Handling Optional Features in Software Product Lines

Thomas Leich, Sven Apel, Marko Rosenmueller, Gunter Saake
Department of Computer Science, Otto-von-Guericke-University
Magdeburg, 39106, Germany

Abstract

Software product lines have a long tradition and will gain momentum in future. *Feature Oriented Programming (FOP)* is a design methodology and implementation technique to build product lines based on features. Features interact with others in many ways. In this paper we focus on structural interactions of features and especially on handling optional features. We present our first ideas dealing with a problem in this context (a.k.a. *feature optionality problem*).

1 Overview

Software product lines are subject of ongoing research. FOP is a design methodology and implementation technique to build product lines [2]. The idea of the FOP model is to decompose software into separate modular units (features) and to compose stacks of features to derive a concrete program. When adding new programs to a product line, existing features of other programs can be reused. This is also known as *step-wise refinement*. The benefit is maintainable, comprehensible software that can be easily reused, configured and extended [2].

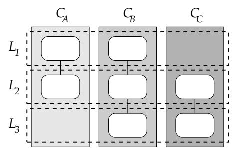

Figure 1: Stack of mixin layers.

Mixin Layers. *Mixin layers* are one appropriate technique to implement features in the sense of FOP [2, 8]. The basic idea is that features are often implemented by a collaboration of class fragments. A mixin layer is a static component, encapsulating fragments of several different classes (*mixins*) so that all fragments are composed consistently. Figure 1 depicts a stack of three mixin layers (L1 - L3) in top down order. The mixin layers crosscut multiple classes (C_A - C_C). The rounded boxes represent the mixins. Mixins that belong together constitute a complete class are called refinement chain.

FeatureC++. FeatureC++ [1] is a C++ language extension to implement mixin layers. In FeatureC++, mixin layers are represented by file system directories. Therefore, they have no programmatic representation. Those mixins found inside the directories are assigned to be members of the enclosing mixin layers. Figure 2 depicts a class and two refinements (implemented as mixins). Refinements are declared by the *refines* keyword (Lines 10,17). Usually, they introduce new attributes and methods or extend methods of their parent classes (e.g. Line 10-27). To access the extended method the `super` keyword is used (Lines 20,23,26).

[1]http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

2 Feature Optionality Problem

To explain our ideas to solve the feature optionality problem we use a FOP model of a stack product line (adopted from [7]). Figure 2 shows a modified version of this stack. The base feature `base` implements basic stack operations such as `push()`,`top()`, and `pop()`. The `concat` feature extents the stack with functionality for combining two stacks. The `concat` method adds this functionality (Line 11).

```
 1  //Layer ../Stack/BaseStack/BaseStack.fcc
 2  class stackOfChar {           // the base feature
 3      String s;
 4      void empty() {s = "";        }
 5      void push ( char a ) {s = a + s;}
 6      void pop() {s = s.substring(1);}
 7      char top() {return s.charAt(0);}
 8  };
 9  //Layer ../Stack/Concat/Concat.fcc
10  refines class stackOfChar {  // the concat feature
11      void concat(stackOfChar& other) {
12         while (!other.empty()) {
13             push(other.top());
14             other.pop();}}
15  };
16  //Layer ../Stack/Log/Log.fcc
17  refines class stackOfChar {  // the log feature
18      void concat(stackOfChar& o) {
19        cout << "concat␣2␣stacks" << endl;
20        super::concat(o);}
21      void push(char a) {
22        cout << "push:␣" << a << endl;
23        super::push(a);}
24      void pop() {
25        cout << "pop:␣" << top() << endl;
26        super::pop();}
27  };
```

Figure 2: Refinements.

The feature `log` adds logging support to the stack product line. Doing so, the basic stack operations (`push()`, `pop()`, `concat()`) are refined. This is the common way to implement refinements in a FOP style. But, however, in certain situation this leads to a problem: If we want to derive a stack with logging support and without the `concat` feature, we get an error during the compilation process. If the `concat` feature is removed the `log` feature tries to extent a non-existing feature. This may happen because the `concat` feature is an *optional* feature. This problem is also called the *feature optionality problem* caused in feature-oriented designs [7].

Prehofer proposes *lifters* to solve this problem in FOP [7]. Lifters encapsulate feature dependencies and decouple features. The idea is to separate those parts of a feature that depend on other features (in lifters) from those that are independent. Doing so, a feature is split into several abstract (sub) features. The goal is that dependent and independent features are separated into different concerns [7, 4].

