Current State and Future Challenges in Optional Weaving

Constanze Adler
Department of Computer Science
Otto-von-Guericke University
Magdeburg, Germany
constanze.michaelis@st.ovgu.de

***Abstract*—In software development, software product lines become more and more attractive. Feature- and aspect-oriented programming are techniques to realize software product lines very comfortable. This paper deals with a well known issue among software product lines: The feature optionality problem, which occurs when features depending on each other are optional. One possible, promising solution to this issue, the optional weaving will be discussed. Minimal implementations of common software are an ambitious aim of software product lines. Realizing optional features with optional weaving will bring shortend resultant code, because of unnecessary omitted code.**

I. Introduction

Feature-oriented programming (FOP) and aspect-oriented programming (AOP) are solutions for realizing software product lines. The idea of both methods is developing software according to a domain and implementing features which describe concerns of the software for domain, not programming, purpose. With these programming paradigms the separation of concerns [1] can be realized, which is one of the major design principles with respect to software product lines. Developing software with software product lines offer a lot of advantages. These are for example customized programs, shortend development periods and reusable code. Features, defined within software product lines, can be mandatory or optional and in many cases they depend on each other. These dependencies lead to the feature optionality problem and therefore this paper deals with the current state of this problem and a possible solution called optional weaving.

Optional weaving brings shorter resulting code, which can be essential for embedded systems, like smart cards, mobile phones or sensor networks. These embedded systems are characterized by less storage, less processing power and battery-operation. Therefore it is necessary to develop tailor-made solutions [2], [3]. The objective of this paper is to survey the current state of solutions to the feature optionality problem, whereas the concentration on optional weaving will be made. With the current state of optional weaving there is a partial solution to the feature optionality problem within AspectJ. With FeatureC++ it is possible to have two interacting optional features, which is a partial solution to the problem.

First of all, this paper gives a brief background about aspect and feature-oriented programming. Afterwards the feature optionality problem will be discussed and the current approaches in optional weaving will be introduced. Due to the fact, that this approach is in its infancy, suggestions for improvement will be made in the section about Future Challenges. Finally the paper will be summarized and concluded.

II. Background

This section gives a very brief review on techniques to implement software product lines like AOP, FOP and the connection of both.

A. Aspect-oriented programming

AOP aims at separating crosscutting concerns, i.e. code of an implemented feature is scattered across multiple components [4], [5]. According to Kiczales et al. [6] the idea of AOP is to implement the crosscutting concerns (also called crosscuts [4]) as aspects to eliminate code tangling and scattering. The core features are implemented with traditional design and implementation concepts, just pointcuts, advice and the aspect weaver build the program consisting of aspects, representing the additional features, and the core.

B. Feature-oriented programming

FOP aims at feature traceability. According to Apel et al. [7] the idea of FOP is to build a program by composing features, where the feature refines another feature incrementally which leads to a step-wise refinement. Features are mapped within a feature model and composed by the mixin approach within the AHEAD[1]

[1]Algebraic Hierarchical Equations for Application Design

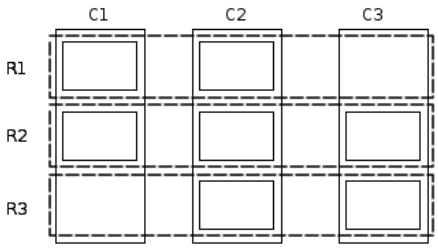

Figure 1. Roles and Collaborations

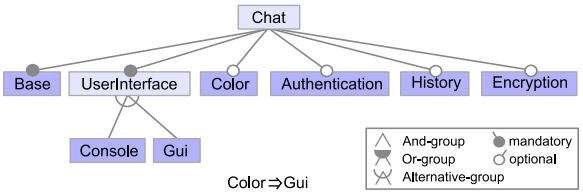

Figure 2. Feature Model Chat Program

toolsuite, which is a prominent example for the Java Community [7].

A *Feature model* is the result of the analysis of the commonality and variability of a domain perspective [2] and displays the feature dependencies in a graphic representation.

