Analysing the Impact of Feature Dependency Implementation on Product Line Stability: An Exploratory Study

Bruno B. P. Cafeo, Francisco Dantas, Alessandro Gurgel, Everton Guimarães, Elder R. Cirilo, Alessandro Garcia, Carlos J. P. Lucena

Informatics Department, Software Engineering Laboratory, OPUS Research Group
Pontifical Catholic University of Rio de Janeiro - Brazil
{bcafeo, fneto, agurgel, eguimaraes, ecirilo, afgarcia, lucena}@inf.puc-rio.br

***Abstract* – The evolution of software product lines (SPLs) is particularly challenging. SPL functionalities, usually decomposed into features, naturally depend among them. As the SPL evolves, the implementation of each feature dependency might increasingly affect more code elements. Therefore, as the complexity of feature dependency implementation grows up through code evolution, it is likely to negatively impact on the SPL stability. Stability refers to the amount of changes between SPL releases and it is directly influenced by the complexity of feature dependency implementation. In this context, a number of emerging programming techniques can be used to keep the complexity of feature dependency code under control. Nevertheless, there is a lack of studies analysing the impact of different programming techniques on the stability of feature dependency code in evolving SPLs. This paper presents a first exploratory analysis on the use of programming techniques to implement and evolve feature dependencies. Our analysis was developed in the context of three evolving SPL systems, which are made up of 210 feature dependencies. Our initial findings reveal that two particular types of feature dependency implementation using different programming techniques play a role in the SPL stability.**

***Keywords* – Feature dependency; Stability; Programming techniques; Software product line.**

I. INTRODUCTION

Software Product Line (SPL) development is becoming increasingly incremental to cope with new stakeholders needs [1][10]. SPL-based development focuses on the software decomposition into modular units of functionality defined as features. However, with the growing complexity and incremental development of SPLs, relationships between features in the code are often created, removed or changed [2]. These relationships are the so-called feature dependencies. Feature dependencies are materialised in SPL source code by means of relationship between program elements (e.g. a method call between the code that implements features A and B). Cho et al. [4] state that feature dependency code must be localised in one or a few modules aiming that the inclusion, exclusion or modification of a feature causes minimal changes to other features' code during the evolution of the SPL [4].

Design stability encompasses the sustenance of the product line's modularity properties in the presence of changes [7]. Thus, it stands out as one of the most desirable attributes of high-quality software as it affects SPL maintenance [5][6]. In this context, implementation of feature dependencies may become a barrier for the adoption of SPLs in volatile software domains if such dependencies are not kept under control [3]. The pieces of code that realise feature dependencies can be spread throughout many program elements in several modules. This indicates the number of program elements likely to be affected by modifications in the feature code. This distribution of dependency code between units is called in this paper by feature dependency diffusion. The higher diffusion of feature dependency implementation prevents the independent development and change of features. Thus, the instability of an SPL is often caused by (i) a high number of program elements containing feature dependency code, and (ii) program elements indirectly affected by changes related to feature dependency implementation [2][4][29].

A number of programming techniques could be used in order to ameliorate the likelihood of these negative consequences related to feature dependency implementation. Each programming technique provides different mechanisms to support the realisation of feature dependencies in the SPL code. The use of conditional compilation to implement SPLs, despite the controversial discussion [9], is still common in practice [10]. There are also other programming mechanisms provided by contemporary languages (e.g. AspectJ and CaesarJ) which have been gaining ground as alternatives to accommodate modular implementation of feature dependencies. In fact, there are several studies focusing on the use of programming mechanisms to implement and evolve SPLs [5][7]. In addition, many studies proposed means to handle feature dependencies in SPL with these mechanisms [4][8]. Nevertheless, none of them analyse the impact of different programming techniques on instabilities related to feature dependency implementation.

In this context, we argue that the instability related to feature dependency implementation can vary with the adopted programming techniques and their mechanisms. To analyse this, we conducted an exploratory study of forty-five releases of tree different SPLs. In particular, we aim to compare the instability caused by feature dependency

changes[1] over program elements implemented by different programming techniques, such as Conditional Compilation, Aspect-Oriented Programming (AOP) and Feature-Oriented Programming (FOP). Our main findings were: we confirm the intuition that, in general, the complexity of feature dependency implementation negatively affects SPL stability. However, we also observed that the type of feature dependency plays a key role on the SPL stability. Finally, we also identified that different programing techniques are more appropriate to implement these different types of feature dependencies from the SPL stability perspective.

The remainder of this paper is structured as follows. Section II presents background on the main topics of this paper. A complete description of our study is provided in Section III. Section IV presents the results of our exploratory study. Section VI discusses the limitation of this work. Finally, Section VI presents related work and Section VII concludes the paper with some remarks and future directions.

II. Programming Techniques and Feature Dependency

This section presents an overview of the programming techniques used in our analysis (Section II.A), concepts of feature dependency (Section II.B), as well as the notions of feature dependency diffusion (Section II.C) and stability (Section II.D).

A. Programming Techniques

In our study we are considering Java with pre-processor directives as representative implementation of Conditional Compilation, which is a well-known and industry-strength technique for handling SPL evolution [7]. Moreover, we are considering AspectJ and CaesarJ and their programming mechanisms as alternative to Conditional Compilation. AspectJ and CaesarJ were chosen because both are well-known representative of aspect-Oriented (AOP) and feature-Oriented programming (FOP) languages. In addition, they support a wide range of mechanisms to implement features that are significantly different. They would enable us to address our research question by analysing a broad range of programming mechanisms used to implement feature dependencies.

