Enhancing Feature Interfaces for Supporting Software Product Line Maintenance

Bruno B. P. Cafeo
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro - RJ, Brazil
bcafeo@inf.puc-rio.br

ABSTRACT

Software product line (SPL) is a technology aimed at speeding up the development process. Although SPLs are widely used, their maintenance is a challenging task. In particular, when maintaining a SPL feature, developers need to know which parts of other dependent features might be affected by this maintenance. Otherwise, further maintenance problems can be introduced in the SPL implementation. However, the identification and understanding of the so-called feature dependencies in the source code are an exhaustive and error-prone task. In fact, developers often ignore unconsciously feature dependencies while reasoning about SPL maintenance. To overcome this problem, this PhD research aims at understanding the properties of feature dependencies in the source code that exert impact on SPL maintenance. Furthermore, we propose a way to structure and segregate feature interfaces in order to help developers to identify and understand feature dependencies, thus reducing the effort and avoiding undesirable side effects in SPL maintenance.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; K.6.3 [**Software Engineering**]: Software Management—*Software Maintenance*

General Terms

Design, Experimentation, Measurement

Keywords

Software Product Lines, Software Maintenance, Feature Modularisation, Feature Interface, Feature Dependencies.

1. MOTIVATION

Software Product Line (SPL) emerged as a prominent technology to generate tailored programs (products) from a set of reusable assets, speeding up the software development process [6]. SPL-based development focuses on the software decomposition into modular units of functionality defined as features. A feature is a semantically cohesive unit of behaviour of a software system. Then, features are used to describe the commonalities and variabilities of the products of a SPL [6]. Thus, to generate a tailored program, stakeholders select the features that satisfy their requirements.

Feature Dependency vs. Maintainability. SPLs typically have many features, and each feature may have from dozens to thousands of lines of code. Differently from stand-alone programs, the implementation of features is often scattered throughout several code units (e.g., classes and methods) [9]. A challenge faced in SPL maintenance is to identify, buried into hundreds of lines of code of several code units, the realisation of feature dependencies. A feature dependency naturally occurs, at source code level, whenever program elements within the boundaries of a feature depend on elements external to that feature, such as an attribute defined in one feature and used in another feature. However, developers have to devote a large extent of their time during a SPL feature maintenance in order to identify those feature dependencies. More importantly, after the identification of dependencies, it is also not trivial to understand the multiple purposes of some of them. So, all feature code related to the maintained feature might need to be revisited and/or changed. Even worse, missing or misunderstood dependencies lead to maintenance side effects, such as fault introduction in dependent features and incomplete fixes [15].

A Motivating Example. Figure 1 illustrates an example of the complexity to identify and understand feature dependencies in SPL implementation. Moreover, it is shown maintenance side effects caused due to the presence of this feature dependencies. In this example, features are implemented through the well known mechanism of conditional compilation (i.e. `#ifdef` directives). Figure 1 (a) shows part of the code of features *F1*, *F2* and *F3* implemented in class *C1*. Figure 1 (b) shows part of the code of feature *F3* implemented in class *C2*. In this example, there is a dependency between features *F1* and *F2* realised through the access to the attributes *v1*, *v2* and *v3*, that belongs to feature *F1*, in the method *m1* of the feature *F2*. Moreover, there is a dependency between *F1* and *F3* realised by means of (i) the use of the attributes *v5*, *v6* of feature *F1* in the method *m2* of *F3*, and (ii) the use of *v4* in the method *m3* of *F3* (Class *C2*, Figure 1 (b)). Clearly, this ad hoc visual identification of members that realise a dependency is only feasible in small SPLs.