Jak is a programming language which represents a superset of the Java language used in the AHEAD toolsuite for composing features [8]. It is a Java extension for metaprogramming, state machines and refinements.

Mixin is one appropriate technique to build a feature-oriented program. The basic idea is that features are not implemented inside one class, but within a set of collaborating classes. Due to [9] classes play different roles in different collaborations. For a better understanding Figure 1 shows a model of the mixin approach. The roles (R1..R3) represent the collaborations and the classes (C1..C3) implement the features. One collaboration represents one feature within a set of classes.

According to current research the trend goes to the combination of FOP and AOP for implementation of software product lines, because of the compensation of the FOP disadvantages with the AOP advantages. The modularity of FOP for the base features and the encapsulation of AOP for the crosscutting concerns. The group around Apel realizes this with Aspectual Mixin Layers and Aspectual Mixins (for further information see [9], [10], [7], [5]). Similarly Lee et al. [4] persuit the combination of both for software product line development with respect to the mentioned advantages.

Now having a short summary of the techniques to implement software product lines where features are the central elements, the next section will discuss the problem with optional features interacting with each other.

III. Feature Optionality Problem

First of all, this section introduces feature interactions and feature dependencies. Secondly the feature optionality problem will be discussed.

A. Feature Interactions and Feature Dependencies

Feature interactions occur when one or more features modify or influence another feature. There are many ways in which features can interact whereas the optional weaving focuses on a particular form of interactions that are static and structural, i.e. influencing or changing the source code of another feature [11]. Due to the fact that features interact with each other, their dependencies are classified by [4] to configuration and operational dependencies. Configuration dependencies determine whether a feature is *required* or *excluded* from configuration. Lets come to an example shown in the feature model on Figure 2 represents a short chat application. This example indicates a configuration dependency. The *Color* feature can just be selected if the *Gui* feature has been selected. Operational dependencies describe the feature interactions. Therefore operational dependencies include usage, modification and activation dependencies, thus the interaction between *Authentication* and *Encryption* as will be presented in Section III. These operational dependencies are considered in optional weaving according to optional features.

B. Feature Optionality Problem

The feature optionality problem is a well known problem among software product line development handling optional features [12], [13], [11]. This problem occurs according to Kästner [13] when multiple optional features interact with each other, e.g. feature A refers to feature B or feature B extends feature A. This refers to the crosscutting concerns within software product lines mentioned above.

The small chat example shown on Figure 2 with just rudimentary implementation (Figure 3) to illustrate the feature optionality problem. Figure 3 represents the composed class refinements of the `Server` class, which is located in the *Base* feature (see Figure 2). From line one to five the base implementation is shown, where a method `logIn` is defined. This method will be refined by two features: *Authentication* and *Encryption*. The `logIn` method of the

```
1  SoUrCe RooT Base "workspace/FOPChat/src/Base/Server.jak";
2  abstract class Server$$Base {
3   public void logIn(Connection c, TextMessage msg) {...}
4   ...
5  }
6  SoUrCe  Encryption "workspace/FOPChat/src/Encryption/Server.jak";
7  abstract class Server$$Encryption extends  Server$$Base  {
8   public void logIn(Connection c, TextMessage msg)
9   {
10   msg = Encryption.decrypt(msg);
11   Super().logIn(c,msg);
12  }
13 }
14 SoUrCe  Authentication "workspace/FOPChat/src/Authentication/Server.jak";
15 abstract class Server$$Authentication extends  Server$$Encryption  {
16  public void logIn(Connection c, TextMessage msg)
17  {
18   Super().logIn(c,msg);
19   c.enableClient(this.checkPassword(msg));
20  }
21  private boolean checkPassword(TextMessage tm){...}
22 }
```