Regarding *Conditional Compilation*, the implementation of three target SPLs relies on the use of pre-processors directives like *#ifdef* and *#endif* to realise the feature code. This means that annotated code fragments enclosed by compilation directives are added or removed from the original SPL depending on the selection of features. On the other hand, *AspectJ* supports the implementation of separate modules, the so-called *aspects*, which have the ability to modularise concerns that might spread their behaviours throughout the base code. The constructions often used to implement features are: *advices* that are programming units representing behaviours and *intertype declarations* which allow static modification of classes' structure (e.g. inclusion of attributes and methods) and hierarchy changes. In addition, *CaesarJ* supports advanced mechanisms that enable other ways of decomposing software modules [5]. In particular, *virtual classes* and *mixin composition* are prominent mechanisms of CaesarJ to support feature implementation. The use of virtual classes makes it possible to apply overriding and late binding to inner classes in a similar way of late-bound of virtual methods. Then, by means of *mixin composition* mechanism, different modules are composed to build more complex modules without compromising the independence of each one.

B. Feature Dependency

Features are a useful abstraction to express variability in an SPL [12]. According to Ye and Liu [12], to fulfil their tasks, features usually need to interact with other features. These relationships are so-called feature dependency [4]. In the source code, a feature dependency occurs when program elements inside the boundaries of a feature depend on elements outside the feature [28]. There exist three types of program elements: attributes, operations and declarations. For instance, methods and advices are classified as operations in CaesarJ; the same applies to AOP-specific languages, such as AspectJ. Pointcut expressions, intertype declarations and mixin composition expressions are classified as declarations. In this paper, program elements and feature dependencies can be defined as follows.

Definition 1 (Program, Module and Program Element). *A program P consists of a set of modules, M. A module M is a sequence of program elements, E_M. A program element can be an attribute, an operation or a declaration. Let Att_M be the set of attributes of M, Op_M be the set of operations of M and Dec_M be the set of declarations of M, $E_M := Att_M \cup Op_M \cup Dec_M$.*

Definition 2 (Feature Dependency). *Let F_1 and F_2 be two different features. F_1 depends on F_2 when at least one program element in F_1 refers to a program element in F_2.*

Figure 1 exemplifies the occurrence of feature dependencies in the same SPL implemented by different programming techniques presented in Section II.A. In conditional compilation, blocks of *#ifdef* statements (BOX #1 – lines 07 and 11) associated with feature F7 are included in code associated with feature F6. This means there is a dependency between F6 and F7 due to the use of the attribute *a* (BOX #1 – line 08), method *setX* (BOX #1 – line 08) and method *getX* (BOX #1 – line 12), all belonging to feature F6, in part of code enclosed by directives related to the feature F7.

In AspectJ, using *advice* (BOX #3 – lines 03 and 08), the dependency between F6 and F7 is realised. In other words, the definition of the value for the attribute *x* is performed by an advice *around* (BOX #3 – line 03) which changes the value of the argument of the method *m1*, and after returns the control flow to *m1*. Moreover, the value of *x* is printed after

[1] From hereafter we use the terms "stability" and "instability" to refer only to changes occurred by the implementation or modification of feature dependencies.

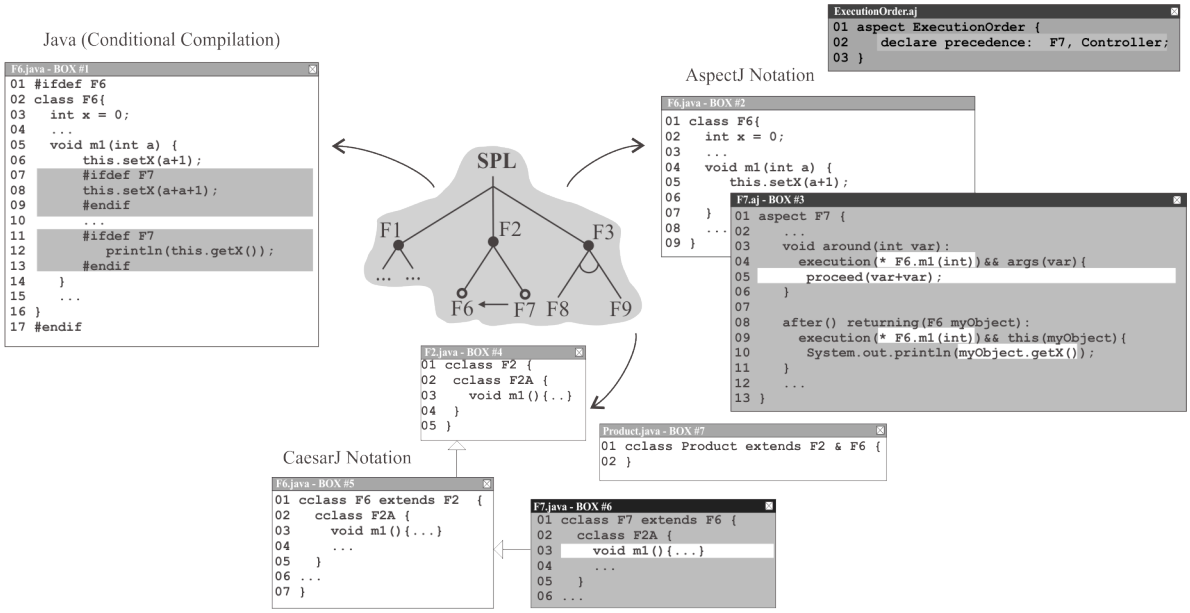


Figure 1. Feature dependency using different advanced programming mechanisms

the execution of *m1* through the advice *after* (BOX #3 – line 08), which calls the method *getX* (BOX #3 – line 10) from F6. Whereas in CaesarJ, the dependency between F6 and F7 is implemented using *virtual class*. The method *m1* (BOX #5 – line 03) is overridden by F7 by means of virtual class F2A in F7 (BOX #6 – line 02).

C. Feature Dependency Diffusion

Feature dependency diffusion refers to the complexity of a feature dependency regarding the program elements involved in such relationship. In other words, it indicates the number of program elements likely to be directly affected by eventual modifications in the feature dependency. In this paper, feature dependency diffusion can be defined as follows.

