words, the core module is implemented with just mandatory features. The next releases (R2 to R5) were implemented by the core delta module and other delta modules, responsible for the source code of varying features. For instance, since release 2 has only one varying feature (Sorting), two delta modules are required: one for the core and one to implement Sorting.

After, we started the complex core. The source code of release 1 remains the same for both strategies, since this release of MobileMedia only contains mandatory features. Release 2 contains just a varying feature: Sorting. Therefore, it was chosen to be part of the complex core. To be consistent in our decision, we applied the same strategy to implement the other three releases (R3 to R5). In other words, the complex core of all releases includes only one varying feature, namely Sorting. If a configuration does not have this feature, a delta module is responsible to remove Sorting from the product. In release R3, we opted for not including the Favourites varying feature into the complex core, since if it has all varying features it does not seem to be the right implementation decision for any evolving SPL. In releases 4 and 5, no varying features were included in the complex core due to some limitation of the scenarios that will be better detailed in Sections V and VI.

C. *Measurement Process*

Our measurement process consists on the analysis of three granularity levels: components, operations, and lines of code (LOC). In this study, we consider components as delta modules and delta structures that add, modify, or remove classes or interfaces. We also consider operations as methods and delta structures that add, modify or remove methods. Before we describe the measurement process, we briefly describe each used metric.

In addition to the granularity level, we analyze size, change propagation, and modularity. To measure size, we use the number of components, operations, and lines of code. To measure change propagation, we use the number of components, operations, and lines added, removed or modified. To measure modularity, we use a set of four concern metrics. The metrics of this set and their definitions are as follows. Concern Diffusion over Components (CDC) counts the number of components contributes to the implementation of a feature. Concern Diffusion over Operations (CDO) counts the number of operations that contributes to the implementation of a feature. Lines of Concern Code (LOCC) counts the total number of lines of code that contribute to the implementation of a feature. Finally, Concern Diffusion over Lines of Code (CDLOC) computes the degree of feature tangling. It counts the number of "switches" between lines of code realizing one feature and lines of code realizing other features.

We select these metrics to measure size, change propagation, and modularity because they are well-known to measure these properties and they have been used and validated in several previous work [7][8][9][12].

Regarding the measurement process, all releases using both strategies were developed using Git[1] and GitHub[2]. Hence, we collected the size and change propagation metrics from the GitHub graphical interface. The data collection for modularity metrics followed a distinct procedure. To measure CDC, CDO, LOCC, and CDLOC, we created a prototype tool that captures the metrics from the integrated development environment and run scripts developed in R[3].

IV. Results and Analysis

This section presents and analyzes the results, which were obtained by metrics concerning size, change propagation and modularity, extracted from the source code of the whole SPL evolution scenarios implemented.

A. *Size*

This section presents size metrics, which are counted in three granularity levels: number of components, number of operations, and LOC (Section III.C). Table II summarizes size metrics at the three granularity levels for each release using both strategies. For example, release 1 (R1) of MobileMedia has number of components, operations, and LOC equals to 37, 135, and 1744, respectively. Regarding the same metrics for release 2 (R2), we can see 46, 163, and 2099 implementing simple core strategy and 47, 175, and 2140 implementing complex core strategy.

TABLE II. Results of Size Metrics

		Simple Core				Complex Core			
#	***R1***	***R2***	***R3***	***R4***	***R5***	***R2***	***R3***	***R4***	***R5***
C	37	46	53	68	91	47	54	71	93
O	135	163	173	200	263	175	186	215	278
L	1744	2099	2250	2635	3474	2140	2296	2683	3524

C = #Components; O = #Operations; L = #LOC

As we can see, in all three granularity levels the size metrics increase while the SPL evolves. In addition, we can see that, in general, values of complex core implementation grow faster than that of simple core. This observation is justified, mainly, due to the source code concerning the Sorting feature. In the simple core version, it is implemented by one specific delta module while in the complex core, two delta modules were needed, the core and a specific one.