Figure 3. Refinements

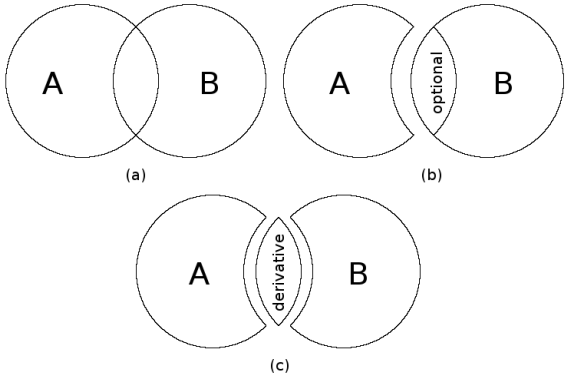

Figure 4. (a) Optional Features, (b) Optional Weaving Approach, (c) Derivative Feature Approach [13]

Authentication feature represents the verification of the password to login and to start the chat (see lines 14 to 22). The `logIn` method of the *Encryption* feature represents the decryption of the Message consiting of the password before it can be verified. Because of the dependency between *Authentication* and *Encryption* and the fact that *Encryption* has to refine `logIn` method before *Authentication* this method can not be introduced by *Authentication*, but has to be introduced by base class `Server`. This results in code replication and shows the feature optionality problem.

In Figure 2 are four optional features: *Color*, *Authentication*, *History* and *Encryption*. If *Encryption* and *Authentication* is selected then *Authentication* must implement the *Encryption*. The authentication message has to be sent encrypted, if the *Encryption* feature was selected, to ensure a secure transfer of the authentication message. If just *Authentication* is selected the *Encryption* algorithms are not required, but what, if *Authentication* is not selected and *Encryption* wants to refine the not existing *Server* Method `logIn` (see Figure 3)? The system will fire an error with this selection. Furthermore History, which does some logging functionality, also must implement *Encryption* if it is selected simultaneously. This very small example shows that there is a need to overcome the stated feature optionality problem.

Figure 4 illustrates the feature optionality problem regarding two interacting features (a) and approaches to overcome this problem. One of these approaches is the derivative feature approach [14] (c), where the interacting code is swapped to a derivative, which will be implemented when feature A and feature B is selected (see Section VI).

Optional Weaving is the other approach which is a solution to the stated feature optionality problem. This approach, firstly introduced by Leich et al. [12], is the implementation of optional interactions within the features, but with language constructs as advice statements that are ignored when the second feature is not selected. Therefore programs generated out of software product lines contain just the necessary code which is reduced to a minimum.

Here the issue of feature optionality was discussed. In the next section the current state of the optional weaving approach will be shown.

IV. Optional Weaving - Current State

This section refers to optional weaving implemented with two programming languages. First of all, the FeatureC++ approach will be described and afterwards the AspectJ approach.

A. Optional Weaving with FeatureC++

The first approach within FeatureC++ of Leich et al. [12] was based on ideas of aspect-oriented programming. As stated in the section before, the optional parts are implemented in the original feature and these parts are woven when necessary, i.e. the optional part is woven, when the interacting feature is selected. Using **`before`**, **`after`**, and **`around`** the pointcut is implicitly defined by the signature of the refined method. Figure 5 from [12] shows a log feature implementation using this new extension. The `concat()` method is optional by using the keyword **`before`** and it also describes when the functionality will be processed (AOP Style). The **`super`** keyword is still used for mandatory features.

B. Optional Weaving with AspectJ

A similar approach was discovered by Kästner [13] with the Java equivalent AspectJ. It was shown that optional weaving avoids the need of creating derivative features resolving dependencies unlike the derivative approach shown in Figure 4(c). Another advantage

```
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 5. Optional Method Refinement [12]

```
1  public class Server {
2   public void broadcast(TextMessage tm)  {...}
3  }
```

```
4  aspect Authentication{
5   void Server.logIn(Connection c,TextMessage msg){...}
6   around(Connection c, Object o) :
7     call (* Connection.send(TextMessage)) && withincode(* Server.broadcast
          (TextMessage)) && target(c) {...
8               c.server.logIn(c, tm);...
9               }
10 }
```