Definition 3 (Feature Dependency Diffusion). *Let F_1 and F_2 be two different features, where F_1 depends on F_2. The feature dependency diffusion FD is the number of different program elements E_M involved in the realisation of this dependency.*

Figure 1 shows examples of feature dependency diffusion. For instance, considering Figure 1 (BOX #1 – Java version), we can note that only method `m1` belonging to F6 is involved in a feature dependency with F7, and thus the diffusion of feature dependencies is one. In Figure 1 (BOX #2 and #3 – AspectJ version), the two advice (BOX #3 – lines 03 and 08) are involved in the feature dependency of F6 with F7 and so, in this case, the diffusion of feature dependencies is two. The same reasoning is followed in CaesarJ.

D. Stability

Based on the stability definition provided by Kelly [27] we are considering that an SPL is stable regarding feature dependencies if the implementation of such dependencies do not affect many program elements every time that a new product is released. Thus, instabilities are caused by changes related to feature dependency implementation which affects several program elements directly or indirectly. Of course, this notion of "several" depends on the particular context where stability is being measured. In the context of our study, this is a relative notion as our intent is to compare multiple SPL implementations produced with alternative programming techniques. In this paper we define instability as follows.

Definition 4 (Instability). *Let F_1 and F_2 be two different features, where F_1 depends on F_2. Instability consists of any direct or indirect change in program elements E_M of F1 or F2 caused by the implementation of the feature dependency.*

Instabilities associated with feature dependencies are exemplified in Figure 1. In order to implement the dependency between features F6 and F7 changes were realized using the programming techniques described in Section II.A. Using conditional compilation two changes were required to implement F7 (Figure 1 – BOX#1 – lines 07 to 11 and 13 to 15). The same implementation using aspects leads to three changes (yellow lines in BOX #3). Finally in CaesarJ only one modification was required (line 03 in the BOX#6).

III. Study Setting

This section describes our study configuration. Section III.A presents our research goal. Section III.B provides an overview of the target SPLs. Section III.C explains the evaluation procedures applied to the target systems.

A. Research goal

Our objective is to evaluate the impact of feature dependency implementation on the stability of evolving SPLs. We are particularly interested in observing how specific programming techniques and their mechanisms impact on the implementation of feature dependencies in terms of stability. In order to achieve this goal, we performed a comparative analysis based on the stability in conditional compilation, AspectJ and CaesarJ implementations of the target SPLs in each evolution (Section II.B). Our research goal relies on the answer of the following research question: *Does the use of different programming techniques play a role in the stability of evolving SPLs?* The focus of the comparison is on the use of these programming techniques to implement feature dependencies.

To answer the research question we applied a number of evaluation procedures. Our analysis embraced 15 releases of three SPL systems. Each release was implemented in Java with conditional compilation, AspectJ and CaesarJ. The target SPL systems and evaluation procedures are following described.

B. The Target SPLs

The three medium-sized SPL systems used in this study include two board games and one embedded mobile application. The first one is actually a family of two board games called *GameUP* and encompassing the games called *Shogi* and *Checkers* [5][13]. Each game is also an SPL in itself. The second SPL is an embedded mobile software called *MobileMedia* (MM) [7] which allows users to manipulate images, videos and music on different mobile devices. Following we describe the target SPL systems.

GameUP. It is a program family of three board games which are by themselves SPLs. In this work we only analysed Shogi and Checkers game releases. Shogi is a chess games whereas Checkers is an American checker game. Both of them provide features to manage various functionalities for customising the board (e.g. indicating moveable pieces) and the matches between players (e.g. indicating player turns). The evolution scenarios comprise the inclusion of optional and alternative features providing us a variety of feature dependencies, which are fundamental to conduct the investigation of this work. Figure 2 presents the Shogi feature model referent to the fifth release. The Checkers feature model is not shown due to its similarity with the Shogi feature model.

MobileMedia. It is an SPL that provides support to manage photo, music, and video on mobile devices. The core feature represents basic media management actions such as create/delete media, label media and view/play media. The alternative features are the types of media supported such as photo, music and/or video. The optional features are transfer photo via SMS, count and sort media, copy media and set favourites. The evolution scenarios comprise different types of changes involving the inclusion of mandatory, optional and alternative features, as well as changing of one mandatory feature into two alternatives. Figure 3 shows the MobileMedia feature model referent to the seventh release. Table I shows some general characteristics of the three target SPL systems. For more information about each of them, the reader may refer to the respective work [5][7][13].

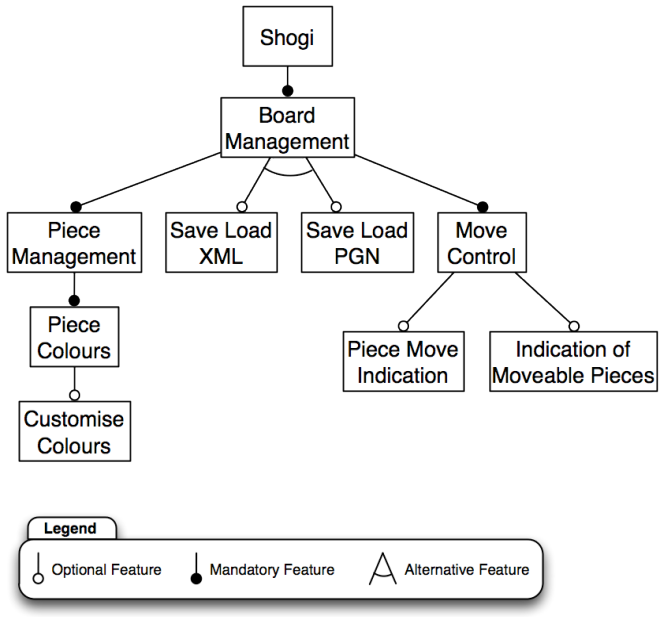


Figure 2. Simplified Shogi feature model

TABLE I. CHARACTERISTICS OF THE TARGET SPL SYSTEMS

	Shogi	Checkers	MobileMedia
Application type	Board game	Board game	Mobile data
Code availability	Java, AspectJ, CaesarJ	Java, AspectJ, CaesarJ	Java, AspectJ, CaesarJ
Number of releases	4	4	7
Avg. # of features	9	8	25
Avg. KLOC	4	2	10

C. The Evaluation Procedures

The study was divided in five major phases: (1) the implementation of SPL releases with all programming techniques described in Section II, (2) the implementation alignment of SPLs, (3) the assignment of features to elements of the source code, (4) the identification of feature dependencies, (5) the quantification of feature dependency diffusion and stability.