Finally, although the number of components is slightly higher in the complex core implementation (R2 to R5, as can be seen also in Table II), the number of delta modules was exactly the same in each release of both simple and complex core implementation strategies. Release 1 has only one delta, the core module. Releases from 2 to 4 have 2, 3 and 4 deltas, respectively: the core module and one delta module for each varying feature. In release 5, there are six delta modules. All three features related to SMS were included in one single delta module because a product configuration must either contain all three or none of these features. In addition, a new delta module was implemented to compose products when Copy Photo and all SMS features are selected. This solution is better explained in Section V.

Regarding to answer RQ1, the simple core SPL resulted in a smaller SPL when compared with the complex core SPL. Hence, in this case we assume that simple core is better because the inclusion of new features means in less effort than complex core, mainly, from release 1 to 2.

B. *Change Propagation*

This section presents the results of change propagation metrics throughout all releases of the SPL. Hence, Table III summarizes the change propagation results for all evolutionary releases (R2 to R5) for both implementation strategies. As explained in Section II.C, the analysis uses traditional metrics of change impact considering three levels of granularity: components, operations, and LOC. Symbols "+", "-" and "~" in Table III means, respectively, number of additions, removals and modifications.

A general interpretation is that a lower number of modified and removed artifacts suggest more stable

[1] https://git-scm.com/
[2] https://github.com/
[3] https://www.r-project.org/

implementation, possibly supported by the variability mechanisms. Furthermore, a higher number of additions indicate that evolution is being supported by non-intrusive extensions and the solution is more close to the Open-Closed principle. Even though the number of additions using complex core strategy is always equals or higher than simple core strategy, we cannot conclude that the complex core strategy adheres better to the Open-Closed principle, because the number of changed and removed artifacts are almost the same than the simple core strategy.

TABLE III. RESULTS OF CHANGE PROPAGATION

		Simple Core				Complex Core			
		R2	***R3***	***R4***	***R5***	***R2***	***R3***	***R4***	***R5***
Components	+	10	7	18	22	11	7	17	22
	-	1	0	1	0	1	0	0	0
	~	9	0	24	19	9	2	24	19
Operations	+	33	10	48	28	37	10	58	28
	-	4	0	27	2	4	0	30	2
	~	9	0	20	14	12	2	9	20
LOC	+	386	160	739	981	429	172	833	990
	-	28	0	359	151	27	10	453	150
	~	19	0	44	5	19	0	38	4

Another interesting analysis is that the values of complex core are mostly equal or higher than simple core. It means that this implementation strategy tends to demand more effort than simple core regarding the three granularity levels analyzed. Releases 2 and 4, comparing simple with complex core strategies, present relevant differences. The reason of concerning values of release 2 is due to the inclusion of the feature Sorting in the delta core module, as was explained in Section IV.A. Regarding release 4, in both simple and complex core implementations, some refactorings were performed in the source code.

Regarding to answer RQ2, the simple core strategy had less lines of code and almost the same components and operations removed or changed in the majority of cases than the complex core strategy. Hence, we assume that the simple core strategy have smoother change propagation impact than complex core strategy.

C. *Modularity*

This section presents the results for modularity metrics throughout all releases of the SPL. The measurement data have been collected and analyzed for all features, but we choose two features with significant difference to graphically analyze. This analysis aims the comparison of both implementation strategies. For example, the upper left plot of Figure 2 shows values of CDC metric for the Sorting feature in all four releases that it is presented. Note that this feature was included in the second release (R2) and, hence, the first release (R1) was suppressed from the chart.

Three trends concerning the metrics values emerged from the analyses. The first one is related to features that did not produce different impact on the modularity of the implementation strategies: the lines fully overlap in the plots or lines partially overlap, presents same behavior and the difference between values can be considered insignificant.

The second and most expressive trend refers to the values comparison for the Sorting feature using both implementation strategies. Figure 2 shows the four modularity metrics obtained for this feature, presenting values significantly different between the strategies. The lower values of simple core mean that this strategy is better modularized than complex core strategy.