More importantly, after identifying the members that participate of the dependencies, developers need to understand

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591089

the extent of the impact of a maintenance in the code that affect these members. For instance, a change in the type of the attribute *v4* (line 04, Figure 1 (a)) from `string` to `string[]` might require from the developer to revisit the code of *F3*, since there is a dependency between *F1* and *F3* realised by means of *v4* and other members. However, the maintenance of *v4* only affects a small part of the code of *F3*, thus making developers to waste their time revisiting code that are irrelevant for the maintenance. Even worse, if the developer miss that *v4* is part of the dependency between *F1* and *F3*, a fault can be introduced. In our example, the method *m3* of class *C2* will always return `null` after the maintenance, thus introducing an unexpected behaviour.

```
01. public class C1{
02.  #ifdef F1
03.   int v1=0; int v2=3, int v3=5;
04.   public static string v4="";
05.   float v5=0, v6=0;
06.  #endif
07.  ...
08.  #ifdef F2
09.   public void m1(){
10.   //use v1, v2, v3
11.  }
12.  #endif
13.  ...
14.  #ifdef F3
15.   public void m2(){
16.    //use v5, v6
17.   }
18.  #endif
19. }
```

(a) Class C1

```
01. public class C2{
02.  #ifdef F3
03.   public int m3(){
04.    ...
05.    if (obj.v4.equals(""))
06.     //return an int value
07.    else
08.     return null;
09.   }
10.  #endif
11. }
```

(b) Class C2

Figure 1: Example of a SPL code using conditional compilation to implement variabilities.

Feature Modularity. In the source code of a SPL implementation, the boundaries of a feature are often not the same as the module in a program [10]. The feature code may be scattered to each other and share elements such as attributes and methods [9]. In this context, this lack of intended modularity presented in feature implementation is one of the main cause of the absence of a structured and explicit feature interface. A feature interface is supposed to help identifying and understanding communications between features. The fully recognition of the program elements that are realising feature dependencies (i.e., the so-called implicit feature interface) is essential to the proper identification and understanding of these dependencies. However, the absence of such structured and explicit feature interface leads to difficulties for identifying and understanding dependencies, thus causing many maintenance problems as earlier described.

2. PROBLEM STATEMENT AND RELATED WORK

Given the motivating context presented earlier, the problem we aim to tackle in this PhD research is twofold. First, there is a need to comprehensively explore the nature of feature dependencies in order to identify their relevant properties that may negatively impact SPL maintenance. Second, there remains a need for enhancing the representation of implicit feature interfaces. In order to gather the requirements for feature interface representation, we must empirically observe these relevant properties that impact the identification and understanding of features dependencies. In the following, we detail the problem and limitations of related work.

Characterising Feature Dependencies Properties. Feature dependencies entail new dimensions of complexity in the SPL maintenance. It is known that feature dependencies are related in some way to SPL maintenance problems [13]. However, there is no conceptual framework that characterizes feature dependency properties in the source code. As a result, it is not possible to study which of them may impact SPL maintenance. Therefore, our *first problem* consists in revealing the properties that characterise feature dependencies and that may exert an impact on SPL maintenance. An analysis of the motivating example (Section 1) demonstrates the need for extensive reasoning about the feature code involved in a dependency. A number of properties of the feature dependency, independently from specific programming technique used, need to be considered. For instance, it is required to identify which program elements are in the *scope* of a feature dependency. In other words, which program elements are likely to be affected by modifications in the feature code. Another example is the *interface size* which is the number of program elements that are members of the implicit interface of a feature. A large *interface size* indicates a hazardous feature, since a maintenance in this feature is likely to affect many other related feature code.

Lack of Modular Reasoning. Our *second problem* is to improve feature modularity by enhancing feature interface representation. The difficulty in identifying and understanding the elements that configure feature dependencies, the so-called implicit feature interface, is related basically to the lack of the intended modularity of features [10]. It is not known what should comprise a feature interface, and how it can be structured to externalise relevant properties of feature dependencies regarding SPL maintenance. Thus, due to the absence of a structured and explicit feature interface, developers are likely to ignore unconsciously the dependencies between features while reasoning about maintenance of SPL features. This leads to maintenance problems that arise in the presence of feature dependencies. A careful analysis of Figure 1 gives us some insights on how the absence of a structured and explicit feature interface may lead to problems during the maintenance. For instance, if the developer miss that *v4* is a member of the implicit feature interface of *F1*, a fault can be introduced. The method *m3* of the class *C2*, that is not under maintenance, will always return `null` after the maintenance, thus introducing a faulty behaviour.