All phases were conducted by an independent group of three postgraduate students using the implementation of the three target SPLs in Java, AspectJ and CaesarJ. However, design practices were used, enforced, and reviewed throughout the creation and evolution of all the SPL releases and versions [14]. In all the cases, the implementations were also reviewed by experts in the field. For instance, CaesarJ implementations were reviewed by contributors to the CaesarJ design and implementation. Moreover, the Java and

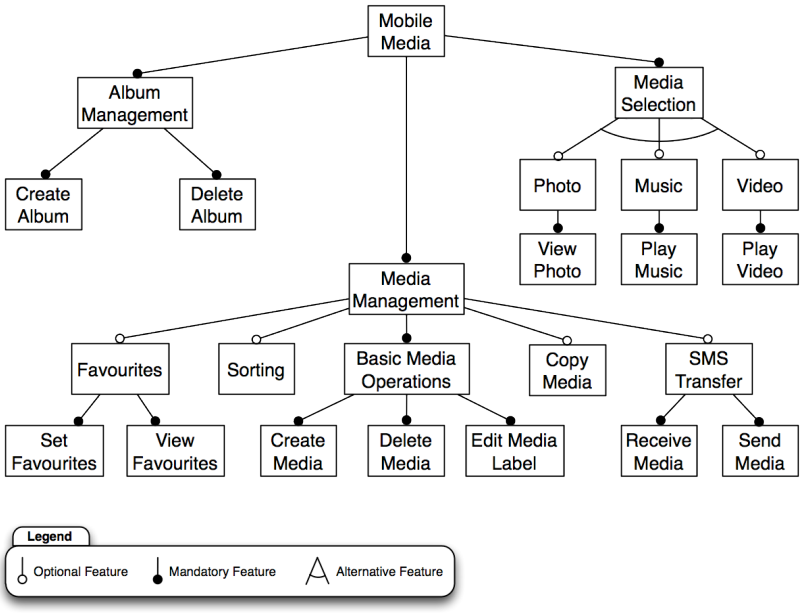


Figure 3. Simplified MobileMedia feature model

AspectJ implementation and scenarios of the MobileMedia SPL were extracted from a previous study conducted by Figueiredo et al. [7]. The CaesarJ implementation of MobileMedia was extracted from the work of Dantas and Garcia [5]. The scenarios of Checkers and Shogi SPLs and their AspectJ and CaesarJ implementations were also extracted from previous work of Dantas and Garcia [5]. In the following, we detail the five major phases of our study.

SPL releases implementation. In the first phase, we created the conditional compilation version for four releases of Checkers and Shogi SPLs. The AspectJ and CaesarJ versions for four releases were available in Dantas and Garcia [5] and Gurgel et al. [13]. The AspectJ and Java with conditional compilation versions for the eight releases were available in Figueiredo et al. [7]. Moreover, the CaesarJ implementation was available in Dantas and Garcia [5].

SPL alignment rules. All SPL releases were verified according to a number of alignment rules to assure that the implemented functionalities in different programming techniques were exactly the same in all versions. Furthermore, design practices to ensure high degree of modularity and reusability were used throughout the creation of all the SPL releases [15]. It is important to highlight that some minor refactoring and corrections had to be performed when misalignments were observed.

Feature assignment. In this phase, we mapped features in the source code for further feature dependencies identification. Each feature was mapped following the principles suggested by Kastner *et al.* [16]. So, we used background colours to highlight pieces of code that were implementing features. We used one-to-one mapping to ensure that each feature has one colour and every piece of code was mapped to one feature.

Feature dependency identification. Once all features were mapped, we analysed the source code to identify the dependencies between features. Basically a feature dependency occurs when the body of a module is marked with more than one colour or when there is a dependency relationship between modules marked by different colours.

Quantifying feature dependency diffusion and stability. To quantify feature dependency diffusion we count every different program element involved in a feature dependency (See Definition 3). To quantify instabilities we count every different program element changed during a modification related to the implementation of a feature dependency (See Definition 4).

IV. Discussion and Results

The evolution of the SPLs was analysed following the evaluation procedures presented in Section III.C. We discuss whether and how different programming techniques (Section II.A) contribute to instabilities caused by feature dependency implementation in evolving SPLs. First, we analyse the feature dependency diffusion and the influence of different programming techniques on its implementation (Section IV.A). In addition, we also discuss the instabilities regarding feature dependency implementation through the evolution of the target SPLs (Section III.B). Finally, it is discussed the influence of the use of programming techniques (Section II.A) on SPL stability (Section III.C).

A. Feature Dependency Diffusion vs. Programming Techniques

The occurrence of feature dependencies takes place in different ways depending on the programming technique used to implement them. Two types of feature dependencies emerged from our analysis: *disjoint* and *overlapping*

dependencies. An overlapping dependency occurs when at least one program element is shared between the F1 and F2 implementation [7]. In this dependency the shared elements entirely contribute to both features rather than being disjoint. On the other hand, a disjoint dependency between two features F1 and F2 occurs when their implementation does not share program elements apart from the dependency code (i.e. F1 and F2 are disjoints). This means that the realization of a disjoint dependency creates a relationship between two independent features.

