Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability

Eduardo Figueiredo[1], Nelio Cacho[1], Claudio Sant'Anna[2], Mario Monteiro[3], Uira Kulesza[4], Alessandro Garcia[1], Sergio Soares[3], Fabiano Ferrari[1], Safoora Khan[1], Fernando Filho[3], Francisco Dantas[5]

[1]Computing Department, Lancaster University, United Kingdom
[2]Pontifical Catholic University of Rio de Janeiro, PUC-Rio, Brazil
[3]Computer Science Department, Pernambuco State University, Brazil
[4]CITI/DI/FCT, Universidade Nova de Lisboa, Portugal
[5]Computer Science Department, State University of Rio Grande do Norte, Brazil

{e.figueiredo, n.cacho, a.garcia, f.ferrari, s.shakil-kahn}@lancaster.ac.uk, claudios@inf.puc-rio.br, {mqm, sergio, fernando.castor}@dsc.upe.br, uira@di.fct.unl.pt, franciscodantas@uern.br

ABSTRACT

Software product lines (SPLs) enable modular, large-scale reuse through a software architecture addressing multiple core and varying features. To realize the benefits of SPLs, their designs need to be stable. Design stability encompasses the sustenance of the product line's modularity properties in the presence of changes to both the core and varying features. It is usually assumed that aspect-oriented programming promotes better modularity and changeability of product lines than conventional variability mechanisms, such as conditional compilation. However, there is no empirical evidence on its efficacy to prolong design stability of SPLs through realistic development scenarios. This paper reports a quantitative study that evolves two SPLs to assess various design stability facets of their aspect-oriented implementations. Our investigation focused upon a multi-perspective analysis of the evolving product lines in terms of modularity, change propagation, and feature dependency. We have identified a number of scenarios which positively or negatively affect the architecture stability of aspectual SPLs.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics-*Product metrics*; D.3.3 [**Programming languages**]: Language Constructs and Features

General Terms

Measurement, Design, Experimentation.

Keywords

Product lines, aspect-oriented programming, empirical evaluation.

1. INTRODUCTION

Software product lines (SPLs) [9, 23] represent an increasingly popular technology to support the derivation of a wide range of applications. They enable modular, large-scale reuse through a core software architecture addressing recurring features in a certain domain and multiple variability points. The design of industrial SPLs is often incremental and gradually evolves to cope with new stakeholders' needs [1, 9, 18]. However, their longevity is highly dependent on the ability of the implementation-level variability mechanisms to sustain their architecture stability [1, 3, 4, 9, 16]. Ideally, the modular structure of the core and variable features of a SPL should not succumb in the presence of change requests. In fact, evolution of product lines imposes deep concerns on software engineers due to the diverse nature of certain frequent changes, such as: (i) introduction and removal of crosscutting and non-crosscutting features, and (ii) the transformation of mandatory features in optional or alternative ones and vice-versa.

The inefficacy of the 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 [1, 16]. Many authors [1, 3, 22] advocate that aspect-oriented programming (AOP) [17] is an effective technique to support feature variability and prolong the stability of product-line designs. AOP is aimed at supporting the encapsulation of crosscutting features into new modular units - the aspects - through new composition mechanisms, such as pointcut-advice and inter-type declarations. The intention is to make the variation of crosscutting features more modular and evolvable when compared to industry-strength conventional variability mechanisms, such as conditional compilation [2].

Recent work has started to explore the use of AOP to improve the isolation of specific features in designs of frameworks [19] and product lines [16, 21]. However, none of them has analysed the impact of AOP on heterogeneous evolution scenarios of program families. Most of them are either of methodological nature [19] or not focused on objectively assessing the role of AOP on sustaining the SPL design's stability. They also do not investigate to what extend ripple effects are reduced across the core architecture and optional modules. In this context, it is important to systematically verify the suitability of AOP [17] for designing stable product lines, especially when compared to mainstream variability mechanisms, such as object-oriented (OO) constructs and conditional compilation. Even though there are a number of emerging academic programming techniques for supporting product-line development [21], the empirical evaluation of core AOP mechanisms is still very limited in the literature.

This paper presents a case study that quantitatively and qualitatively assesses the positive and negative impacts of AOP on a number of changes applied to both the core architecture and variable features of SPLs. Our investigation focused on several

evolution scenarios of two heterogeneous product lines (Section 3), called MobileMedia [29, 30] and BestLap [2], which were both implemented in Java and AspectJ. Conditional compilation was the variability mechanism used in the Java releases, which were used in turn with the goal of supporting an analysis of AOP. In other words, the goal of our comparative analysis was to observe to what extent AOP mechanisms provide or not enhanced product line stability in the presence of change tasks. The design stability evaluation of the Java and AspectJ versions were based on three conventional metrics suites for change impact [28] (Section 4), modularity [25] (Section 5), and feature dependency [15] (Section 6). We documented scenarios where AOP succeeds or not. For example, AOP copes well with the separation of features with no shared code. Furthermore, when adding an optional or alternative feature, AspectJ adheres better than Java to well-known design principles, such as the Open-Closed principle [20]. However, AOP is particularly vulnerable to changes targeting core features. For example, turning a mandatory feature into alternatives leads to ripple effects across the SPL design.

2. STUDY SETTING

This section describes the configuration of our study. Section 2.1 briefly explains the two variability techniques evaluated in this study. Section 2.2 describes the evaluation methodology.

2.1 Variability Programming Mechanisms

In order to enable the variation of software product lines (SPLs), this work considers two variability implementation techniques: AOP [17] and conditional compilation. We chose AspectJ [27] to implement variability with AOP because it is the most consolidated AOP language. Besides, our goal was to assess the suitability of core AOP mechanisms for handling variability rather than other emerging AOP mechanisms available in programming languages, such as CaesarJ [21]. Conditional compilation, on the other hand, is a well-known technique for handling variation [2]. Basically, preprocessor directives indicate pieces of code that should compile or not based on the value of preprocessor variables. Such decision may be at the level of a single line of code or to a whole file. For instance, Figure 1 describes a slice of code of the MobileMedia application where a logical connector is used to determine when the enclosed code of two features (*smsFeature* and *captureMedia*) must be compiled.

```
01 //#ifdef copyMedia
02 private void processMediaData(String mediaName,
                                  String albumName) {
03   MediaData mediaData = null;
04   //#if smsFeature || captureMedia
05   byte[] mediaByte = getCapturedMedia();
06   if (mediaByte == null)
07   //#endif
08    mediaData = getAlbumData().getMediaInfo(mediaName);
09   //#if smsFeature || captureMedia
10   if (mediaByte != null)
11     getAlbumData().
          addMediaData(mediaName, mediaByte, albumName);
12   else
13   //#endif
14     getAlbumData().addMediaData(mediaData, albumName);
15 }
16 //#endif
```