Source code fragments showing part of the implementation of Sorting in simple core (Listing 2) and in complex core (Listings 3 and 4) helps us to understand the differences in metrics values. In simple core, Sorting is completely implemented only by the delta module "dSorting" while it is implemented by both delta modules ("dBase" – the core – and "dNotSorting") in the complex core. Therefore, more components, operations and LOC were needed, resulting in an increasing of CDC, CDO and LOCC values. In addition, since the source code is completely implemented in "dSorting" the Sorting code is not tangled with other feature's code, while it is tangled with other features in "cBase" of complex code. Such tangling increases the CDLOC values.

Other interesting observation for the Sorting feature in Figure 2 is the downward behavior for CDO, LOCC and CDLOC metrics between R3 and R4 releases. This behavior is related to refactoring in design at component level. Component refactoring that impacts the architectural design tends to affect more complex core than simple core strategy, because not only core features but also the variabilities are affected. In the case of complex core, this impact is even better because the core modularization is improved. This situation occurs in release 4 (R4), where some components were split and renamed with a specific goal to prepare the SPL for future releases.

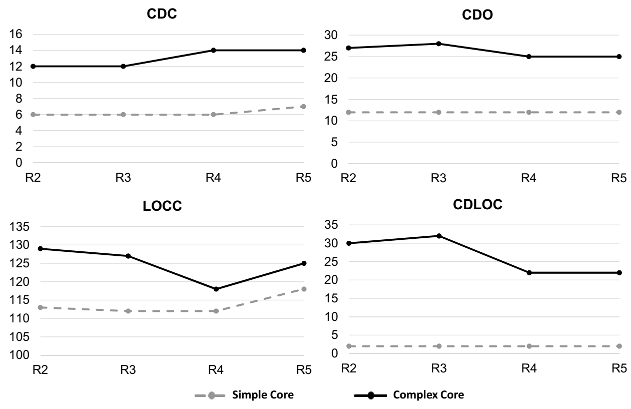

Figure 2. Metrics for the Sorting feature.

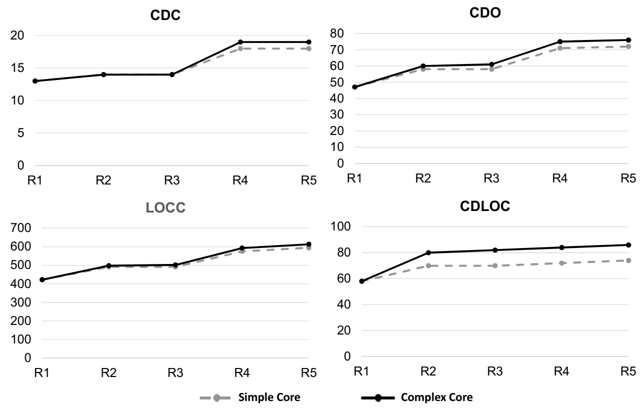

Figure 3. Metrics for Photo Management feature.

Finally, the third trend observed in Figure 3 is concerning PhotoManagement. This feature was the most affected by the inclusion of Sorting into the complex core module. In other words, the source code of PhotoManagement became more tangled after including Sorting-related code. The chart for CDLOC in Figure 3 highlights this trend.

Regarding to answer RQ3, the simple core implementation version of MobileMedia is more modular than the complex core. This can be easily seen comparing the four modularity metrics graphics concerning the Sorting feature and the CDLOC graphic concerning the Photo Management feature.

V. Discussion

From the analysis carried out in the Section IV and the experience acquired with the SPL implementations, three interesting situations, discussed below, naturally emerged with respect to ease of SPL evolution using DOP technique.

Flexibility of delta modules over features. In this study, the most of the resources provided by DeltaJ could be used along the whole implementation activity. The delta modules are used to add, modify and remove classes, methods and attributes. In addition, as mentioned in Section IV.A, one delta module was necessary to provide code to integrate the features Sorting and SMS Transfer, when both are selected to compose a product. Since delta modules are independent from features in DeltaJ, we used this flexibility, one of the strengths of DOP in comparison with FOP [13], to implement a modularized and independent solution for such integration.