Feature dependency diffusion and disjoint feature dependencies. In Figure 5 we can see that the behaviour of the three programming techniques analysed considering the variation of feature dependency diffusion is different in each target SPL. In this context, a key factor influencing feature diffusion was the degree of code overlapping (sharing) between two features. The GameUP scenarios encompass features with no shared code, thereby explaining the superiority of conditional compilation in the Shogi and Checkers SPLs. We observed that the use of conditional compilation is superior in these cases: it often entails lower diffusion of feature dependency than the AspectJ and CaesarJ counterparts. Figure 4 illustrates a scenario where the feature dependency diffusion using conditional compilation is equal to 1 (relationship between F5 and F9 - lines 04 to 06 in F5 and lines 03 to 05 in F9), whereas using AspectJ is equal to 2 (relationship between F11 and F5, and between F11 and F9). In scenarios like these, the ability of AOP (AspectJ) and FOP (CaesarJ) of modularising features in separate modules (e.g. aspect or cclass) has a negative consequence: the number of programming units realising those features tends to be higher, and feature dependency diffusion tends to increase. This is the opposite with conditional compilation as *#ifdef* blocks rely just on an OR/AND conditional operator.

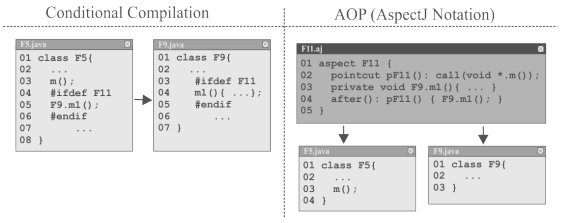


Figure 4. Diffusion: conditional compilation vs. AOP

The AspectJ and CaesarJ notation in Figure 1 makes evident the higher number of feature dependency diffusion in AOP and FOP. In AspectJ, the aspect F7 (Figure 1 – BOX #3), which implements the feature with the same name has a dependency with the code that implements the feature F6 (Figure 1 – BOX #2). In addition, there is also a dependency between the aspect F7 and the aspect ExecutionOrder. Therefore, the feature dependency diffusion in the AspectJ solution is two; CaesarJ solution also leads to two dependencies ((F6 and F7) and (F6 and F2)). The diffusion of feature dependency in CaesarJ tends to increase due to the massive use of virtual classes, which rely on multiple inheritances supported by CaesarJ.

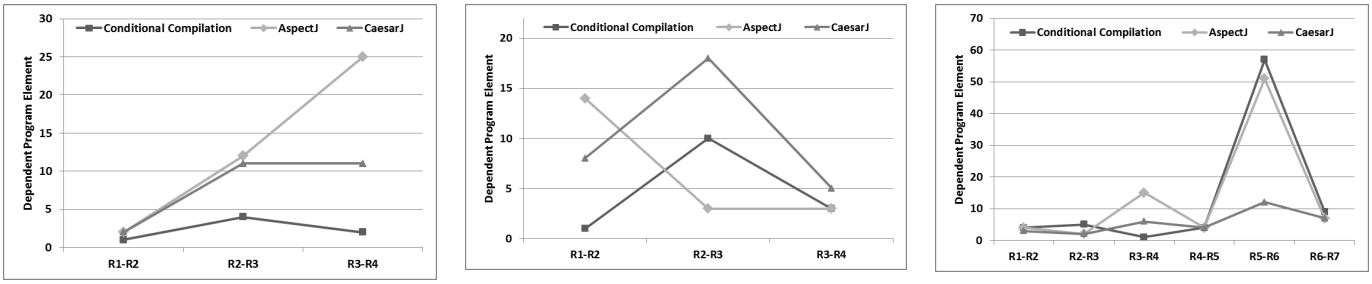
However, in an exceptional case, the evolution R2-R3 of Checkers (Figure 5-b) presents favourable feature dependency diffusion to AspectJ. In order to understand the superiority of AspectJ in R2-R3, it is essential to highlight that the number of existing methods in the AspectJ version (R1-R2) is higher than other techniques. This happens because the join point model supported by AspectJ forces the developer to split a single method, *m*, in more than one (e.g. *m1* and *m2*), when the join point shadows are pieces of code that are inside m. As most of the new feature dependencies originated in R2-R3 take place in already dependent methods that only exist in the AspectJ versions, the creation of new features dependencies are not detrimental to feature dependency diffusion in AspectJ.

Feature dependency diffusion and overlapping feature dependencies. Features with shared code are those that require a part of other feature code in its implementation. Most feature dependencies created in MobileMedia evolution are based on existing features that share code among them. Figure 5-c illustrates the variation of feature dependencies along MobileMedia evolution. Apart from the evolution R3-R4, where conditional compilation presented lower feature dependency diffusion than AspectJ, the use of different programming techniques does not promote significant difference in terms of dependency diffusion. This happens because, using conditional compilation, features depend among them through lines of code included in their implementation (Figure 1 – BOX#1), which takes place in R3-R4. On the other hand, in AspectJ a new module is required to realise the same dependency and, as consequence the number of dependencies increases.

B. Feature Dependency Diffusion vs. Stability

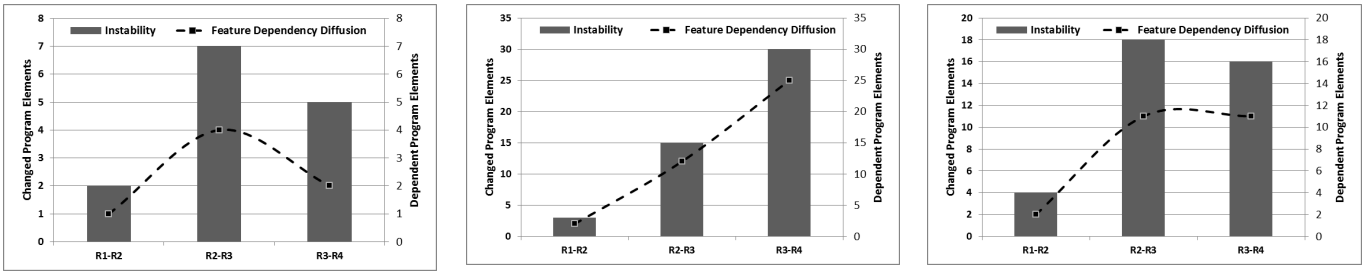
From our initial findings, we could observe that implementation of feature dependencies usually requires preparatory modifications in the SPL source code. This means that in order to implement dependencies by means of the advanced programming mechanisms analysed (Section II.A), the source code often needs to undergo preliminary modifications. This preparation consists of refactoring operations to accommodate the use of those mechanisms. Examples of modifications required to implement feature dependencies were previously explained in Section IV.A.

