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

Eduardo Figueiredo¹, Nelio Cacho¹, Claudio Sant’Anna¹, Mario Monteiro², Uira Kulesza¹, Alessandro Garcia¹, Sergio Soares², Fabiano Ferrari¹, Safoora Khan¹, Fernando Filho², Francisco Dantas²

¹Computing Department, Lancaster University, United Kingdom
²Computer Science Department, Pernambuco State University, Brazil
{figueire, cacho, santanna, kulesza, garcia, ferrari, khan}@lancaster.ac.uk,
{mario, sergio, fernando, francisco}@dsc.upe.br

ABSTRACT

Software product lines (SPLs) enable modular, large-scale reuse through a software architecture addressing multiple core and varying features. To reap 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. However, there is no empirical evidence on its efficacy to prolong design stability of product lines in realistic development scenarios. This paper reports a quantitative study that evolves two SPLs to assess various facets of design stability of aspect-oriented implementations. Our investigation focused upon a multi-perspective analysis of the evolving product lines in terms of modularity, change propagation, and feature interaction.

General Terms
Measurement, Design, Experimentation.

Keywords
Product lines, aspect-oriented programming, empirical studies, metrics.

1. INTRODUCTION

Software product lines (SPLs) [8, 22] represent a common and important 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 real-life SPLs is often incremental and gradually evolves to cope with new stakeholders’ needs [1, 8, 17]. However, their longevity is highly dependent on the ability of the implementation-level variability mechanisms to sustain the stability [1, 3, 4, 8, 15] of the modular structure of the core and variable features 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 unpluggability of the optional code [1, 15]. Many authors [1, 15, 21] advocate that aspect-oriented programming (AOP) [16] 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].

Some works have started to explore the use of AOP to improve the isolation of specific features in designs of frameworks [18] and product lines [15, 20]. However, none of them has analyzed the impact of AOP on heterogeneous evolution scenarios of program families. Most of them are either of methodological nature [18] or are not focused on objectively assessing the role of AOP on sustaining core design's modularity and reducing observance of ripple effects across the core architecture and optional modules [15, 21]. In this context, it is important to systematically verify the suitability of AOP [16] for designing stable product-line architectures, especially when compared to mainstream variability mechanisms, such as object-oriented (OO) mechanisms and conditional compilation. Even though there are a number of emerging academic programming techniques for supporting product line development [20], 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 requirements of two heterogeneous product lines, called MobileMedia [28] 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 the positive and negative impacts 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 modularity and stability in the presence of realistic change tasks. Such tasks emerged from the analysis of the evolution history of those product lines in realistic development settings. The design stability evaluation of the Java and AspectJ versions were based on three conventional metrics suites for modularity [24], change impact [27], and feature interaction [13].
2. STUDY SETTING
This section describes the configuration of our study. Section 2.1 briefly explains the two variability mechanisms 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: conditional compilation and AOP [16]. We chose AspectJ [26] in order 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 [20]. Conditional compilation 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 encoded code of two features (includeSmsFeature and captureMedia) must be compiled.

```java
01 // if define copyMedia
02 private void processMediaData(String mediaName, String albumName) {
03     MediaData mediaData = null;
04     // if includeSmsFeature || captureMedia
05     byte[] mediaByte = getCapturedMedia();
06     if (mediaByte == null) {
07         // endif
08         mediaData = getAlbumData().getMediaInfo(mediaName);
09     } else {
10     // if includeSmsFeature || captureMedia
11         getAlbumData().addMediaData(mediaData, albumName);
12     }
13 }
14    // 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, 15]. For instance, the piece of code enclosed by `if` and `endif` described in Figure 1 belongs to an 
optional feature and AOP separates it using `pointcuts`, advice 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 `includeSmsFeature` and `captureMedia` features is now modularized in a unique place (around advice).