Limitations of Related Work. There are several studies dealing with dependencies and interactions in software by different levels of abstractions [1, 7, 8]. However, these studies are not directly related neither to the first nor to the second problems presented earlier. To the best of our knowledge, there are only few studies emphasising maintenance problems that arise in the presence of feature dependencies. The problem is that all studies developed so far related to our *first problem* tend to carry out a narrow analysis, focusing basically on the correlation between feature dependency and maintenance problems. For instance, Ribeiro et al. performed an empirical evaluation on preprocessor-based SPLs focusing on the maintainability and faults in the presence of feature dependencies [13]. The limited conclusions of related studies are largely due to the lack of knowledge of characterising feature dependency properties in the source code. These properties are all focused on measuring language-based structures rather than properties of features and their dependencies. So, such studies do not provide insights on which feature dependency properties must be captured in the structural organisation of explicit feature interfaces.

Regarding our *second problem*, modularity of feature representations has been a long standing goal of feature-oriented software development [10]. Lately, there has been considerable research effort to propose solutions that improve the modularity of features. Examples of solutions encompass from architecture-based SPLs that use frameworks or components [2] to feature-oriented programming [12]. Despite the improvement of feature modularity in these cases, there is a lack of studies driving efforts towards feature interfaces. To the best of our knowledge, Ribeiro et al. proposed the pioneer work dealing with interface in SPL features [13]. In this work it is proposed the concept of emergent interfaces for parts of the code under maintenance in SPLs implemented with conditional compilation. In spite of the interface generation, such approach do not externalise feature dependency properties in feature interface. Moreover, emergent feature interfaces are not organised to improve modular reasoning.

3. QUESTIONS AND HYPOTHESES

Our main goal is to enhance feature interface representation in order to support SPL code maintenance by focusing on feature dependency problems. In this context, we define our research questions as follows.

RQ1. *How to comprehensively understand the impact of feature dependency code on SPL maintenance?*

RQ2. *How to enhance feature interfaces in order to support the SPL code maintenance?*

We narrow our research questions by formulating the following research hypotheses:

H1. *There are characterising properties of feature dependency that exert impact on SPL maintenance;*

H2. *A structured and explicit feature interface:*

(i) *reduces maintenance problems that arise in the presence of feature dependencies; and*
(ii) *reduces the code maintenance effort.*

To answer RQ1, we identify characterising feature dependency properties by means of exploratory studies, and to test H1 we must study the correlation of these properties with SPL maintenance problems. The answer to RQ2 involves a proper structural organisation of feature interfaces in order to support SPL maintenance. To test H2, we should conduct empirical studies in order to reveal the improvements (and drawbacks) on the use of an enhanced feature interface.

4. RESEARCH METHOD AND PROGRESS

In order to address our research questions, firstly we have carried out some exploratory studies [3, 4] to understand the causes and consequences of the impact of feature dependencies on the maintenance of the code of SPL features. This will be achieved by identifying feature dependency properties that exert impact on SPL maintenance rather than language-based structure, such as class or methods. It is worth to notice that, as an essential procedure of our research method, we consider different programming techniques used to implement SPLs in some of our studies.

Secondly, we need to tackle the maintenance problems that arise in the presence of feature dependencies by providing means to support the developer during the SPL maintenance. In this case, we tackle such problem by enhancing the representation of feature interfaces in order to support the maintenance of SPL features. This enhancement will be achieved basically by structuring implicit feature interfaces, which are otherwise large and not cohesive. Feature dependency properties identified in our exploratory studies must be considered in this step.In the following, we detail the main steps to address our research questions.