Feature dependency diffusion as an indicator of stability. Figure 6 presents a representative case derived from Shogi game, where we can observe the variation of both feature dependency diffusion and instabilities. As illustrated in the figure, the variation of feature dependency diffusion follows the variation of instability regardless of programming technique and type of feature dependency (i.e. disjoint and overlapping dependencies). Analysing the numbers, it is possible to state that feature dependency diffusion operates as an indicator of stability in evolving SPLs. In the example illustrated in Figure 1 it is possible to visualise the relation between diffusion and stability in each programming



(a) Shogi (b) Checkers (c) MobileMedia

Figure 5. Feature dependency diffusion variation through target SPLs



(a) Condition Compilation (b) AspectJ (c) CaesarJ

Figure 6. Diffusion vs. Stability for Shogi game

technique (Section II.A). In order to implement feature dependencies using conditional compilation the method *m1* was modified once (see Figure 1 – BOX#1). The diffusion associated with this dependency relationship is equal to one as only one the program element (method *m1*) is involved in the dependency between F6 and F7. In AspectJ the number of changes is equal to two (see BOX#3 – yellow lines) and the diffusion is two as well as two advice are involved in the feature dependency. Finally, in CaesarJ, only one change was required and the feature diffusion in this implementation is equal to one as just one program element is involved in the dependency.

This relationship between feature dependency diffusion and stability is particularly important to early adopters who wish to use different implementation techniques when moving towards the evolution of SPLs. As the existing relationship between feature dependency diffusion and stability takes places regardless of programming technique, it is essential to understand how particular mechanisms of specific programming techniques deal with the implementation of feature dependency (Section IV.C).

Accuracy of the stability indicator. As aforementioned there is a correlation between feature dependency diffusion and stability. This correlation is characterised by a gap in the quantification of diffusion and stability and this gap varies depending on the programming technique used. In Figure 5, it is possible to observe that this gap is bigger when conditional compilation is used. This means that programmers can better foresee the stability of the AspectJ or CaesarJ evolving code, based on the analysis of feature dependency diffusion.

Figure 7 illustrates the percentage of program elements affected by feature dependency diffusion per technique for the Shogi game. The percentage values are associated with feature dependencies instabilities, which are represented by the light grey bars, whereas the diffusion is expressed by means of the dark grey bars. Analysing Figures 6 and 7, we can say that the accuracy of the feature dependency diffusion as stability indicator can vary depending on the programming technique adopted. Based on the results presented in Figure 7, for instance, the use of AspectJ operates as a better indicator. The explanation for this is that AspectJ gap between stability and diffusion is smaller when compared to CaesarJ and conditional compilation. This happens due to power of the programming mechanisms, which allows us to keep the complexity of feature dependency under control. The use of AOP and FOP, for instance, promotes fewer changes in source code to implement overlapping feature dependency (Section IV.A) when compared to conditional compilation.

C. Programming Techniques vs. Stability

According to our findings, the use of different programming technique tends to produce SPLs with different degrees of stability. Based on this understanding, the results of our study suggest the most appropriate programming techniques to implement each case of feature dependencies (Table II).

In general, program elements that take part of the feature dependency implementation appear within method bodies. In general, they precede or come after other elements (e.g. method calls), this a typical case of overlapping feature dependency. This scenario can be better implemented in terms of stability using pointcut-advice on *joinpoints* of call

type in AspectJ. In Figure 4, the evolutions R2-R3, R3-R4 and R6-R7 involved changes on overlapping feature dependencies, which were implemented by means of pointcut-advice, avoiding invasive changes.

TABLE II. SUGGESTION OF PROGRAMMING TECHNIQUES

Programming Technique	Overlapping Features	Disjoint Features
Conditional Compilation		✓
AOP (AspectJ)	✓	
FOP (CaesarJ)	✓	

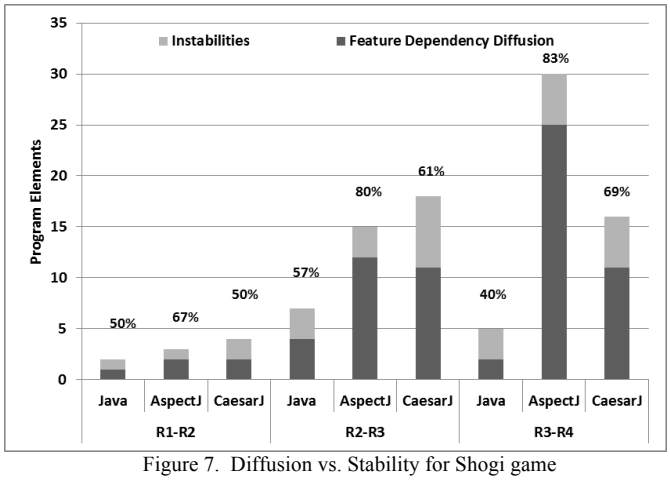


Figure 7. Diffusion vs. Stability for Shogi game

In addition, the use of virtual classes supported by CaesarJ also brings benefits to the implementation of overlapping features. In this case, the code overlapping is implemented by an overridden operation (Figure 6). On the other hand, the use of conditional compilation facilitates the implementation of feature dependencies that do not share code between them (Section IV.B).

Additionally, the implementation of disjoint feature dependencies is better managed using conditional compilation. Disjoint feature dependencies take place in most cases in a fine-grained way. This means that no refactoring operations (i.e. changes in the source code) are required to implement disjoint feature dependencies even when just lines of code, which are not candidate joinpoints, are involved in the dependency implementation. A detailed explanation supported by concrete example is provided in Section IV.A.