```java
01 public aspect SMSOnCaptureMedia {
02     pointcut processMediaData (...) : execution(*
03     .processMediaData(...)) & this(...) & args(...));
04     MediaData mediaData = null;
05     byte[] mediaByte = controller.getCapturedMedia();
06     if (mediaByte == null) {
07         MediaData proceed(controller, mediaName, albumName);
08     } else {
09         controller.getAlbumData().addMediaData(mediaName, mediaByte, albumName);
10         return mediaData;
11     }
12 }
```

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 realization of SPL change scenarios, (2) the alignment of SPL versions, and (3) the quantitative and qualitative assessments of the SPL versions and the successive releases. In the first phase, an independent group of five post-graduate students was responsible for implementing the successive evolution scenarios of the two SPLs: BestLap [2] and MobileMedia [28] (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 refactoring the previous release of the respective SPL, i.e., releases X+1 in Java and AspectJ represent respectively the implementation of evolution scenarios over the respective releases X. For example, AspectJ release 2 has evolved from AspectJ release 1. Best-of-breed design practices [5, 8] were applied throughout the generation 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.

Generation of the SPL Releases. We generated eight releases of the MobileMedia (Section 3.1) and five of Best Lap (Section 3.2) were generated in the first phase. The MobileMedia releases are available from [9]. Due to copyright constraints we cannot publish the BestLap code, but this is not a problem as we found representative of the same problems and relevant issues in both SPLs; when there is an exception for a particular SPL, we explicitly mention that in the following sections. Both MobileMedia and BestLap have been successfully used in other studies involving modular design of SPLs [2, 28], and so provided a solid foundation for the study presented in this paper. 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 3) in order to assure that coding styles and implemented functionality were exactly the same. Moreover, the implementations followed the same design decisions in that best practices [5, 8] 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 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.

3. TARGET PRODUCT LINES
For comprehensive investigation the initial decision entailed the selection of the targeted product lines. The two chosen SPLs are BestLap [2] and MobileMedia [28]. They were selected due to several reasons. First, we believe these software SPLs are representatives for the mobile devices domain, since they have (i) several variability points related to the 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 implementations of both SPLs were available facilitating the analysis of the two variability mechanisms under investigation: conditional compilation and AO techniques.

3.1 MobileMedia
MobileMedia is a SPL for applications that manipulate photo, music, and video on mobile devices, such as mobile phones. It was developed based on a previous SPL called MobilePhoto [28], 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 of MobileMedia. The core features of the MobileMedia are: create/delete media (photo, music or video), label media, and view/play media. The alternative features are just the types of media supported: photo, music, and/or video. Finally, the optional features are: send photo via SMS, count and sort media, copy media and set favourite media. The core features of MobileMedia are applicable to all the mobile phones devices that are J2ME enabled. The optional and alternative features are configurable on selected mobile phones depending on the API support they provided. MobileMedia was developed for a family of 4 brands of devices, namely Nokia, Motorola, Siemens, and RIM.

Figure 3. Simplified MobileMedia feature model.

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 in this study, which lead 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 assess the design stability in terms of how the implementation of feature boundaries and their dependencies evolved through the SPL releases.

<table>
<thead>
<tr>
<th>Release</th>
<th>Description</th>
<th>Type of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>MobilePhoto core [28]</td>
<td>Inclusion of alternative feature</td>
</tr>
<tr>
<td>R2</td>
<td>Exception handling included (in the AspectJ version, exception handling was implemented according to [11])</td>
<td>Inclusion of non-functional concern</td>
</tr>
<tr>
<td>R3</td>
<td>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.</td>
<td>Inclusion of optional and mandatory features</td>
</tr>
<tr>
<td>R4</td>
<td>New feature added to allow users to specify and view their favourite photos.</td>
<td>Inclusion of optional feature</td>
</tr>
<tr>
<td>R5</td>
<td>New feature added to allow users to keep multiple copies of photos.</td>
<td>Inclusion of optional feature</td>
</tr>
<tr>
<td>R6</td>
<td>New feature added to send photo to other users by SMS.</td>
<td>Inclusion of optional feature</td>
</tr>
<tr>
<td>R7</td>
<td>New feature added to store, play, and organize 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.</td>
<td>Changing of one mandatory feature into two alternatives.</td>
</tr>
<tr>
<td>R8</td>
<td>New feature added to manage videos.</td>
<td>Inclusion of alternative feature</td>
</tr>
</tbody>
</table>

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

3.1.2 AO Architectural Design
Both Java and AspectJ designs of the MobileMedia SPL 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 version 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 of the MVC roles. 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, PlayVideoController class was added during the implementation of release eight (+R8).