Figure 1. Variability with conditional compilation

Alternatively, AOP languages support the modular definition of features which are generally spread throughout the system and tangled with core features [1, 4, 16]. For instance, the pieces of code enclosed by *#if* and *#endif* in Figure 1 belong to *optional features* and AOP separates them using *pointcuts*, *advices* or *inter-type declarations*. Figure 2 shows a possible implementation of the variability points of Figure 1 using AspectJ mechanisms. The tangled code common to the *smsFeature* and *captureMedia* features is now modularised in a unique place (*around* advice).

```
01 public aspect SMSOrCaptureMedia {
02  pointcut processMediaData (...): execution(*
      *.processMediaData(...)) && this(...) && args(...);
03  void around (...): processMediaData(...) {
04    byte[] mediaByte = controller.getCapturedMedia();
05    if (mediaByte == null)
06      proceed(controller, mediaName, albumName);
07    else
08      controller.getAlbumData().
          addMediaData(mediaName, mediaByte, albumName);
09  }
10 }
```

Figure 2. Variability with AOP mechanisms

2.2 Study Phases and Assessment Procedures

The study was divided into three major phases: (1) the design and realisation of SPL change scenarios, (2) the alignment of SPL versions, and (3) the quantitative and qualitative assessments of the SPL versions and successive releases. In the first phase, an independent group of five post-graduate students was responsible for implementing the successive evolution scenarios of two SPLs: BestLap [2] and MobileMedia [29, 30] (Section 3). The original releases of both product lines used in this study were available in both AspectJ and Java (the Java versions use conditional compilation as the variability mechanism). Then, each new release was created by modifying the previous release of the respective SPL. For example, AspectJ release 2 evolved from AspectJ release 1. Best-of-breed design practices [5, 9] were applied throughout the creation of all the SPL releases. In order to assure them, there was also a validation of each scenario with professionals (e.g. BestLap developers) and researchers with long-term experience on the development of the target SPLs. Besides, the scenarios were extracted based on the consultation with such real designers in order to understand typical changes in product-line designs.

Development of the SPL Releases. In the first phase, we created eight releases of the MobileMedia (Section 3.1), available from [10], and five of BestLap (Section 3.2), not available due to copyright constraints. Both MobileMedia and BestLap have been successfully used in other studies involving modular design of SPLs [2, 29, 30], and so provided a solid foundation for our study. Notice that we did not target the comparison of the two SPLs (i.e. MobileMedia and BestLap). On the contrary, the objective of using more than one sample was to allow us to yield broader conclusions that are agnostic to specific SPLs.

SPL Alignment Rules. All SPL releases were verified according to a number of alignment rules (phase 2) in order to assure that coding styles and implemented functionality were exactly the same. Moreover, the implementation followed the same design decisions in that best practices [5, 9] were applied in all implementations to ensure a high degree of modularity and reusability. This alignment and validation activities were performed by two independent researchers. A number of test cases were exhaustively used for all the releases of the Java and AspectJ versions to ease the alignment process. These alignment procedures assure that the comparison between aspect-oriented

(AO) and non-AO versions is equitable and fair. Inevitably, some minor refactoring in the two versions had to be performed when misalignments were observed at the implementation artefacts or even at the design level. When these misalignments were discovered the developers for that particular version were notified and instructed to correct the implementation accordingly.

SPL Stability Assessment. The goal of the third phase was to compare the design stability of AO and non-AO designs. In order to support a multi-dimensional data analysis, the assessment phase was further decomposed in three main stages. The first stage (Section 4) evaluates the two implementations from the perspective of change propagation. The following stage (Section 5) is aimed at examining the overall maintenance effects in fundamental modularity properties through the product-line releases. The last stage (Section 6) focuses on assessing design stability in terms of how the implementation of feature 'boundaries' and their dependencies have evolved through the SPL releases. Traditional metrics were used in all the assessment stages, and will be discussed in the respective sections. All measurement results are available from [10].

3. TARGET PRODUCT LINES

For comprehensive investigation the initial decision entailed the selection of the target product lines. The two chosen SPLs are BestLap [2] and MobileMedia [29, 30]. They were selected due to several reasons. First, we believe these SPLs are representatives for the mobile devices domain, since they have (i) several variability points related to heterogeneous mobile platforms and (ii) many alternative and optional features. In fact, one of them is a real application of a software company. Second, both encompass different degrees of complexity and different levels of scalability. Also, assessment of more than one application from the same domain provides us with a fair comparison of design stability. Besides, Java and AspectJ solutions of both SPLs were available facilitating the analysis of the two investigated variability mechanisms: conditional compilation and AOP.

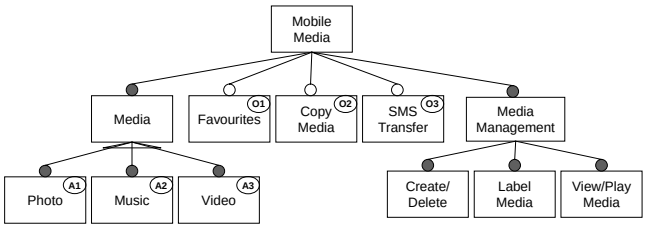

Figure 3. Simplified MobileMedia feature model

3.1 MobileMedia

MobileMedia is a SPL for applications with about 3 KLOC that manipulate photo, music, and video on mobile devices, such as mobile phones. It was developed based on a previous SPL called MobilePhoto [29], developed at University of British Columbia. In fact, in order to implement MobileMedia, the developers extended the core implementation of MobilePhoto including new mandatory, optional and alternative features (Section 3.1.1). Figure 3 presents a simplified view of the feature model [23] of MobileMedia. The alternative features are just the types of media supported: photo, music, and/or video. Examples of core features are: create/delete media, label media, and view/play media. In addition, some optional features are: transfer photo via SMS, count and sort media, copy media and set favourites. The core features of MobileMedia are applicable to all the mobile devices that are J2ME enabled. The optional and alternative features are configurable on selected devices depending on the provided API support. MobileMedia was developed for a family of 4 brands of devices [29, 30], namely Nokia, Motorola, Siemens, and RIM.

3.1.1 Change Scenarios

As mentioned in Section 2.2, in the first phase of our investigation we designed and implemented a set of change scenarios. In the MobileMedia product line, a total of seven change scenarios were incorporated, which led to eight releases. Table 1 summarises changes made in each release. The scenarios comprise different types of changes involving mandatory, optional, and alternative features, as well as non-functional concerns. Table 1 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, assess the design stability of the product line.

