Using Feature Dependency Analysis and AOP for Incremental Software Development*

Kwanwoo Lee

[1]Hansung University, 389, Samsun-dong-3ga, Sungbuk-gu,
136-792 Seoul, Korea
kwlee@hansung.ac.kr

Abstract. Incremental addition of features into a product may not be an easy task, as its addition may cause changes to many parts of existing components. Aspect-oriented programming (AOP) provides effective mechanisms for encapsulating crosscutting concerns and integrating them without modifying existing components. However, it is not sufficient for incremental software development. Feature dependency analysis provides essential information for incremental software development. By tightly coupling feature dependency analysis with AOP, we can support incremental software development. In this paper, we provide specific guidelines on how feature dependency analysis and an AOP technique (i.e., AspectJ) can be combined to support incremental software development.

1 Introduction

Features are considered as units of increments in requirements. When we incorporate a feature into a product, its implementation may affect several existing components. This problem partly comes from the fact that the unit of features does not always correspond to that of implementation components, i.e., the code implementing a particular feature may be scattered across multiple components. This crosscutting problem makes it difficult to incrementally add features into a product without modifying existing components.

To address this problem, we need to implement features that crosscut modular units (i.e., components) as separate entities and integrate them without affecting the rest of the system. Recently, aspect-oriented programming (AOP) languages, such as AspectJ [1], were proposed to clearly separate crosscutting concerns from modular components and modularize them as separate entities, which would otherwise be scattered across modular components. Using AOP, we can implement features that may crosscut several modular units as separate aspects. This approach makes it easy to incorporate crosscutting features into a product without modifying existing components.

* This Research was financially supported by Hansung University in the year of 2005.

However, one-to-one mapping between features and aspects is not sufficient for incremental software development. In AOP, aspects modify existing components statically (static crosscutting) or dynamically (dynamic crosscutting). If aspects implementing features are independent of each other, their inclusion does not cause problems. However, if they are not, their inclusion may cause changes to other parts of the system. The change to other (modular or aspectual) components is called an invasive change. Due to this invasive change problem, applying AOP with this simple mapping relationship results in insufficient support for incremental software development.

This invasive change problem mainly comes from lack of understanding of feature dependencies [2], [3], [4]. To address this problem, dependencies between features must be analyzed thoroughly before engineering of applications. Then dependencies between features must be decoupled from components implementing core functionality of features so that effects of feature addition can be localized.

AOP just provides effective mechanisms for encapsulating crosscutting concerns and integrating them without modifying existing components. On the other hand, feature dependency analysis [2] provides essential information for incremental software development. By tightly coupling feature dependency analysis with AOP, we can support incremental software development. In this paper, we provide specific guidelines on how feature dependency analysis results can be mapped to AOP to support incremental software development.

Problems with incremental software development using AOP are discussed in section 2. An overview of feature dependency analysis is presented in section 3. Section 4 presents detailed guidelines on how feature dependency analysis and AOP can be combined to support incremental software development. Section 5 concludes this paper.

2 Problems with Incremental Software Development Using AOP

Intuitively, AOP seems to be an effective way of incorporating additive features into a product, as it can isolate additive features into *aspects* and integrate them in an additive way. That is, based on an initial base structure implementing core features, additive features are modeled as aspects. Then, an aspect weaver produces products by weaving the base modular structure and aspects implementing additive features.

This kind of aspect-oriented software development, however, leads to the invasive change problem. Let's take an elevator example. Fig. 1 shows a simplified list of elevator features and the base component model implementing only core features. As *CallCancellation* is an additive feature, we can implement it as a separate aspect.

Fig. 2 shows an aspectual implementation of the *CallCancellation* feature, which deletes a call request if the call has already been registered. The pointcut in the *Cancellation* aspect intercepts a call to the *exist* method that returns a truth value. After returning from the matched join point, the aspect deletes the requested call. In this way, we can implement an additive feature as a separate entity and integrate it without modifying existing components.

Core features	AutoService	A normal driving service of an elevator
	CallRegistration	An operation that registers a passenger's call button request
additive features	VIPService	A special driving service for exclusive use of VIPs
	FireService	A special driving service for fire fighters
	CallCancellation	An operation that deletes a registered call button request
	Anti-Nuisance	An operation that deletes all registered calls if an excessive number of calls are registered compared with passenger load determined by the weight sensor