```
14 aspect Encryption{
15  TextMessage encrypt(TextMessage tm){...}
16  TextMessage decrypt(TextMessage tm){...}
17  before(Connection c, TextMessage tm): call (* Server.logIn(Connection,
        TextMessage)) {
18        tm = this.decrypt(tm);
19  }
20 }
```

Figure 6. Optional Weaving with AspectJ

has been pointed out, that the dependencies are implemented as optional extension within the genuine feature so it can be maintained locally.

The implementation of the in Section III introduced example with this approach is shown in Figure 6: The aspect `Encryption` advises the Method `logIn` which is introduced by the aspect `Authentication` . When the `Authentication` aspect is not included in the compilation this advice statement is not woven. With this approach the implementation dependencies can be resolved and features can be composed individually.

In the case study of Kästner [13] the optional weaving approach was compared to the derivative approach (see Section VI). With the optional weaving approach in contrast to derivative approach, there is no problem with scale, no need of tools for hiding complexity and the interactions are woven self-acting by AspectJ compiler, when the dependent feature was selected.

In contrast to the advantages resulting from the in [13] given case study, big lacks in current AspectJ have been found, so it is unusable for the optional weaving approach. First of all it is not feasible to reference optional classes, methods or member variables in optional advice statements, which result in code replication. Secondly this approach can only be used for advice statements and not for inter-type member declarations. Furthermore these stated scope problems impede advising methods in optional classes. Finally in [13] was pointed out, that because of these lacks in the current version of AspectJ ”the Optional Weaving Approach cannot be used as a solution for the feature optionality problem” in the given case study.

In this section was stated what approaches where made to implement the idea of optional weaving and which problems are still left. Therefore the next section will give some possible solutions to enhance the approach of optional weaving to overcome the feature optionality problem potentially.

V. Future Challenges

This section will emphasize the possibilities with AspectJ to overcome the problems stated in the section above. Furthermore another approach, that is not directly related to optional weaving, will be briefly introduced to suggest another way to implement optional weaving with CaesarJ.

A. Future directions with AspectJ

As stated in the section before, AspectJ is not able to cope with the problems of referencing optional classes, methods or member variables and can only handle advice statements and no inter-type member declarations. In [13] was shown, that with the current AspectJ language and its restrictions it is ”hard or even impossible to implement optional interactions with this approach.” Therefore it was suggested to define groups of statements (advice and inter-type member declaration) as optional refinements, where the entire group has transactional semantics, i. e. either is woven with all advice or no statement is woven. Therefore it is suggested to introduce an `optional` keyword to define the optional part within the feature.

B. Possible directions with CaesarJ

Within [15] another approach with CaesarJ in contrast to AspectJ is discussed. They show, that features can be freely composed in CaesarJ by instantiating and deploying instances of the mixed classes representing a bound feature. CaesarJ has a mechanism that specifies in which context the advice definitions must be activated. This mechanism is called *deployment* (for more information see [15]). Additionally, CaesarJ has also a concept of bidirectional interfaces, which supports reusable features. Owing to this promising already in this language integrated appendages, which are still missing in AspectJ, we suggest to evaluate the optional weaving approach with CaesarJ.

Here the future ways for optional weaving were indicated. The next section will introduce another concept to overcome the feature optionality problem.

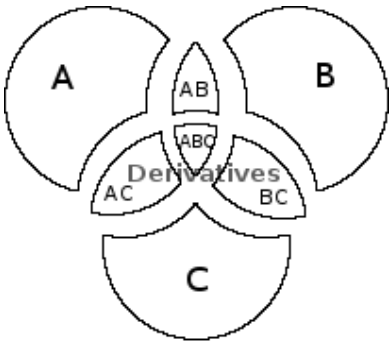

Figure 7. Derivative Feature Approach with three Features

VI. Related Work

Within this section an overview of the to the optional weaving related approach of Derivatives which is illustrated in Figure 4 and Figure 7 will be given.

A. Derivative Feature Approach

Another approach to overcome the feature optionality problem is the derivative feature approach. In Lee [4] and Liu et al. [14] the solution is to separate the feature dependencies. In these works one suggestion is to encapsulate the feature dependencies into aspectual components. Secondly there is suggested to seperate the dependencies between features from components implementing the core functionality.