Table 1. Summary of scenarios in MobileMedia

Release	Description	Type of Change
R1	MobilePhoto core [29, 30]	
R2	Exception handling included (in the AspectJ version, exception handling was implemented according to [13])	Inclusion of non-functional concern
R3	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
R4	New feature added to allow users to specify and view their favourite photos.	Inclusion of optional feature
R5	New feature added to allow users to keep multiple copies of photos	Inclusion of optional feature
R6	New feature added to send photo to other users by SMS	Inclusion of optional feature
R7	New feature added to store, play, and organise music. The management of photo (e.g. create, delete and label) was turned into an alternative feature. All extended functionalities (e.g. sorting, favourites and SMS transfer) were also provided	Changing of one mandatory feature into two alternatives
R8	New feature added to manage videos	Inclusion of alternative feature

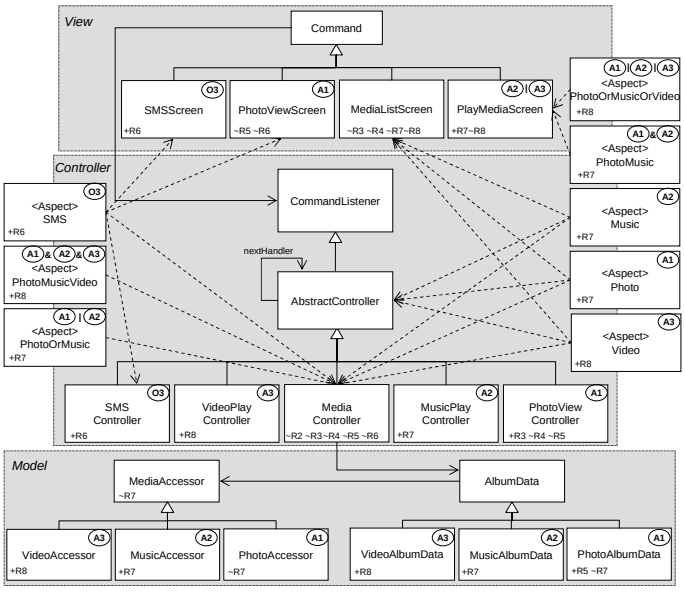

Figure 4. AO MobileMedia Architecture: marked a sub-set of modules affected by the scenarios

3.1.2 AO Architectural Design

Both Java and AspectJ designs of MobileMedia are mainly determined by the use of the Model-View-Controller (MVC) architectural pattern [5]. Figure 4 presents a representative partial view of the AspectJ architectural design. Due to space constraints, we do not present the Java architecture. The three grey boxes encompass classes that realise each of the three roles of the MVC pattern, namely model, view, and controller. The aspects do not belong to a specific role since they crosscut classes in more than one MVC role. Figure 4 also relates the design elements with the features in the feature model (Figure 3). This is done by the circles on the right top of the classes and aspects. For instance, the O3 on the top of the SMS aspect (Figure 4) indicates that this aspect contributes to the implementation of the feature marked with the O3 in the feature model (Figure 3). The sequence of Rs on the bottom of classes and aspects represent whether a class or aspect was added (+R) or changed (~R) during the implementation of a particular release. For instance, the `VideoPlayController` class was added during the implementation of the eighth release (+R8).

Taking the eighth release as example, it comprises three alternative features, PHOTO, VIDEO, and MUSIC, and the code of these features is realised by classes and aspects in the AspectJ version (Figure 4). On the other hand, in the Java implementation code related to each of these features is entirely realised by the view, controller, and model classes. Optional features are implemented in the same way. For instance, in the Java version, the SMS optional feature is implemented by the `SMSController` and `SMSScreen`. In the AspectJ solution, this feature is implemented by the same classes plus the `SMS` aspect.

3.2 BestLap

The second chosen SPL is a commercial project, called BestLap, developed by our industrial partner Meantime Mobile Creations[1] . BestLap is a racing car mobile game developed as a software product line where players have to achieve the pole position on a racing track. The score in the game is calculated on the basis of lap time and collected bonuses. Top scores are saved in the Hall of Fame and posted on the server, which shows ranking of multiple users with high scores. This product line has approximately 10 KLOC, can be deployed on 65 mobile devices, and has a total of 16 instances. Each instance is compatible for one family of devices that are grouped considering their compatibility to support the same game code. Although BestLap includes several mandatory, optional, and alternative features, this investigation focuses on the mandatory features SOUND, SCREEN, and GRAPHICS, which have further alternative sub features.

Change Scenarios. In the Bestlap product line, a total of four change scenarios were incorporated, which led to five releases. Table 2 summarises the changes that were made and their respective types. The scenarios encompassed the inclusion of an optional feature (ARENA) and the extension of alternative features. Each change scenario generates an instance for a family of devices. For example, the mandatory feature SCREEN was extended in release 2 to be supported by Motorola V300 and L6 devices. The purpose of these changes is to assess the design stability of BestLap through the SPL releases.

[1] http://www.meantime.com.br/en/

Table 2. Summary of scenarios in BestLap

Release	Description	Type of Change
R1	Features to support Motorola V220 devices	
R2	Extended screen size feature to support different sizes for Motorola V300 and L6 devices	Extension of the Screen Size feature
R3	Extended sound feature to support pre-allocating sound policy before playing the game for Nokia devices family	Extension of the features Sound, Graphics, and Screen
R4	Extended the Nokia shortcut keys for Siemens and Sony Ericsson devices	Extension of the features Keys, Sound, Graphics, and Screen
R5	New feature added to allow multiple users to post their respective lap time on the server	Inclusion of the Arena optional feature

4. CHANGE IMPACT ANALYSIS

Section 2.2 described how the assessment procedures were organised in three stages. This section presents the first stage where we quantitatively analyse to what extent each maintenance scenario entails change propagations in the target AO and non-AO product lines. This phase relies on a suite of typical change impact measures [28], such as number of components (classes and aspects) added or changed, number of added or modified lines of code (LOC), and so forth. The purpose of using these metrics is to quantitatively assess the propagation effects, when introducing or changing a specific feature, in terms of different granularities: components, operations, and LOC. Besides, the suite includes metrics to assess the changes in *pointcut* and *#ifdef* declarations which are the two main variability constructs of AOP and conditional compilation, respectively. The lower the change impact measures the more stable and resilient the design is to a certain change.

Table 3: Measures of change propagation in MobileMedia

			Mandatory		Optional			Alternative	
			R.2	R.3	R.4	R.5	R.6	R.7	R.8
Components	Added	OO	9	1	0	5	7	17	6
		AO	12	2	3	6	8	21	16
	Removed	OO	0	0	0	0	0	8	1
		AO	1	0	0	0	0	8	0
	Changed	OO	5	8	5	8	6	12	22
		AO	5	10	2	8	5	16	9
Operations	Added	OO	32	21	3	36	37	110	45
		AO	49	28	10	37	47	118	71
	Removed	OO	0	2	0	19	0	71	13
		AO	2	2	0	20	0	63	13
	Changed	OO	28	12	7	10	7	22	23
		AO	25	16	1	20	4	69	10
Lines of Code	Added	OO	273	162	51	521	443	1296	520
		AO	374	220	97	436	469	1188	729
	Removed	OO	10	8	2	205	9	897	120
		AO	57	16	0	278	16	663	111
	Changed	OO	29	29	7	18	2	67	24
		AO	29	40	1	70	8	222	12
IFDEF	Added	OO	0	11	9	15	10	75	53
	Removed	OO	0	0	0	0	0	8	6
	Changed	OO	0	0	0	0	3	20	11
PCs	Added	AO	43	6	7	2	7	19	26
	Removed	AO	0	0	0	0	0	0	5
	Changed	AO	0	8	0	16	2	50	2

Table 3 shows the change propagation in the MobileMedia design as it evolves through the change scenarios (Table 1). According to the similarities among the results observed in the measurement, we classified the scenarios into 3 groups: introduction of mandatory features (Section 4.1), optional features (Section 4.2),

and alternative features (Section 4.3). Section 4.4 presents a discussion of the stability of the variability mechanisms.