4.1 Identifying Feature Dependency Properties

To answer RQ1, we conduct an exploratory study [4] to analyse the effectiveness of using our suite of feature dependency metrics as indicators of feature code instability in SPLs. This study compared specific feature dependency metrics against conventional language-based coupling metrics (CK metrics [5]) commonly used in SPL empirical studies. These specific feature dependency metrics quantifies two properties of feature dependencies (i.e. agnostic of programming technique) that exert impact on the stability of SPLs. To quantify these properties, we defined two metrics for each property (total of 4 metrics). The study was carried out considering two different programming techniques used for implementing the target SPLs. Our results basically shown that our metric suite outperformed CK metrics in SPLs. This result support the hypothesis H1 by giving us evidence that there are properties of feature dependency that exert impact on SPL maintenance.

In another work [3] related to RQ1, we analysed instabilities caused by feature dependencies in evolving SPLs implemented in three different programming techniques. In this study we are applying metrics related to feature dependency properties identified in the aforementioned study to have a fair comparison between different programming languages. As one of the main results until now, it was observed that the negative impact of feature dependency on SPL code stability is not problem of a specific programming technique. One feature dependency property (*scope* property) considered in this study was the most determinant factor that indicates instabilities in SPLs, thus supporting hypothesis H1.

Currently, we are working on study to complete this step and identify relevant feature dependency properties. This study aims to analyse the change propagation through feature dependencies during the evolution of a SPL. In particular, we want to identify other possible feature dependency properties that exert impact on SPL maintenance, such as feature dependency *purpose* . Moreover, we aim to understand whether feature dependencies are different in terms of change propagation and which feature dependency properties can better indicate change propagations.

4.2 Enhancing Feature Interfaces

In order to answer RQ2, we aim to provide a layer of abstraction on the source code which is composed of features and their interfaces. The goal of such an explicit representation is to help the understanding of feature dependencies. The representation will provide well-defined organisation of feature interface information. Based on our initial results and ongoing research (Section 4.1), we noticed that feature dependencies are complex and may present a multiple *pur-*

pose . In this way, only expliciting the elements realising these dependencies could make the feature interface not cohesive. So, to be useful for SPL maintenance, feature interfaces must go beyond than only expliciting their members.

Considering our motivational example (Section 1), expliciting that the dependency between *F1* and *F3* are realised by means of *v4*, *v5* and *v6* would give a hint for developers to revisit part of the code of a related feature that is in a module that is not under maintenance (method *m3* in class *C2* due to the use of *v4*). Moreover, if we indicate the multiple *purpose* of a feature dependency by showing that *v4* and *v5*/*v6* have different *purposes* in the realisation of the dependency, it is likely that maintenance effort reduction could be achieved. In other words, if we could indicate that the program elements in the *scope* of *v4*, and thus that are likely to be affected, are not the same of v5/*v6*, developers could drive efforts only to relevant parts of the code.

In this context, the organisation of the elements into explicit feature interfaces we are proposing relies on the idea of *Interface Segregation Principle* (ISP). The ISP states that *"clients should not be forced to depend upon interfaces that they do not use"* [11]. In this way, this principle proposes that complex interfaces must be segregated into more cohesive interfaces. We observed the idea of ISP from the point of view of maintenance and we argue that *developers should not be forced to understand parts of interfaces that are not useful to their tasks*. So, the main idea is to split the large and not cohesive implicit interface that features might have into smaller and more specific ones. This segregation will happen considering the properties of feature dependencies identified in our studies in a way that the resulting explicit feature interfaces support the maintenance of SPLs. Figure 2 presents a possible way to segregate a feature interface. In this case, the members that are realising the dependency between *F1* and *F2* are grouped. However, the members that realise the dependency between *F1* and *F3* are separated due to the multiple *purpose* of this dependency. The *scope* that may be affected by a maintenance in *v4* is not the same of *v5* and *v6*. So, even making part of the same dependency, *v4* is separated from *v5* and *v6*.