Finally, we can conclude that the use of different programming techniques to implement feature dependencies generates source code with different degrees of stability. In particular, it seems that the use of FOP or AOP, as supported by CaesarJ and AspectJ, is more indicated than conditional compilation to implement SPLs when overlapping feature dependencies dominate the code. We observe that these implementation alternatives consistently provided higher degree of stability (Section IV.A and IV.B) for all the analysed SPLs.

V. THREATS TO VALIDITY

In our study, the conclusion validity threats are related to the feature identification and their relationships. To reduce the influence of this threat, we identified features based on the feature models, which were validated in previous studies [5][7][13]. In addition, the three postgraduate students that participated of the study investigation have good knowledge on feature-oriented domain analysis. Moreover, they performed strong revisions to check the alignment of each feature represented in the feature model with the SPLs code. One additional threat is how the features and their dependencies were identified. They directly impact the stability measurement (Section III.C).

A threat to construct validity relies on the procedures for quantifying the changes and the degree of feature dependencies in the target SPL application (Section III.C). They can be directly associated with the developer's style as this quantification are code-based. In order to ameliorate this issue, the quantification of feature dependency in each programming technique was widely discussed among experienced Java, AOP and FOP developers.

Threats to internal validity reside on alignment rules used to identify feature dependencies in all programming techniques. To reduce this threat, we performed a detailed analysis of the SPLs code in order to reduce the inconsistencies on the identification process. Threats to external validity are conditions that allow results generalization. To better generalise the results, we selected applications from different domains and developed by different companies. These applications are representative for feature-dependency and have a significant size (Section III.B). Moreover, they embrace (non-) crosscutting mandatory, optional and alternative features which allowed us to investigate a wide range of feature dependency scenarios.

VI. RELATED WORK

Since the introduction of Feature-Oriented Domain Analysis (FODA) by Kang et al. [17], many works have emerged [18][19][20][21] to study the effects of feature dependencies in SPL development. We distinguish two groups of such works. First, we highlight related works aimed at identifying the impact of feature dependencies on SPL maintenance in terms of failures and feature cohesion [2][22][23][29]. Second, there are researches that categorised the spectrum of feature dependencies in different software evolution scenarios [7][20][24]. Nevertheless, to the best of our knowledge, our results are the first to empirically investigate the influence of feature dependency implementation on the stability of SPLs. Thus, we can see this work as a first step to a more ambitious agenda of studies on the influence of feature dependencies at the SPL implementation level.

Impact of feature dependency on SPL maintenance. Cataldo *et al.* [22] have empirically studied feature-oriented development in order to observe the impact of feature

dependency on integration testing failures. Recently, Garvin et al. [23] performed an exploratory study to conduct an investigation of faults on two feature-oriented open source systems. They observed that only a few number of faults are associated to feature dependency implementation. Liebig et al. [29] have analysed forty pre-processor-based SPLs in order to provide a better understanding on how this programming mechanism is employed to implement variability. Complementing this study, Apel and Beyer [28] performed an exploratory analysis of the same SPLs, nevertheless in terms of feature cohesion. Finally, Ribeiro et al. [2] performed an empirical evaluation of these forty pre-processor-based SPLs, but now focusing on the maintainability of feature dependency code. They described numerous scenarios susceptible to failures when feature dependency code is maintained. In contrast to these studies, our work is concerned on providing a broadly analysis of the impact of feature dependency on the stability regarding the impact of other programming mechanisms (i.e., AOP and FOP), as well as, the SPLs' evolution history.

Feature dependency categories. There are some works [19][20][24][25] that have been proposing an initial categorization of feature dependencies focusing mainly on variability management. Based on such categorisation, in [8], Lee and colleagues have also proposed a general categorization for feature dependencies and aspect-oriented patterns to implement such dependencies. However, the authors did not consider the effects of these categorised dependencies on the stability of incremental development of SPLs. Figueiredo et al. [7] analysed the evolution of two SPLs in order to provide empirical evidence of how feature dependency changed over many releases. They observed how two different category of feature dependency (interlacing and overlapping) affect the stability of the source in aspect-oriented systems. As we mentioned, we provided a broader study that investigates the influence of different programming techniques on SPL stability.

VII. CONCLUSIONS

The use of specific programming techniques for feature dependency implementation exerts a major impact on SPL stability. In this context, it is important to know which and when of these programming techniques tend to promote positive and negative effects over feature dependency implementation and, as a consequence, the stability. This paper reported a comparative assessment of conditional compilation, AOP and FOP in the context of different SPLs, focusing on feature dependency stability.

Our analysis suggests that the type of feature dependency determines the programming technique that must be used in order to reach better SPL stability. For instance, while disjoint feature dependencies are better implemented using conditional compilation, overlapping features dependencies are better implemented using the composition mechanisms supported by AspectJ and CaesarJ. In addition, we also observe that the diffusion of feature dependencies through the SPLs evolution is a good indicator of stability independently from the programming technique employed. This means that as feature dependency diffusion increases, so does the instability and vice-versa. Even though we cannot claim our findings extrapolate beyond the scope of three analysed SPLs, it provides a number of aforementioned insights that can further explored and assessed in the future. As future work, we believe that this initial work can be further improved in many directions, including (but not limited to): (i) the same investigation can be carried out in the context of other systems or using rigorous forms of empirical methods, such as controlled experiments, and (ii) other attributes could be assessed such as feature dependency reuse.

REFERENCES

[1] P. Clements and L. Northrop, "Software Product Lines: Practices and Patterns", 3rd ed. Addison-Wesley Professional, 2001.

[2] M. Ribeiro, F. Queiroz, P. Borba, T. Toledo, C. Barbrand and S. Soares. "On the Impact of Feature Dependencies when Maintaining Pre-processor-based Software Product Lines," Proc. 10th International Conference on Generative Programming and Component Engineering (GPCE'11), pp. 23–32, October, Oregon, USA 2011.