4.1 Including Mandatory Features

This section reports and discusses the results of the change impact in releases 2 and 3 together because they share the common characteristic of adding mandatory features: EXCEPTIONHANDLING and LABELMEDIA, respectively. AO solution usually does not cope with the introduction of mandatory features in this study since it is not targeted at modularising them. For instance, all components added in the Java release 2 (new exceptions classes) are also included in the AO one. The main difference is that, the AO version added additional aspects to handle the exceptions included in this release, which also implies more operations (advices handling the exceptions [6, 13]) and LOC. The same phenomenon happens in release 3, where the class `PhotoView Controller` was included in both AO and non-AO versions. Besides, AspectJ solution also added to this release a new aspect related to the incorporation of the optional feature SORTING. It is important to notice that a perfective refactoring in release 3 which changes a reference from `String` to `Image` in the `ImageData` class implies more changes in operations and LOC of the AO version due to the number of pointcuts relying on that reference.

4.2 Including Optional Features

Regarding the introduction of optional features (releases 4 to 6), the AspectJ solution introduced more components because new aspects have to be included in addition to the classes realising the features. Note that in the AO implementation of the SPL, aspects usually work as glue between the core and optional features [1, 4]. Operations are also included more in the AO solution due to the newly created advices. Despite the drawback of adding more elements, the AO solution often changes less components and operations. As a result, considering the Open-Closed principle [20], which states that 'software should be open for extension, but closed for modification', AO approach conforms more closely to this principle in scenarios which include optional features. For instance, the `PersistenceManager` component demanded changes in the Java release 4 in order to make favourite images persistent. On the other hand, the AO counterpart required no change in this component because the feature was implemented by new classes and the `Favourites` aspect.

A direct result of more operations and components included in the AspectJ version is the increase in LOC. However, it is interesting to notice that sometimes AspectJ overcame this problem by avoiding some replicated code. For instance, despite the fact that the AO version has more added components in release 5 (6 components in contrast to 5 in the Java version), the number of added LOC in the non-AO version is higher (19% more).

4.3 Including Alternative Features

The last two releases of MobileMedia introduced the alternative features MUSIC and VIDEO, respectively. However, release 7 turned a mandatory feature into alternative leading to a big impact in the change propagation metrics. Such impact is a result of changes applied to the core assets of the SPL and, therefore, release 7 affected all optional features which rely on this core. When changing a mandatory feature into two alternatives (release 7), AspectJ adds and changes more components/operations/LOC because all aspects rely on the points of intersection (join points) provided by the core. For instance, consider that a method in the core evolves to become optional in some concrete instances of the SPLs. In addition to changes in the class which contains this method, aspects that use this method as point of advising have to be changed as well.

Unlike release 7, release 8 added a new alternative feature to an existing set of alternatives and the AO version required fewer changes of components/operations/LOC. Note that in this situation, changes are not targeted at a mandatory feature and, therefore, they do not change the points which aspects rely on. The measures of scenario 8 are similar to the introduction of an optional feature. That means, more components and operations added, but less of them are changed. Again, AspectJ adheres better to the Open-Closed principle [20].

4.4 Stability of the Variability Mechanisms

Another point to consider is the fragility of pointcut expressions and conditional compilation declarations. In terms of added constructs, in all scenarios (except Scenario 1) it is necessary to add more *#ifdefs* in the Java version than pointcuts into aspects (Table 3). This situation is due to the pointcut concept which allows a selection of a set of join points in the code while the conditional compilation mechanism spread over each place where intersections between core and other features exist. Therefore, a new *#ifdef* construct has to be added to capture each specific point of interception between the core and an optional/alternative feature. The only exception is release 2 because exception handling does not require conditional compilation in the non-AO version since it is mandatory.

Depending on the evolution scenario, AspectJ pointcuts can be more fragile than conditional compilation. In release 7, for instance, it was necessary to refactor the name of a mandatory feature (PHOTO) in order to generalise it into two alternative features (PHOTO or MUSIC). In this case, every occurrence of this name had to be changed. Since certain aspects have several pointcuts relying on the syntactic match (e.g. names of methods and classes), this implies in many pointcuts being changed. On the other hand, *#ifdefs* do not need to be changed very often because they refer only to the feature name. In fact, conditional compilation tags had to be changed in releases 6 to 8 due to the sharing of code among more than one feature (see Figure 1).

5. MODULARITY ANALYSIS

This section presents the results for the second stage where we analyse the stability of the BestLap and MobileMedia product lines throughout the implemented changes. We used a metrics suite that quantified four fundamental modularity attributes, namely separation of concerns (Section 5.1), coupling, cohesion, and conciseness (Sections 5.2). Such metrics were chosen because they have already been used in several experimental studies and proved to be effective maintainability indicators (e.g. [7, 13-15]).

The metrics for coupling, cohesion, and size were defined based on classic OO metrics [8]; the original metrics definitions were extended to be applied in a paradigm-independent way, supporting the generation of comparable results. In addition, this suite introduces four new metrics for quantifying separation of concerns (SoC) [11, 25]. They measure the degree to which a single concern (feature, in the case of this study) in the system

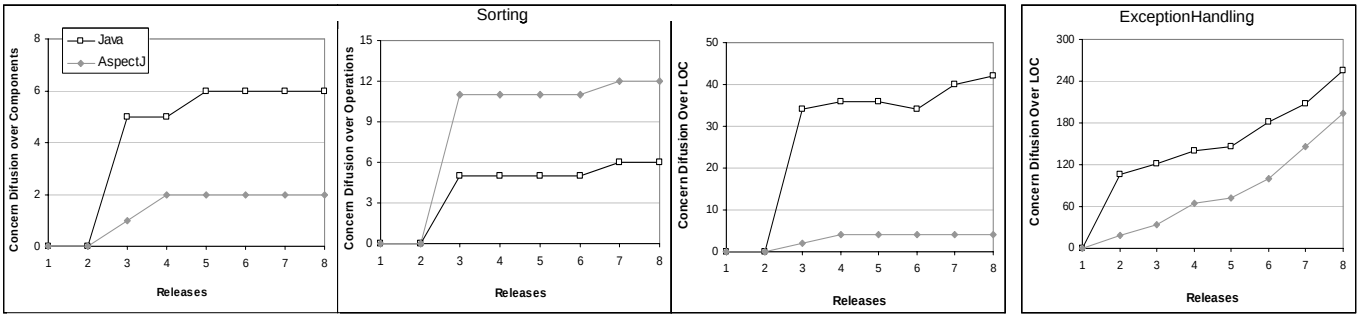

Figure 5. SoC metrics for SORTING and EXCEPTIONHANDLING in MobileMedia

maps to: (i) components (i.e. classes and aspects) – based on the metric Concern Diffusion over Components (CDC), (ii) operations (i.e. methods and advices) – based on the metric Concern Diffusion over Operations (CDO), and (iii) lines of code – based on the metric Concern Diffusion over Lines of Code (CDLOC). The majority of these metrics can be collected automatically by applying an existing measurement tool [12].