The eighth release of the MobileMedia product line comprises three alternative features: PHOTO, VIDEO, and MUSIC. In the Java implementation the code related to each of these features is entirely modularized by the view, controller, and model classes. In the AspectJ version, code of these features is modularized by classes and aspects. 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 Best Lap

The second chosen SPL is a commercial project, called BestLap, developed by our industrial partner Meantime Mobile Creations1. It is a racing car mobile game developed as a software product line and has approximately 10 KLOC. The game can be deployed on 65 mobile devices and has a total of 16 instances. Each game instance is compatible for one family of devices that are grouped logically considering their compatibility to support the same game code.

BestLap players have to achieve the pole position on a racing track by achieving the best time in trial mini games. The score for BestLap is calculated on the basis of lap time and bonuses collected during the game. The highest scores are saved in the high score table and posted on the server, which shows ranking of multiple users with high scores. For this investigation the mandatory features of our interests are S OUND, SCREEN, and GRAPHICS, which have further sub feature of varying optionality. (Due to copyright restriction we do not discuss the feature model in details).

3.2.1 Change Scenario

In the Bestlap product line, a total of four change scenarios were incorporated, which lead to five releases. Table 2 summarises the changes that were made and types of change each release encompassed. The scenarios included different type of changes, which were extension and inclusion of features. Each change scenario generates an instance for a devices family. For example, mandatory feature SCREEN was extended to support Bestlap game for screen sizes supported by Motorola V300 and L6 devices family. The purpose of these changes is to assess the design stability through the SPL releases.

4. CHANGE IMPACT ANALYSIS

We described in Section 2.2 how the assessment procedures were organized in three stages. This section describes the first stage where we quantitatively analyze to what extent each maintenance scenario entails change propagations in the target AO product lines. This phase relies on a suite of typical change impact measures [27], such as number of components (aspects/classes) added or changed, number of added or modified 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 levels of abstraction: components, operations, and lines of code. 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>
<thead>
<tr>
<th>Release</th>
<th>Description</th>
<th>Type of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Features to support Motorola V220 devices.</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>Extended screen size feature to support different sizes for Motorola V300 and L6 devices.</td>
<td>Extension of feature screen size</td>
</tr>
<tr>
<td>R3</td>
<td>Extended sound feature to support pre-allocating sound policy before playing the game for Nokia devices family.</td>
<td>Extension of feature sound, graphics, and screen</td>
</tr>
<tr>
<td>R4</td>
<td>Extended the Nokia shortcut keys for Siemens and Sony Ericsson devices.</td>
<td>Extension of feature keys, sound, graphics, and screen</td>
</tr>
<tr>
<td>R5</td>
<td>New feature added to allow multiple users to post their respective lap time on the server.</td>
<td>Inclusion of optional feature Arena</td>
</tr>
</tbody>
</table>

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), introduction of optional features (Section 4.2), and introduction of 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 LABEL.MEDIA, respectively. AO solution usually does not cope with the introduction of mandatory features in this study since it is not targeted at modularizing 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 [11]) and LOC. The same phenomenon happens in release 3, where the class 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 introduced in addition to the classes realizing the features. Note that in the AO implementation of SPLs 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 changed less components and operations. As a result, considering the Open-Closed Principle [19], 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 component PersistenceManager needs to be changed in release 4 of the Java version in order to make favourite images persistent. On the other hand, the AO version requires no change in this component because this feature is implemented by 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 AO version has more added component 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. Differently from previous scenarios, this release changed the core assets of the SPL and, therefore, such a scenario might affect 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 provided by the core (joinpoints). For instance, consider that a method in the core might not be there in some concrete instances of the SPLs. In addition to the changes in the class which contains the method, aspects that use this method as point of advising have to be changed as well.