However, this approach has its limitations: It is a main goal of FOP to encapsulate user intuitive requirements into features, and moreover in a one-to-one pattern [3]. Using lifters enforces the programmer to introduce additional abstract features. Thus, the one-to-one mapping is broken. Furthermore, this prevents *feature cohesion.* Cohesion is the property of a feature to encapsulate all implementation units that contribute to the feature in one module [5]. Feature cohesion is important to implement and handle large-scale building blocks. The explicit mapping of requirements onto implemented features ensures a long-life, maintainable system [9]. Features in FOP have a high degree of cohesion, but this introduces the feature optionality problem. Using the lifter to cope with the feature optionality problem breaks this cohesion through separating interaction and basic concerns into different units.

3 Our First Approach

This section presents our first solution to overcome the feature optionality problem without depending on lifters. Our main goal is to reach a high tolerance against optionals feature in FOP style product lines. Our approach is based on ideas of *Aspect-Oriented Programming (AOP).* AOP tackles the feature optionality problem as follows: By expressing join points in pointcuts the programmer is able to define wildcards that are robust against changes of features and their compositions. Using languages like Caesar [6] or FeatureC++ [1] that combine FOP and AOP the programmer can decide which functionality is implemented by using aspects or mixins. In this way he can select the adequate technique to implement a feature hierarchy that is tolerant and reliable against changes and optional features. This allows for a coherent encapsulation, but the separation of this optional functionality into aspects and mixins look still like a hack. Therefore, our approach tries to improve mixins and mixin-based inheritance themselves to cope with optional features. So the programmer does not need aspects. That does not mean that we generally want to program without aspects. We argue that aspects have a lot of strengths to implement product lines [1], but regarding the feature optionality problem we tailored solutions.

Our approach introduces well known AOP concepts into mixins. Mixin may declare their methods with the keywords `before`, `after` and `around`. This emphasizes that these method refinements are optional. This means that if there is no method to refine, these refinements are ignored. Furthermore, this extension makes the idea of adding refinements before, after, and around a join point (a call to a method) more explicit that the common FOP way (by using the `super` keyword inside the refined method body). An alternate would be the introduction of an `optional` keyword that states that a refinement is optional.

In our approach (using `before`, `after`, and `around`) the pointcut is implicitly defined by the signature of the refined method. Figure 3 shows the `log` feature implementation using our new extension. Now, the `concat()` method is optional by using the `before` keyword (Line 3). Moreover, the keyword also describes when the functionality of the refinement has to be processed (in an AOP style). The `super` keyword (Lines 7,10) is still used for refinements of mandatory features. This distinction between mandatory and optional features allows us to find design failures statically during the composition of features. An alternative solution uses the `around` keyword (see Fig. 4).

```
 1  //Layer ../Stack/Log/Log.fcc
 2  refines class stackOfChar {
 3     void concat(stackOfChar& other) : before() {
 4          cout << "concating␣2␣stacks" << endl;}
 5     void push(char a) before() {
 6          cout << "push:␣" << a << endl;
 7          super::push(a);}
 8     void pop() {
 9          cout << "pop:␣" << top() << endl;
10          super::pop();}
11  };
```

Figure 3: Optional Method Refinement.

```
   ....
   void concat(stackOfChar& other) : around() {
   cout << "begin␣concating␣2␣stacks" << endl;
   proceed(other);
   cout << "end␣concat" << endl;}
   ....
```

Figure 4: Optional Method Refinement using `around`.

Up to now we only investigated single classes. Optional features may also introduce new classes that are refined by subsequent features. To be independent from these features as well we introduce the `optional` keyword to the definition of class refinements. Thus features can add an optional class refinement by using `refines optional class`. We have limited the access to these optional classes only to those classes that contribute to the enclosing mixin layer and to those methods that are optional.

4 Further Directions

Feature interactions are not limited to only two features. It is also natural that different combinations of two or more optional features may interact in different ways. Therefore, we intend to improve our current solution. Currently, we investigate *multi mixins* [1] to deal with this problem.

References

[1] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *GPCE'05*, 2005.

[2] D. Batory, J. N. Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6), 2004.

[3] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[4] J. Liu, D. Batory, and S. Nedunuri. Modelling Interactions in Feature Oriented Software Design. In *ICFI'05.* 1997.

[5] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technlogies. In *ECOOP '05*, 2005.

[6] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004.

[7] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP'97*, 1997.

[8] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM TOSEM*, 2002.

[9] P. Sochos, I. Philippow, and M. Riebisch. Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. In *Object-Oriented and Internet-Based Technologies.* 2004.