The SoC metrics require the manual 'shadowing' of the code, i.e. identifying which segment of code contributes to which feature in the SPLs. Although the mapping of features to the source code is not completely automated, it is facilitated with tool support [24]. This involved six post-graduate students (four of them not involved in the implementation phase of the study) grouped in three pairs. In circumstances when it was not clear which concern the segment contributes to, cross-discussions among all groups involved in the shadowing took place to reach a common agreement. For all the employed metrics, a lower value implies a better result. Detailed discussions about the metrics appear elsewhere [11, 14, 25].

5.1 Separation of Features

This section presents the measurement results for the SoC metrics. We analysed 15 features (12 from MobileMedia and 3 from BestLap) which include 5 optional, 6 alternative and 4 mandatory features. These were selected because optional and alternative features are the locus of variation in the SPLs and, therefore, they have to be well modularised. On the other hand, mandatory features need to be investigated in order to assess the impact of changes on the core. From the analysis of SoC measures, three groups of features naturally emerged with respect to which type of modularisation paradigm presents superior stability.

Group 1: AspectJ succeeds in features with no shared code. This group encompasses two optional features (SORTING and FAVOURITES), one alternative feature (GRAPHICS), and one mandatory feature (EXCEPTIONHANDLING). A common characteristic of all these features is that they do not share any piece of code with other features. Figure 5 shows examples of SoC metrics for two representative features of this group, namely SORTING and EXCEPTIONHANDLING. The AO solution of SORTING presents lower values and superior stability in terms of tangling (CDLOC) and scattering over components (CDC). The effectiveness of AO mechanisms to localise this kind of features is due to the ability to transfer the code in charge of realising the optional feature from classes to a set of dedicated classes and one or more glue aspects. 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 locally at the places where features intersect.

Although AO solutions are more stable, in some cases they require an increase of operations (CDO) to realise features of this category. For instance, the number of operations of SORTING (Figure 5) rises through the evolution of MobileMedia because (i) advices are created in order to mimic the behaviour of the feature when the join point is reached and (ii) new operations are created in the core classes to expose join points that aspects can capture. The AO solution of EXCEPTIONHANDLING also increases the CDO value because, unlike *try-catch* blocks in Java, each handler advice is counted as a new operation. Feature tangling tends to be very low and stable in this category (see CDLOC of SORTING in Figure 5) because every feature is realised by its individual set of aspects and classes. However, EXCEPTIONHANDLING does not follow this trend. Even though the AO implementation scales better than the OO one, its value for CDLOC rises at each new release. This unstable behaviour of CDLOC is a consequence of a design decision we have made to not extract every *try-catch* block to aspects. This decision stemmed from our previous knowledge that there are situations where aspectisation contributes negatively to the quality of exception handling code [6, 13]. Since we have adhered to the policy of using only the best design practices, we have aspectised only scenarios in which aspects are beneficial.

Group 2: Increased scattering of code-sharing features. Some features have not presented explicit superiority in either of the paradigms. These include three optional features (COPYMEDIA, SMS and CAPTUREMEDIA) and five alternatives features (SOUND, SCREENSIZE, PHOTO, MUSIC and VIDEO). Figure 6 (left side) shows the results of the SOUND feature as a representative of this group. As observed in the charts of CDC and CDLOC, both paradigms experience inverted result in terms of these metrics. This inversion occurs for two main reasons. First, all of those features share one or more slice of code with other features. For instance, Figure 1 (Section 2.1) depicts a scenario where SMS shares two distinct pieces of code with CAPTUREMEDIA. In general, the aspectisation process of this kind of sharing consists of creating a separate aspect to handle this common code (Figure 2). As a consequence, the number of components implementing those features (CDC) is higher in the AO version because each set of common code must be modularised in a separated aspect (unlike *#if* blocks which use just an *OR/AND* conditional operator). Second, as the features were modularised into aspects, the CDLOC metric is less affected on AO solutions since changes are localised in the initial modules which seem to cope well with the newly introduced scenarios.

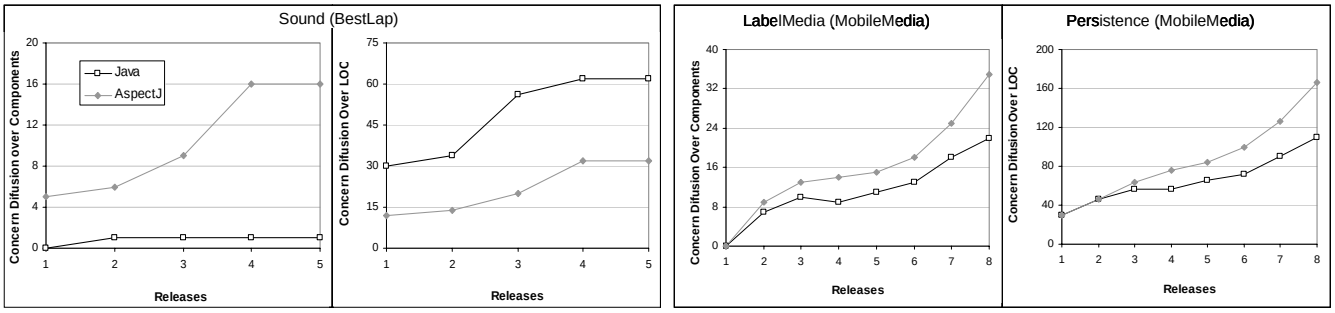

Figure 6. SoC metrics for SOUND (BestLap) and LABELMEDIA and PERSISTENCE (MobileMedia)

Group 3: AspectJ is harmful to modularity of mandatory features. Mandatory features and some widely-scoped concerns tended to present slightly superior design stability in the Java implementation of the product lines. These include, for instance, the LABELMEDIA feature as well as the concerns PERSISTENCE and CONTROLLER of MobileMedia. Figure 6 (right side) shows the metrics results for LABELMEDIA and PERSISTENCE as representatives of this group. We observe (Figure 6) that the modularisation of LABELMEDIA is more stable in the Java version, since this feature is spread over fewer components (CDC) in this solution. Besides, the difference increases throughout the releases due to the rising of CDC in the AspectJ solution. The CDLOC results for the PERSISTENCE concern show the same trend.

The features and concerns in this group constitute the core of the SPLs, and, therefore, were not aspectised – our strategy was to use aspects only for optional and alternative features. In addition, most of the optional and alternative features depend on the core features and concerns. For instance, PHOTO, MUSIC and VIDEO alternative features depend on LABELMEDIA and PERSISTENCE, once every photo, song or video must be labelled and persisted. Therefore, as new optional and alternative features are included over the different releases, the number of components that contains mandatory and non-functional concerns increases. Hence, the reason for this difference on modularity stability is that the number of components included over the releases is higher in the AspectJ version, as discussed in Section 4. As a consequence, the number of components where, for instance, LABELMEDIA and PERSISTENCE are, increases more in the AspectJ version than in the Java one. As a conclusion, the results of this group indicate that using aspects to modularise only optional and alternative features in the investigated product lines negatively impacted on the modularity of mandatory features.