However, in the release 8, we added a new alternative feature to an existing set of alternative features and the AO version required the change of less 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 metrics results 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 to the Open-Closed Principle [19] better.

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 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 generalize 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 in unexpected ways. On the other hand, #ifdefs do not need to be changed very often because they refer only to the feature name. In fact, when implementing Scenarios 5 and 7, additional conditional compilation tags had to be added due to the sharing of code among more than one feature (Figure 1).

5. MODULARITY ANALYSIS
This section presents the results for the second stage where we analyze their stability throughout the implemented changes using 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 proven to be effective quality indicators (e.g. [6, 11, 12, 13]).

The metrics for coupling, cohesion, and size were defined based on classic OO metrics [7]; 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). They measure the degree to which a single concern (feature, in the case of this study) in the system maps to: (i) components (i.e. classes and aspects) – based on the metric Concern Diffusion over Components (CDC), (ii) operations (i.e. methods and advice) – 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 [10].

The separation of concern metrics requires the manual ‘shadowing’ of the code, i.e. identifying which segment of code contributed to which feature in the SPL. Although the mapping of features to the source code is not completely automated, it is facilitated with tool support [23]. This involved six post-graduate student (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 contributed 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 are out of the scope of this work and appear elsewhere [12, 24].

5.1 Separation of Features
This section presents the measurement results for the separation of concern metrics. We analyzed 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 modularized. 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 metric results, three distinct groups of features naturally emerged, with respect to which type of modularization 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. 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 localize this kind of feature is due to the ability to transfer the code in charge of realizing 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 ifndef /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 realize features of this category. For instance, the number of operations of SORTING (Figure 5) rises through the evolution of MobileMedia because (i) advice are created in order to mimic the behaviour of the feature when the pointcut 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 realized 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 not extract every try-catch block to aspects. This decision stemmed from our previous knowledge that there are situations where aspectization contributes negatively to the quality of exception handling code [11]. Since we have adhered to the policy of using only the best design practices, we have aspected 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 3 optional features (COPYMEDIA, SMSTRANSFER and CAPTUREMEDIA) and 5 alternatives features (SOUND, SCREENSIZE, PHOTO, MUSIC and VIDEO). Figure 6 shows the results of the SOUND feature as a representative of this group. As can be observed in the charts of Concern Diffusion over Components (CDC) and Concern Diffusion over Lines of Code (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 SMSTRANSFER shares two distinct pieces of code with CAPTUREMEDIA. In general, the aspectization process of this kind of sharing consists of creating a separate aspect to handle this common code (Figure 2). As a consequence, the CDC value is

![Figure 5. SoC metrics for SORTING and EXCEPTIONHANDLING.](image-url)
higher in the AO version because each set of common code must be modularized in a separated aspect (unlike \#if blocks which use just an OR/AND conditional operator). Second, as the features were modularized into aspects, the CDLOC metric is less affected on AO implementations as the initial modules seem to cope well with newly introduced scenarios and the changes are localized in these modules.

**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 both product lines. These include, for instance, the LABELMEDIA feature as well as the concerns PERSISTENCE and CONTROLLER of MobileMedia. Figure 6 shows the metrics results for LABELMEDIA and PERSISTENCE as representatives of this group. As can be observed in Figure 6, the modularisation of LABELMEDIA is more stable in the Java version, since this feature is spread over fewer components (CDC) in this version. Besides, the difference increases throughout the releases due to the rising of CDC in the AspectJ solution. The Concern Diffusion over Lines of Code (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 since the strategy used was to use aspects only for the 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 component 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 modularize 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 size, cohesion and coupling metrics in our case studies have favoured the Java version for most of the evolution scenarios. Figure 7 illustrates the absolute values results for the CBC and LCOO metrics for the Mobile Media, and for the VS and LOC metrics for the Best Lap. As we can see, all the graphics show that the AO implementation has caused the increase of these metrics through the different releases. It is mainly due to the implementation of the different aspects (see in Figure 7, the VS metric for the Best Lap case study) that improve the management of variability by modularizing the new optional and alternative features of each SPL.