Finally, DOP technique looks like to be more effective than FOP, since it provides more flexibility, while detaching features from modules. A delta module can be part of one or more features and one feature can be compound by one or more delta modules. A study in such direction should be performed to investigate this assumption.

Limitations of DeltaJ programming language. Although DeltaJ is in evolution since it was proposed in 2010 [10], this programming language still presents some limitations that may compromise development of wider projects, even in academic environments. Locking text editor to refresh outline window from Eclipse IDE after each word typing, even without saving the current file, forces the developer to close such window that could be used to help keeping up with the source code. Other examples are lack of meaningful error messages and documentation. The possibility of knowing about Java errors only after explicitly generating a product's resulting source code is also a painful process of programming in DeltaJ. Infinity loops while choosing the products to be generated forces the user to close and reopen the projects. Another limitation is that DeltaJ presents three compilation errors every time the operator "*" is used, even on a simple multiplication statement. When it is replaced by operators like "+", "-" or "%", for example, non-errors are displayed.

Simple core strategy is recommended over complex core strategy. Concerning the implementation done in this study, the simple core DOP implementation strategy has shown itself as the most appropriated in comparison with complex core one, because it contributes with a more modular development and with fewer LOC. On the other hand, the complex core strategy really allows the usage of more resources provided by DOP/DeltaJ. During the implementation of complex core version, specifically in release R4, we originally opted by including the varying feature Copy Photo in the core, together with Sorting. Nevertheless, a new problem emerged: more fine-grained levels, where the variability mechanism should act, became necessary. Code statements related to the Copy Photo, located inside methods of general-purpose controller classes, had to be extracted into new specific-purpose methods. For example, one additional method with the same purpose of the "addOtherCommands" in "cBase" module in Listing 2 was necessary. Considering this issue, we opted to keep all the remaining releases with only the varying feature Sorting in complex core, since this approach have been enough to provide us some relevant results.

Regarding to answer RQ4, our quantitative analysis clearly shows that simple core strategy is better than complex core in evolving SPLs, but in a more stable scenarios where not many new features are included and some features are always in the product configuration we recommend the use of complex core strategy.

VI. Threats To Validity

Even with the careful planning, this research work can be affected by different factors that can threat its main findings. This section discusses the study validity with respect to the four groups of common threats: internal, external, construct, and conclusion validity [16].

Internal validity concerns whether the effect is caused by the independent variables or by spurious factors. For instance, only one developer designed and implemented the SPL releases used in this study. Although the code was reviewer by other researchers, there is space for different design and coding decisions. Different decisions would produce different results. To minimize this threat, all designs of MobileMedia were carefully developed to take the best of each implementation technique. Cross-discussion between two or more developers took placed when required.

Concerning external validity, some factors may limit the generalization of our results. Although the MobileMedia SPL were carefully designed to be as general as possible, it should be considered that this SPL is special purpose system. Hence, it may not include all properties of real systems. However, MobileMedia was used in research studies with similar purposes of ours [7][8][9]. Further replications of this study are required to confirm or refute our results.

One issue about construct validity is how much the selected metrics support the findings of this research investigation. The used metrics offer a limited view on design modularity and stability problems, i.e., they only permit to draw indirect conclusions. To minimize this threat, we carefully selected some of the most well established metrics, such as the CK metrics [5]. In addition, change propagation have been used in the previous studies [7][9].

With respect to the conclusion validity, since many data points have been collected, the reliability of the measurement and derived findings might be an issue. However, it is important to consider the results gathered from several metrics rather than from just one metric in particular. In fact, the multi-dimensional analysis allows us to grasp which measurement outliers are significant and which are not.

VII. Related Work

As DOP is a technology based-on FOP, it is not hard to find studies that compare FOP with DOP [13][14][15]. Nevertheless, as this work aims to compare the evolution of DOP using simple and complex core strategies we focus on papers that did that. To the best of our knowledge, we just find a paper that compare simple and complex core. In this paper, Schaefer et al. [13] present a simple evaluation regarding number of delta modules and lines of code between simple and complex core for 4 SPLs, being just one SPL with more than two thousand lines of code and 7 delta modules. Comparing with our work, we develop five versions of one SPL in both strategies. Our SPLs (both strategies) are bigger than the biggest one of the related work and we compare not just the number of delta modules and lines of code, but also, change propagation in the number of components and operations, and modularity of both approaches using 4 concern metrics.