[3] M. Svahnberg and J Bosch. "Evolution in software product lines," Journal of Software Maintenance and Evolution 11.6 (1999): 391-422.

[4] H. Cho, K. Lee and K. C. Kang. "Feature Relation and Dependency Management: An Aspect-Oriented Approach," Proc. 12th International Software Product Line Conference (SPLC'08), pp. 3–11, September, Limerick, Ireland 2008.

[5] F. Dantas and A. Garcia, "Software reuse versus stability: Evaluating advanced programming techniques," Proc. 24th Brazilian Symposium on Software Engineering (SBES'10), pp. 40–49, Salvador 2010.

[6] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. "Change Bursts as Defect Predictors," Proc. 1st International Symposium on Software Reliability Engineering (ISSRE'10), pp. 309-318, November, San Jose, USA 2010.

[7] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, and F. Dantas,"Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability," Proc of. 30th International. Conference on Software Engimeering. (ICSE'08), pp. 261–270, Germany 2008.

[8] K. Lee and K. C. Kang, "Feature Dependency Analysis for Product Line Component Design," Proc. 8th Software Reuse Methods Techniques and Tools (ICSR'04), vol. 3107, pp. 69–85, Madrid, Spain 2004.

[9] K. Pohl, G. Bockle, F. J. Linden. "Software Product Line Engineering: Foundations, Principles, and Techniques," Ed. Springer-Verlag. Springer, 2005.

[10] V. Alves, A. C. Neto, S. Soares, G. Santos, F. Calheiros, V. Nepomuceno, D. Pires, J. Leal and P. Borba. "From Conditional Compilation to Aspects: A Case Study in Software Product Lines Migration," Proc. AOPLE at GPCE, pp. 1–7, 2006.

[11] P. Ye, X. Peng, Y. Xue and S. Jarzabek. "A Case Study of Variation Mechanism in an Industrial Product Line," Proc. 11th International Conference Software Reuse (ICSR'09). Springer-Verlag, ,pp.. 126-136, Edmonton, Canada 2009.

[12] H. Ye and H. Liu, "Approach to Modelling Feature Variability and Dependencies in Software Product Lines," Proc. IEEE Proceedings - Software, vol. 152, no. 3, pp. 101–109, June, 2005.

[13] A. Gurgel, F. Dantas, and A. Garcia, "On-demand Integration of Product Lines: a Study of Reuse and Stability," Proc. 2nd International Workshop on Product Line Approaches in Software Engineering, pp. 35–39, May, Honolulu, Hawaii, 2011.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal. "Pattern-Oriented Software Architecture: A System of Patterns". J. W. Sons, Ed. Wiley, vol. 1, 1996.

[15] M. Kuhlemann, M. Rosenmuller, S. Apel and T. Leich. "On the Duality of Aspect-oriented and Feature-oriented Design Patterns," Proc. 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'07), Vancouver, British Columbia 2007.

[16] C. Kastner, S. Trujillo, and S. Apel, "Visualizing Software Product Line Variability in Source Code," Proc. 12th International Workshop on Visualisation in Software Product Line Engineering (ViSPLE'08), pp. 303–313, September, Ireland 2008.

[17] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson. "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[18] K. C. Kang, J. Lee, and P. Donohoe. "Feature-oriented Product Line Engineering," Proc. IEEE Software, 9(4):58-65, July/August 2002.

[19] Y. Smaragdakis and D. Batory. "Implementing Layered Designs with Mixin Layers," Proc. 12th European Conference on Object-Oriented Programming (ECOOP'98), pages 550-570. Springer-Verlag LNCS 1445, 1998.

[20] S. Ferber, J. Haag, and J. Savolainen. "Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line," Lecture Notes in Computer Science, 2002, Vol. 2379, p.p. 37-60, 2002.

[21] J. Lee, D. Muthig, "Feature-oriented Variability Management in Product Line Engineering."Proc. Communication ACM, vol. 49, no. 12, pp. 55-59, 2006.

[22] M. Cataldo and J. D. Herbsleb. "Factors Leading to Integration Failures in Global Feature-oriented Development," Proc. 33rd International Conference on Software Engineering (ICSE'11), pp. 161–170, May, Honolulu, Hawaii 2011.

[23] B. J. Garvin and M. B. Cohen, "Feature Interaction Faults Re-visited: An Exploratory Study," Proc. IEEE 2nd International Symposium on Software Reliability Engineering (ISSRE'11), Nov/Dec, Hiroshima, Japan 2011.

[24] R. S. S. Filho, and D. F. Redmiles. "Managing Feature Interaction by Documenting and Enforcing Dependencies in Software Product Lines," In Lydie du Bousquet & Jean-Luc Richier, ed., 'ICFI' , IOS Press, , pp. 33-48, 2007.

[25] W. Zhang, H. Mei, and H. Zhao. "Feature-driven Requirement Dependency Analysis and High-level Software Design," Journal of Requirements Engineering. 11, p.p. 205-220, 2006.

[26] M. Acher, P. Collet, P. Lahire and R. B. France. "Comparing Approaches to Implement Feature Model Composition," Proc. 6th European Conference on Modelling Foundations and Applications (ECMFA'10), pp. 3-19, June, Paris, France 2010.

[27] D. Kelly. "A Study of Design Characteristics in Evolving Software Using Stability as a Criterion," Proc. IEEE Transactions on Software Engineering, Vol. 32(6), pp. 315-329 (2006).

[28] S. Apel and D. Beyer. "Feature Cohesion in Software Product Lines: an Exploratory Study," Proc. 33rd International Conference on Software Engineering (ICSE'11), pp. 421-430, May, Honolulu, Hawaii 2011.

[29] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. "An Analysis of the Variability in Forty Pre-processor-Based Software Product Lines," Proc. 32nd International Conference of Software Engineering (ICSE'10), pages 105–114, May, Cape Town, South Africa 2010.