5.2 Coupling, Cohesion and Size

The absolute values collected to the coupling, cohesion, and size metrics in our case studies have favoured the Java version for most of the evolution scenarios. Figure 7 illustrates the absolute values results for Coupling between Components (CBC) and Lack of Cohesion in Operations (LCOO) of MobileMedia, and for Vocabulary Size (VS) and Lines of Code (LOC) of BestLap. The increase of all metrics in the AspectJ solution is mainly due to the creation of the new aspects (VS in Figure 7). In fact, in some releases, it was observed that the difference for the collected metrics between the OO and AO versions was caused not only by the creation of new aspects but also because many of them are heterogeneous. A heterogeneous aspect affects multiple classes and respective join points in different ways by introducing different behaviour in each of them.

The use of aspects improved the modularisation of optional and alternative crosscutting features (Section 5.1). On the other hand, they caused an increase on coupling, cohesion, and size metrics. Some scenarios presented a slight difference between the Java and AspectJ solutions. For instance, Figure 7 shows a minor dissimilarity of LOC in favour of Java for the BestLap case study through all evolution scenarios. In addition, release 1 to release 6 of MobileMedia also presents a slight difference in the measurements of CBC and LCOO. For both product lines, though, we also observed a significant increase on the measurements in some specific releases. Figure 7 shows, for example, a significant increase in the CBC and LCOO metrics of MobileMedia considering scenarios 7 and 8. It happened mainly due to the AspectJ implementations difficulty of addressing different SPL configurations (specific combination of features). While the use of conditional compilation in the Java version allowed to codify

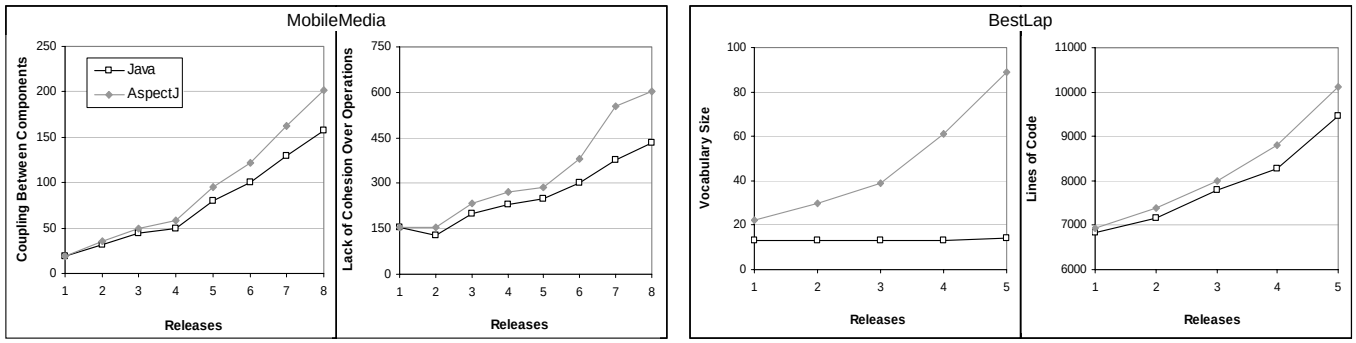

Figure 7. Coupling and cohesion of MobileMedia; and size of BestLap

all the SPL configurations using the AND and OR operators, the AO solution required the coding of different aspects representing different combinations of features, such as, `PhotoOrMusic` and `Photo(And)Music` aspects (Figure 4). This situation could be alleviated with more flexible constructs to define the order for applying aspects to the same join points in the AspectJ language.

6. FEATURE DEPENDENCY ANALYSIS

The analysis of the data gathered based on the change impact and modularity metrics (Sections 4 and 5) shows evidence that most of the features involved in MobileMedia and BestLap are scattered and tangled with each other over the product-line classes and aspects. For example, the aspect `PhotoMusicVideo` incorporates code related to PHOTO, MUSIC, and VIDEO (Figure 4). This section discusses how the features dependencies changed over the releases in the AspectJ and Java implementations. The goal is to observe how changes relative to a specific feature 'traversed the boundaries' of other feature implementations and/or generated new undesirable inter-feature dependencies.

6.1 Categories of Dependency

In order to support such dependency analysis, we have observed different categories of feature dependency. In the context of the studied SPLs, we considered two different ways in which the features interact with each other: interlacing and overlapping. The classification of feature dependencies is based on how the feature realisations share elements in the implementation artefacts. A similar classification has already been defined and exploited in previous studies [7, 13].

Interlacing. This dependency occurs when the implementation of two features, F1 and F2, have one or more components (or operations) in common [7]. We classify a dependency as *component-level interlacing* if F1 and F2 share one or more components (class or aspect). Similarly, we classify as *operation-level interlacing* if F1 and F2 share one or more operations (methods or advices) in a shared component. Both cases produce feature tangling, but at different levels of granularity.

Overlapping. This kind of dependency occurs when the implementations of features F1 and F2 share one or more statements, attributes, entire methods, or entire components [7]. This dependency style is different from interlacing because here the shared elements entirely contribute to both features rather than being disjoint. Depending on the kind of elements participating in the dependency, it can be classified as *component overlapping*, *operation overlapping*, or *lines of code overlapping*.

6.2 Stability of Pair-wise Dependencies

This section focuses on an analysis of stability of each pair of features through the releases. We verify for each feature the number of components shared with other features (component-level interlacing). This kind of analysis supports assessment of feature modularisation and stability because it shows whether the inter-feature coupling drops with the software evolution or not. According to similar results obtained from the dependency analysis, the investigated pairs of interacting features can be classified into two groups: (i) dependency between two mandatory features; and (ii) dependency where at least one of the participant features is optional or alternative.

As a representative of the first group, Figure 8 depicts the measures for the dependency between the CONTROLLER and LABELMEDIA features of MobileMedia. We can grasp from this figure that the number of components with both features increases throughout the releases. However, the AspectJ version presents inferior stability, since the amount of dependency increases faster in this version. This means that the number of points where changes to a mandatory feature can potentially impact other mandatory features tend to be higher in the AspectJ version. This occurs because the analysed mandatory features were not aspectised. As a result, they are spread over the components that implement optional and alternative features, whose quantity is higher in the AspectJ version.

The results about the second group show that pair-wise dependencies involving at least one optional or alternative feature are stable in both AspectJ and Java versions. Figure 8 shows the results for the dependency between two alternative features of BestLap: GRAPHICS and SCREEN. We can see that the number of shared components presents minor variation over the releases. However, the amount of shared components is lower in the AspectJ version. This occurs because GRAPHICS and SCREEN were aspectised in the AspectJ version and, as a result, they are scattered over less components than in the Java version. In the AspectJ version, the two components which mixed these features (Figure 8) modularise shared code of both features (overlapping) that could not be placed in distinct aspects.