VIII. Conclusions and Future Work

Given that DOP is one of the most recent technologies to develop variability-rich software systems and changes are expected in any software life cycle, we evaluate the evolution of an SPL developed using the two strategies of DOP (simple and complex core). Our evaluation includes size, change propagation and modularity. The results shows that simple core strategy seems to be more stable in the majority of the explored scenarios, but complex core can be very useful in more mature systems having some features selected in many product configurations.

As future work, we can list the following tasks. First, the extension of previous works [7][9] in order to make comparison of our data with other variability management mechanisms, such as FOP, AOP and CC. Second, the extension of our prototype tool to collect, automatically, change propagation measures and modularity metrics data from a DeltaJ source code. Finally, the investigation on how DeltaJ or other technologies for developing variability-rich systems could be used to implement variability-rich systems which must interact with third party frameworks and APIs.

Acknowledgments

This work was partially supported by CAPES and CNPq (grant 424340/2016-0).

References

[1] V. Alves, , A. Costa Neto, S. Soares, G. Santos, F. Calheiros, V. Nepomuceno, D. Pires, J. Leal, P. Borba, “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.

[2] S. Apel, C. Kästner, and C. Lengauer, C. FeatureHouse: Language-Independent, Automated Software Composition. In Proceedings of the International Conference on Software Engineering (ICSE), p. 221–231, 2009.

[3] S. Apel, T. Leich, and G. Saake, “Aspectual Feature Modules.”, IEEE Transactions in Software Engineering, 34, p. 162-180, 2008.

[4] D. Batory, J. Sarvela, and A. Rauschmayer, “Scaling step-wise refinement”, IEEE Trans. on Sof. Eng., 30 (6) pp. 355-371, 2004.

[5] S. Chidamber, C. Kemerer. “A Metrics Suite for Object Oriented Design”. IEEE Transactions on Software Eng 20(6):476–493, 1994.

[6] K. Czarnecki, and U. Eisenecker, “Generative programming: methods, tools, and applications”, 2000.

[7] G. Ferreira, F. Gaia, E. Figueiredo, and M. Maia, “On the use of feature-oriented programming for evolving software product lines - A comparative study.” Science of Computer Prog., 93, p. 65–85, 2014.

[8] E. Figueiredo, N. Cacho, C. Sant’Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas, “Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability.” In: 30th International Conference on Software Engineering (ICSE), p. 261-270, 2008.

[9] F. Gaia, G. Ferreira, E. Figueiredo, and M. Maia, “A Quantitative and Qualitative Assessment of Aspectual Feature Modules for Evolving Software Product Lines.” Science of Computer Programming, vol. 96, p. 230-253, 2014.

[10] J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani, “DeltaJ 1.5: delta-oriented programming for Java 1.5.” In Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages and Tools, p. 63-74, 2014.

[11] B. Meyer, “Object-Oriented Software Construction”, first ed., Prentice-Hall, Englewood Cliffs. 1988.

[12] C. Sant’anna, A. Garcia, C.Chavez, A. von Staa and C. Lucena “On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework.” In Brazilian Symposium on Software Engineering (SBES), p. 19-34, 2003.

[13] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, “Delta-Oriented Programming of Software Product Lines.” In Proceedings of the International Software Product Line Conference (SPLC), p. 77–91, 2010.

[14] I. Schaefer, L. Bettini, and F. Damiani, “Compositional Type-Checking for Delta-Oriented Programming”. In Proceetings of the International Conference on Aspect-Oriented Software Development (AOSD), p. 43-56, 2011.

[15] I. Schaefer and F. Damiani, “Pure Delta-oriented Programing”. In Proceedings of International Workshop on Feature-Oriented Software Development (FOSD), p. 49-56, 2010.

[16] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén. Experimentation in Software Engineering. Springer, 2012.