It was observed that the slight difference in some releases for the collected coupling, cohesion and size metrics between the OO and AO versions of each SPL was caused not only by the creation of new aspects but also because many of them are heterogeneous. A heterogeneous aspect is one that affects multiple classes and join points in different ways by introducing different behaviour in each of them. Thus, these aspects help to modularize the SPL optional and alternative crosscutting features, but on the other hand they inevitably cause a slightly increase on the CBC, LCOO and LOC metric. Figure 7 shows, for instance, the slight difference of the LOC metric for the Best Lap case study along the evolution scenarios. Another example, illustrated in Figure 7, is the collected values for the CBC and LCOO metrics from

**Figure 7. Coupling, cohesion and size of the Mobile Media.**
release 1 to release 6 of the Mobile Media case study.

For both case studies, we also observed a significant increase on the metric values for some specific release. Figure 7 shows, for example, there was a significant increase in the CBC and LCOO metrics of the MobileMedia SPL considering scenarios 7 and 8. It happened mainly due to the AspectJ implementations difficulty of addressing different SPL configurations (specific combination of features). While in the conditional-compilation version, the use of conditional compilation allowed to codify all the SPL configurations using the AND and OR operators, the AO version required the codification of different aspects representing different combinations of features (a SPL configuration), such as, PhotoOrMusic and PhotoAndMusic aspects (Fig. 4). This situation could be alleviated if AspectJ provided more flexible constructs to define the order for applying aspects to the same join points.

6. FEATURE INTERACTION ANALYSIS

The analysis of the data gathered based on the change impact and modularity metrics (Sections 4 and 5) makes it evident 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 interactions 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 concern implementations and/or generated new undesirable inter-concern dependencies.

6.1 Categories of Interaction

In order to support such interaction analysis, we have observed different categories of feature interaction. 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 interactions is based on how the feature realizations share elements in the implementation artefacts. It has already been defined and exploited in previous studies [6, 11].

Interlacing. This interaction occurs when the implementation of two features, F1 and F2, have one or more components (or operations) in common [6]. We classify an interaction 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 abstraction.

Overlapping. This kind of interaction occurs when the implementations of features F1 and F2 share one or more statements, attributes, entire methods, or entire components [6]. 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 interaction, it can be classified as component overlapping, operation overlapping, or lines of code overlapping.

6.2 Stability of Pair-wise Interactions

In this section we focus on analysis of the stability of each pair of features through the implementations. We verify for each feature the number of components shared with other features (component-level interlacing). This kind of analysis supports assessment of concern modularization and stability because it shows whether the inter-concern coupling drops with the software evolution or not. According to similar results obtained from the interaction analysis, the investigated pairs of interacting features can be classified into two groups: (i) interaction between two mandatory features; and (ii) interactions 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 interaction between the features CONTROLLER and LABEL.MEDIA 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 interactions increases faster in this version. This means that the number the points where changes to a mandatory feature can potentially impact other mandatory feature tend to be higher in the AspectJ version. This occurs because the analysed mandatory features were not aspectized. 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 interactions involving at least one optional or alternative feature are stable in both AspectJ and Java versions. Figure 8 shows the results for the interaction between two alternative features of Best Lap: 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 aspectized in the AspectJ version and, as a result, they are scattered over less components than in the Java version. The two components which mixed these features (Figure 8) in the AspectJ

Figure 8. Examples of interactions between optional and mandatory features and between two optional features.
version modularize shared code of both features (overlapping) that could not be placed in distinct aspects. Overlapping between features will be fatherly discussed in Section 6.3.

In addition the component interlacing, we also analyze two other categories of interaction: operation interlacing and LOC overlapping. Figure 8 depicts the result of operation interlacing and LOC overlapping for the interaction of two features: SORTING vs. LABELMEDIA. The pair of features is also a representative of optional or alternative features. As discussed before, interactions of this category are more stable in the AO version, as well as it shares fewer components and operations. But we can also see in Figure 8, that although the AO version concentrates the interaction in a few places (lower value of operation interlacing), 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 overlapping) in the AO solution of interactions involving optional or alternative features.