In addition to the component interlacing, we also analyse two other categories of dependency: operation interlacing and LOC overlapping. Figure 8 depicts the results of operation interlacing and LOC overlapping for one pair of features: SORTING vs. LABELMEDIA. This pair is also a representative of optional with mandatory dependency. As discussed before, dependencies of this category are more stable in the AO version, as well as it shares

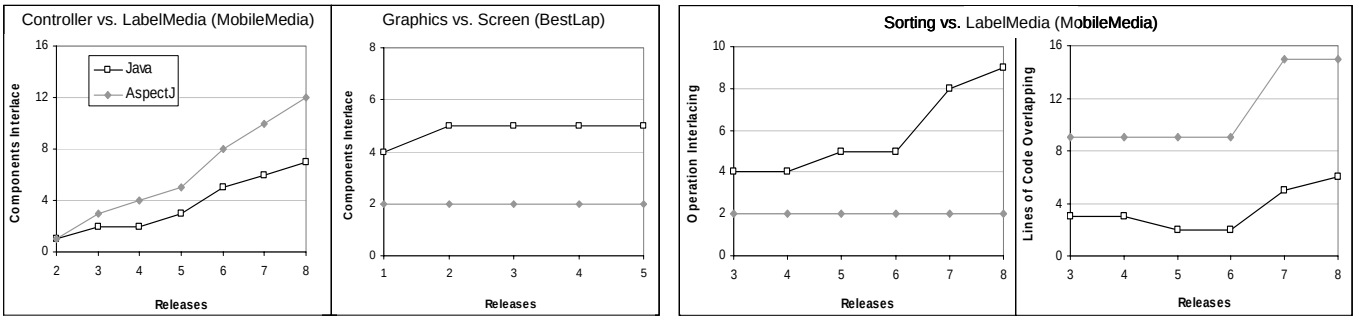

Figure 8. Examples of pair-wise dependencies between mandatory, optional, and alternative features

fewer components and operations. However, although the AO version concentrates the dependency in a few places (lower value of operation interlacing in Figure 8), this dependency is stronger than in the Java solution as highlighted by the higher value of LOC overlapping. In other words, feature boundaries are wider (more operation and component interlacing) in the Java implementation and deeper (more overlapping) in the AO solution of dependencies involving optional or alternative features.

6.3 Scalability in Complex Dependencies

The previous subsection discussed how the aspectisation of pair-wise dependencies impacts different modularity attributes. This section discusses how conditional compilation and AOP scaled in dependencies involving a greater number of features. Figure 9 describes some representative features organised in terms of releases and categories of dependency (Section 6.1). Each feature has different number of bars since they were introduced in different releases. The left-hand side of Figure 9 illustrates the inability of conditional compilation mechanisms to scale when complex dependencies among features occur. These charts support the analysis of impact on a set of features by introducing other new features. For instance, the introduction of two alternative features, MUSIC and VIDEO, in releases 7 and 8 increased significantly the code overlapping among these features and also optional features, such as COPYMEDIA and SMS. This behaviour in some way was expected, since alternative features tend to reuse parts of existing code to implement their functionality. However, surprisingly the introduction of one optional feature also affected other optional features. This occurred, for instance, in the sixth release when the introduction of SMS increased code overlapping of this feature with COPYMEDIA. This higher overlapping represents in practice the existence of more explicit dependency between such optional features.

We observed that the AO implementations usually scale well for all kinds of interlacing dependencies. AspectJ employs inter-type declarations to address component interlacing and pointcut-advices to deal with operation interlacing. The right-hand side of Figure 9 shows that no kind of interlacing is observed in all features analysed. However, the presence of overlapping can hinder a smooth dependency process and, sometimes, negatively affect the features being composed. This occurs because the aspectisation of some specific scenarios with strong coupling between the features can violate modularity (Section 5.2). For example, the code described in Figure 2 was totally dedicated to the COPYMEDIA feature by the fifth release, but this code is moved to a new component (aspect) in release 7 because SMS and CAPTUREMEDIA depend on it. Again, AspectJ presents the same recurrent problem described above in which the introduction of one optional feature affects another optional feature. In addition, Figure 9 is useful to support the findings of Section 5 which claimed that AspectJ succeeds in features with no shared code, i.e., no overlapping dependency.

7. RELATED WORK AND STUDY CONSTRAINTS

Recent research work has explored the use of AOP in the development or refactoring of SPLs [1, 3, 16]. Most of these investigations, however, only concentrate on the qualitative analysis of the features aspectisation process. For instance, Kästner et al. [16] presented a case study on refactoring the Berkeley DB system into a SPL. The authors reported several limitations on the modularisation of features when using AspectJ, such as the increase of coupling between aspects and classes due to the strong dependency of pointcuts on implementation details of the base code. In our work, we also found out some of the limitations reported by Kästner et al. [16]. In addition, we (i) categorised evolution scenarios in which AspectJ succeeds or not (Sections 4 and 5) and (ii) investigated the stability and scalability of this language to address feature dependencies (Section 6).

There are also investigations on the development of SPLs focusing on the decomposition of architectures into features [3, 4, 22]. Mezini and Ostermann [22] identified that feature-oriented approaches (FOAs) are only capable of modularising hierarchical features. They propose CaesarJ [21] that combines ideas from both AOP and FOAs to provide support to manage variability in SPLs. More recently, Apel and Batory [4] have proposed the Aspectual Mixin Layers [3] 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 a SPL implementation. Their study, however, did not consider a significant suite of software metrics and did not address SPL evolution and stability. Greenwood et al. [15] used

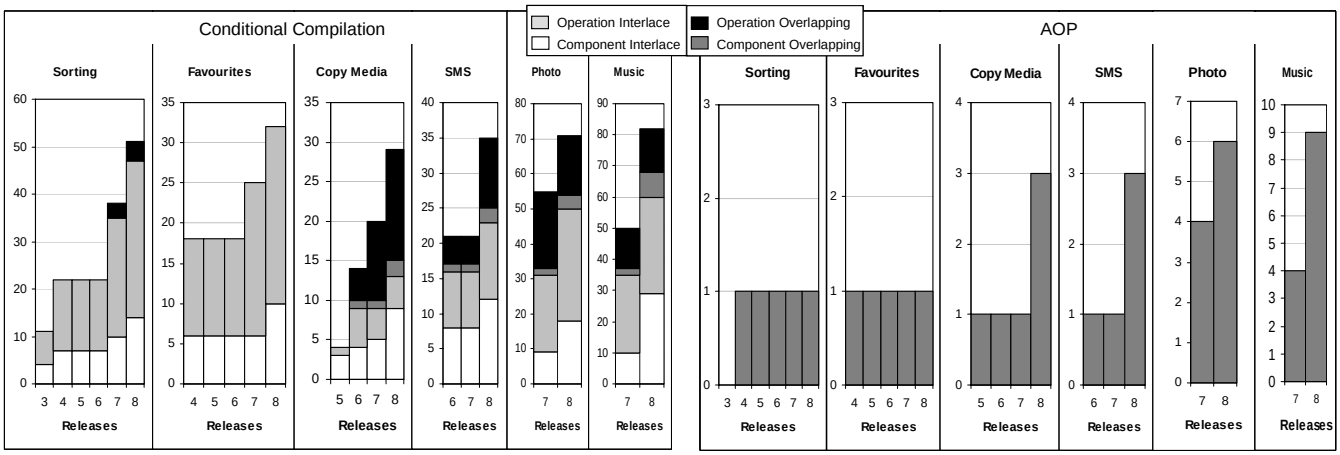

Figure 9. Scalability of conditional compilation and AOP in complex dependencies

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.