In [13] the approach of feature derivatives was evaluated, where the interactions or dependencies are encapsulated within derivatives as abstractly shown in Figure 7. When the number of features increases, also the number of derivatives will increase. Therefore [13] distinguishes between higher and lower order derivatives, whereas higher order derivatives (see Figure 7) are derivatives with more than one feature interaction and lower order derivatives (see Figure 4 (c)) are ones with one feature interaction. For first order derivatives this approach does not scale, but for the maximum number of derivatives a quadratical growth with the number of features is suggested by Liu et al. [14].

VII. Conclusion

Nowadays the feature optionality problem is still an insufficient solved issue, because optional weaving is not fully developed. This paper has shown the current state of the technical research and introduced the results of Leich et. al [12] and Kästner [13]. The experiment with the simple chat example with AspectJ has shown that optional code is possible, but not traceable without an defining keyword. As long as AspectJ has no capabilities of optional refinement, optional weaving is no alternative to the related derivative feature approach which was brought up short in Section VI. The issue of complexity according to scale, makes this approach to a solution for feature interactions of lower order [13]. Therefore it seems to be an equitable decision to overcome the feature optionality problem in software product lines. But, for interactions of higher order the development of optional weaving should be further promoted any way.

Acknowledgment

The author would like to thank all the helpful anonymous reviewers.

References

[1] A. Nyßen, S. Tyszberowicz, and T. Weiler, “Are Aspects useful for Managing Variability in Software Product Lines? A Case Study,” in *Early Aspects Workshop at SPLC 05*, 2005.

[2] L. Fuentes and N. Gamez, “A feature model of an aspect-oriented middleware family for pervasive systems,” in *NAOMI ’08: Proceedings of the 2008 workshop on Next generation aspect oriented middleware*. New York, NY, USA: ACM, 2008, pp. 11–16.

[3] M. Rosenmüller, T. Leich, S. Apel, and G. Saake, “Von Mini- über Micro- bis zu Nano-DBMS: Datenhaltung in eingebetteten Systemen,” *Datenbank-Spektrum*, vol. 20, pp. 33–47, 2007.

[4] H. Cho, K. Lee, and K. C. Kang, “Feature Relation and Dependency Management: An Aspect-Oriented Approach,” in *SPLC ’08: Proceedings of the 2008 12th International Software Product Line Conference*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 3–11.

[5] S. Apel, D. Batory, and M. Rosenmüller, “On the Structure of Crosscutting Concerns: Using Aspects or Collaborations?” in *GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, Oct. 2006.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-Oriented Programming,” in *ECOOP*, 1997.

[7] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, “FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming,” in *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, ser. Lecture Notes in Computer Science, vol. 3676. Springer Verlag, Sep. 2005, pp. 125–140.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer, “Scaling step-wise refinement,” in *ICSE ’03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 187–197.

[9] S. Apel, T. Leich, and G. Saake, "Aspectual mixin layers: aspects and features in concert," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 122–131.

[10] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, "Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution," in *In AMSE05, at ECOOP05*, ser. in 19th European Conference on Object-Oriented Programming (ECOOP'05), Jul. 2005, pp. 3–16.

[11] J. Liu, D. S. Batory, and S. Nedunuri, "Modeling Interactions in Feature Oriented Software Designs," in *ICFI 05: Feature Interactions in Telecommunications and Software Systems VIII*, S. Reiff-Marganiec and M. Ryan, Eds. IOS Press, 2005, pp. 178–197.

[12] T. Leich, S. Apel, M. Rosenmüller, and G. Saake, "Handling Optional Features in Software Product Lines," in *Proceedings of OOPSLA Workshop on Managing Variabilities consistently in Design and Code*, San Diego, USA, 2005.

[13] C. Kästner, "Aspect-Oriented Refactoring of Berkeley DB," 2007, Diplomarbeit, Otto-von-Guericke-Universität Magdeburg.

[14] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 112–121.

[15] M. Mezini and K. Ostermann, "Variability management with feature-oriented programming and aspects," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pp. 127–136, 2004.