6.3 Scalability in Complex Interactions

The previous subsection focused on discussing how the aspectization of pair-wise interactions impacts different modularity attributes. This section discusses how conditional compilation and AOP scaled in interactions involving a greater number of features.

Figure 9 describes some representative features organized in terms of releases and categories of interaction (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 the conditional compilation mechanism to scale when complex interactions amongst features occur. These charts describe the impact of introducing new features on other features. For instance, the introduction of alternative features (PHOTO, MUSIC and VIDEO) increased significantly the overlapping code among such features and also optional features, such as COPYMEDIA and SMS. This behavior in some way was expected, since alternative features tend to reuse some part of their code to implement their functionality. However, surprisingly the introduction of one optional feature could also affect other optional feature. This occurred in the sixth release when the introduction of SMS increases the overlapping code of the COPYMEDIA feature. This higher overlapping represents in practice the existence of more explicit dependency between such optional features.

We have observed that the AO implementations usually scaled well for all kinds of interlacing interactions. We employed inter-type declarations to address component interlacing and pointcut-advice to deal with operation interlacing. The right-hand side of Figure 9 shows that no kind of interlacing is observed in all features analyzed. However, the presence of overlapping interactions can hinder a smooth interaction process and, sometimes, negatively affect the features being composed. This occurs because the aspectization of some specific interactions 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 feature COPYMEDIA by the fifth release, but this code is in a new component (aspect) in release 7 because the SMS and CAPTUREMEDIA features depend on parts of it. Again, AspectJ presents the same recurrent problem described above in which the introduction of one optional feature can affect 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, in this case overlapping.

7. RELATED WORK AND STUDY CONSTRAINTS

Some recent research work explores the use of AOP in the development or refactoring of software SPLs [1, 3, 15]. Most of these works, however, only concentrates on the qualitative analysis of the features aspectization process. Kästner et al [15] present a case study on refactoring the Berkeley DB system into a SPL. The authors reported several limitations on the modularization of features when using the AspectJ language, such as the increasing of coupling between aspects and classes due to the strong dependency of aspect pointcuts and implementation details of the base code. In our work, we have also encountered some of the limitations reported by Kästner et al [15]. In addition, we (i) categorize evolution scenarios which AspectJ succeeds or not (Section 4 and 5) and (ii) investigate the stability and scalability of this language regarding feature interactions.

Figure 9. Scalability of conditional compilation and AOP in complex interactions.
product-line architectures have addressed the decomposition of architectures into features. Mezini and Ostermann [21] have identified that feature-oriented approaches (FOAs) are only capable of modularizing hierarchical features. They propose CaesarJ [20], an AO language that combines ideas from both AspectJ and FOAs, to provide a better support to manage variability in SPLs. More recently, Apel & Batory [4] have proposed the Aspektual Mixin Layers [3] approach to also 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, has not considered a significant suite of software metrics and did not address evolution scenarios and stability.

Regarding our study constraints, the applicability and usefulness of some of the specific metrics used in this study has often been questioned such as the cohesion measure. 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. The multi-dimensional analysis allows us to grasp which measurement outliers are significant and which are not. In fact, when drawing conclusions from the results we have considered all the gathered data and never relied upon one single piece of data from this set. 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 development of SPLs is largely dependent 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, 17]. This study has 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 the analysis of the implementations modularity, change propagation and feature interaction analysis.

From this analysis we have discovered a number of interesting outcomes. Firstly, the AO implementations of the studied SPLs tended to have more stable design particularly when a change target 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 [19]. However, AO mechanisms did not cope with the introduction of widely-scaled mandatory features or when changing a mandatory feature into alternatives (Section 4.1 and 5.1). Furthermore, such mechanisms usually scaled well for interaction that not involved shared code, although AspectJ faces difficulties to address different SPL configurations.

9. REFERENCES