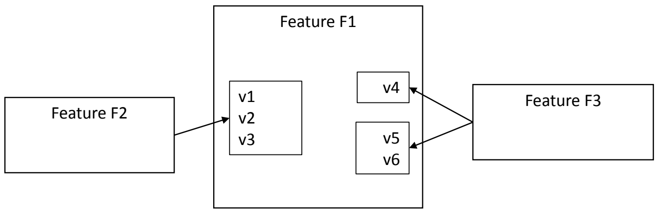

Figure 2: Structured and explicit interface of *F1*.

As a way to segregate the interface, we are considering to use the so-called slicing [14]. The process of slicing abstracts away parts of the program which can be determined to have no effect upon the semantics of interest. So, a program slice is defined as a set of program statements that may affect the values at some point of interest [14]. In our context, our points of interest are feature code that may affect a dependency behaviour. In this way, we want to detect which elements of a feature interface have the same point of interest, and thus organise them. To do so, we are studying slicing methods in order to adapt them to SPL context, and to consider the knowledge we obtained in our studies [3, 4] in these algorithms. Moreover, our slicing algorithm must disregard invalid combination of features in order to improve the scalability of the approach in industrial SPL.

5. EXPECTED CONTRIBUTIONS AND EVALUATION

In summary, our PhD research is intended to: (i) revealing properties that characterise feature dependencies and that exert impact on SPL maintenance; (ii) improve, by means of exploratory studies, the understanding about feature dependencies and their negative impact on SPL maintenance; (iii) improve the modular reasoning of features by enhancing feature interfaces; and (iv) provide a method to segregate feature interfaces in order to support the SPL maintenance.

Regarding the evaluation, we aim to investigate the correlation of feature dependency properties with SPL maintenance problems to test H1. To evaluate H2, empirical studies will be conduct to compare (i) the reduction of maintenance side effects, and (ii) the maintenance effort with and without our enhanced feature interfaces. Finally, we plan to publish our results to stimulate further replications.

6. ACKNOWLEDGEMENTS

This work was funded by CNPq PhD scholarship (grant number 141688/2013-0).

7. REFERENCES

[1] S. Apel et al. Detecting dependences and interactions in feature-oriented design. In *ISSRE '10*, pages 161–170. IEEE, 2010.
[2] L. Bass et al. *Software Architecture in Practice*. Addison-Wesley, 2003.
[3] B. B. P. Cafeo et al. Analysing the impact of feature dependency implementation on product line stability: An exploratory study. In *SBES'12*, pages 141–150. IEEE, 2012.
[4] B. B. P. Cafeo et al. Towards indicators of instabilities in software product lines: An empirical evaluation of metrics. In *WETSoM'13*, pages 69–75. IEEE, 2013.
[5] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
[6] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
[7] S. Ferber et al. Feature interaction and dependencies: Modeling features for reengineering a legacy product line. In *Software Product Lines*, volume 2379 of *LNCS*, pages 235–256. Springer Berlin Heidelberg, 2002.
[8] M. Geipel and F. Schweitzer. The link between dependency and cochange: Empirical evidence. *IEEE Transactions on Software Engineering*, 38(6):1432–1444, 2012.
[9] C. Kästner et al. Visualizing software product line variabilities in source code. In *ViSPLE'08*, 2008.
[10] C. Kästner et al. The road to feature modularity? In *SPLC'11*, pages 51–58. ACM, 2011.
[11] R. C. Martin. The interface segregation principle: One of the many principles of OOD. *C++ Report*, 8(1):30–36, 1996.
[12] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97*, volume 1241, pages 419–443. Springer Berlin Heidelberg, 1997.
[13] M. Ribeiro et al. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *GPCE'11*, pages 23–32. ACM, 2011.
[14] M. Weiser. Program slicing. In *ICSE'81*, pages 439–449. IEEE, 1981.
[15] Z. Yin et al. How do fixes become bugs? In *ESEC/FSE'11*, pages 26–36. ACM, 2011.