Regarding our study constraints, the applicability and usefulness of some specific metrics used in this study, such as the cohesion one, has often been questioned. We accept the criticism of such metrics. However, it is important to consider the results gathered from all metrics rather than just one metric in particular. In fact, the multi-dimensional analysis allows us to grasp which measurement outliers are significant and which are not. The use of AspectJ could also be pointed out as a constraint in our experimental evaluation, since it is not the only existing AOP language. However, we have chosen AspectJ because it is a stable and widely-used AOP language. Besides, most of the previous studies about AOP and product lines used AspectJ as well. Therefore, adopting this language allowed us to compare our results with previous case studies.

8. CONCLUDING REMARKS

The transfer of aspect-oriented technologies to the development of SPLs largely dependents on our ability to empirically understand its positive and negative effects through design changes. Designs of SPLs are often the target of unanticipated changes and, as a result, incremental development has been largely adopted in realistic SPLs development [1, 18]. This study evolved two real-life SPLs in order to assess the capabilities of AOP mechanisms to provide SPL modularity and stability in the presence of realistic change tasks. Such evaluation included three complementary analyses: implementation modularity, change propagation and feature dependency.

From this analysis we discovered a number of interesting outcomes. Firstly, the AO implementations of the studied SPLs tend to have more stable design particularly when a change targets optional and alternative features (Section 4.2 and 5.1). This indicates that aspectual decompositions are superior in those situations, especially when considering the Open-Closed principle [20]. However, AO mechanisms do not cope with the introduction of widely-scoped mandatory features or when changing a mandatory feature into alternatives (Section 4.1 and 5.1). Furthermore, such mechanisms usually scale well for dependencies that do not involve shared code, although AspectJ faces difficulties to address different SPL configurations.

9. ACKNOWLEDGEMENTS

This work is supported in part by the European Commission grant IST-33710: Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE), grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe). We express our thanks to Trevor Young, Vander Alves, and Meantime for providing us with the product lines. Eduardo, Claudio, and Fabiano are supported by CAPES - Brazil.

10. REFERENCES

[1] Alves, V. et al. Extracting and Evolving code in Product Lines with Aspect-Oriented Programming. Trans. on AOSD, pp. 118-144, 2007.

[2] Alves, V. Implementing Software Product Line Adoption Strategies, Ph.D. thesis. Federal University of Pernambuco, March 2007.

[3] Apel, S. et al. Aspectual Mixin Layers: Aspects and Features in Concert. Proceedings of ICSE'06, Shanghai, China, 2006.

[4] Apel, S. and Batory, D. When to Use Features and Aspects? A Case Study. Proceedings of GPCE, Portland, Oregon, 2006.

[5] Buschmann, F. et al. Pattern-Oriented Software Architecture: a System of Patterns. John Wiley & Sons, Inc. 1996.

[6] Cacho, N. et al. EJFlow: Taming Exceptional Control Flow in Aspect-Oriented Programming. Proc. of AOSD.08, Belgium, 2008.

[7] Cacho, N. et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. Proc. of AOSD, Germany, 2006.

[8] Chidamber, S. and Kemerer, C. A Metrics Suite for Object Oriented Design. IEEE Trans. on Soft. Eng. (TSE), pp. 476-493, 1994.

[9] Clements, P. and Northrop, L. Software Product Lines: Practices and Patterns. Addison-Wesley, 2002.

[10] Evolving Software Product Lines with Aspects. http://www.lancs.ac.uk/postgrad/figueire/spl/icse08/

[11] 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.

[12] Figueiredo, E., Garcia, A., and Lucena, C. AJATO: An AspectJ Assessment Tool. Proceedings of ECOOP (demo), Nantes, 2006.

[13] Filho, F. et al. Exceptions and Aspects: The Devil is in the Details. Proc. of Int'l Symp. on Foundations of Software Eng. (FSE), 2006.

[14] Garcia, A. et al. Modularizing Design Patterns with Aspects: A Quantitative Study. Transactions on AOSD, 1, pp. 36-74, 2006.

[15] Greenwood, P. et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. Proc. of ECOOP, Berlin, 2007.

[16] Kästner, C., Apel, S. and Batory, D. A Case Study Implementing Features using AspectJ. Proc. of Int'l SPL Conference (SPLC), 2007.

[17] Kiczales, G. et al. Aspect-Oriented Programming. Proc. of ECOOP, LNCS 1241, Springer, pp. 220-242, 1997.

[18] Krueger, C. Easing the Transition to Software Mass Customization. Proc. of 4th Int'l workshop on Software Product Family Engineering, pp. 282–293, Bilbao, 2001.

[19] Kulesza, U. et al. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming. Proceedings of Int'l Conference on Software Reuse (ICSR), Torino, 2006.

[20] Meyer, B. Object-Oriented Software Construction, 1st ed. Prentice-Hall, Englewood Cliffs, 1988.

[21] Mezini, M. and Ostermann, K. Conquering Aspects with Caesar. Proc. of AOSD, pp. 90-99, Boston, USA, 2003.

[22] Mezini, M. and Ostermann, K. Variability Management with Feature-Oriented Programming and Aspects. Proceedings of FSE, pp.127-136, 2004.

[23] Pohl, K., Böckle, G., and Linden, F. J. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc, 2005.

[24] Robillard, M. and Murphy, G. Representing concerns in source code. Trans. on Software Eng. and Methodology (TOSEM), 16(1), 2007.

[25] Sant'Anna, C. et al. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. Proc. of Brazilian Symposium. on Software Engineering (SBES), pp. 19-34, 2003.

[26] Smaragdakis, Y. and Batory, D. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM TOSEM, 11(2), 2002.

[27] The AspectJ Project. http://eclipse.org/aspectj/.

[28] Yau, S. and Collofello, S. Design Stability Measures for Software Maintenance. Trans. on Softw. Engineering, 11(9), p. 849-856, 1985.

[29] Young, T. Using AspectJ to Build a Software Product Line for Mobile Devices. MSc dissertation, Univ. of British Columbia, 2005.

[30] Young, T. and Murphy, G. Using AspectJ to Build a Product Line for Mobile Devices. Proceedings of AOSD (demo), Chicago, 2005.