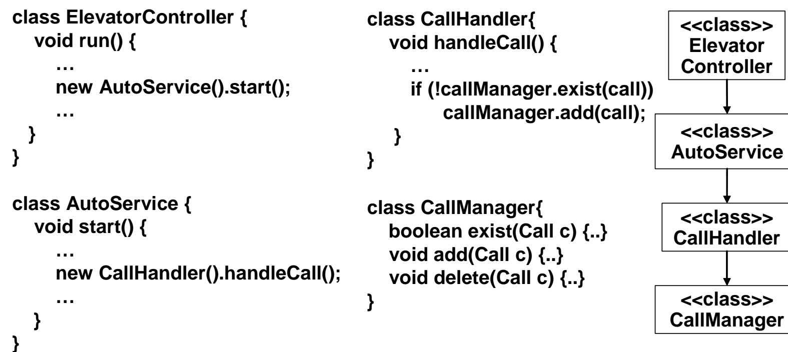

Fig. 1. An elevator example: the base component model

```
1.    aspect Cancellation {
2.        pointcut cancellationVP( CallManager manager, Call call) :
3.         call(* exist(Call)) &&
4.                 target(manager) && args(call) &&
5.                 if (manager.exist(call));
6.
7.        after(CallManager manager, Call call) returning:
8.                      cancellationVP(manager, call) {
9.                      manager.delete(call);
10.        }
11.  }
```

Fig. 2. Aspectual implementation of the *CallCancellation* feature

Similarly, *FireService*, which is a special driving service when a fire happens, can be implemented as a separate aspect. However, if the *FireService* feature does not require the activation of the *CallCancellation* feature, we have to slightly modify the *Cancellation* aspect by restricting the scope of the matched join point as below:

The last line in Fig. 3 restricts the scope of the matched join point occurring within a given control flow during the execution of the *start* method in the *AutoService* class. This change implies that the *FireService* feature cannot be incrementally added without modifying existing components. This invasive problem mainly comes from lack of understanding of dependencies between features. The next section analyzes various dependencies among the features shown in Fig. 1.

```
1.    pointcut cancellationVP( CallManager manager, Call call) :
2.         call(* exist(Call)) &&
3.          target(manager) && args(call) &&
4.          if (manager.exist(call)) &&
5.           cflow (execution(* AutoService.start()));
```

Fig. 3. Modified aspectual code for the *CallCancellation* feature

3 Feature Dependency Analysis

Features are not usually independent of each other. There may be various types of dependencies among features. In this paper, we mainly focus on operational dependencies among functional features, which were reported at [2]. Operational dependency means any implicitly or explicitly created relationship between features during the operation of the system in such a way that the operation of one feature is dependent on those of other features. These dependencies include *Usage*, *Modification*, and *Activation* dependencies.

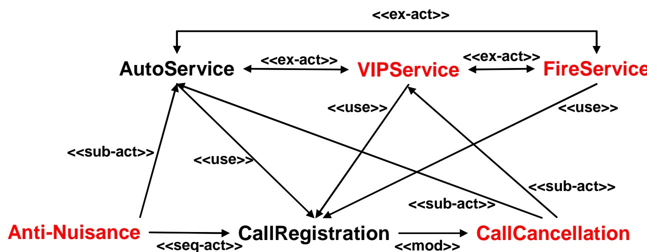

Fig. 4. Operational dependencies among elevator features

Fig. 4 shows the analysis results of dependencies among the elevator features. *AutoService*, *VIPService* and *FireService* are driving service features. As only one of the driving services can be provided at a time, they have *Exclusive-Activation* dependencies with each other. During the operation of each driving service, the elevator registers call requests from passengers. As the driving service features make use of the *CallRegistration* feature for their correct operation, each of them depends on *CallRegistration* in terms of *Usage*. As *CallCancellation* changes the behavior of *CallRegistration* by deleting a registered call request, *CallRegistration* depends on *CallCancellation* in terms of *Modification*. In addition, as *CallCancellation* can be active only during the operation of *AutoService* or *VIPService*, it depends on *AutoService* or *VIPService* in terms of *the Subordinate-Activation* dependency. Finally *Anti-Nuisance* can be active after the completion of *CallRegistration* during the operation of *AutoService*. Thus, it depends on *CallRegistration* and *AutoService* in terms of *Sequential-Activation* and *Subordinate-Activation* dependencies, respectively.

The understanding of various dependencies between features is indispensable for incremental software development. The next section shows how this analysis results can be used to support incremental software development.

4 An Aspect-Oriented Approach to Handling Feature Dependencies

Dependencies between features have significant implications in incremental software development. Whenever additive features are built into a product, components implementing features that depend on additive features may have to be changed. This change mainly occurs because dependencies related to additive features are embedded into components implementing existing features. In order to address this problem, dependencies related to additive features need to be decoupled from components implementing existing features. Detailed guidelines are presented below.

4.1 Encapsulating Feature Dependency

One way of decoupling dependencies related to additive features from existing components is to encapsulate the dependencies into the aspectual components for the features. For example, as shown in Fig. 4, *Anti-Nuisance* depends on *AutoService* and *CallRegistration* in terms of *Subordinate-Activation* and *Sequential-Activation* dependencies. These dependencies can be decoupled from the components implementing *AutoService* and *CallRegistration* and encapsulated into the component implementing *Anti-Nuisance*.

```
1.    aspect Anti-Nuisance {
2.        pointcut anti-nuisanceVP() :
3.         execution(* handleCall()) &&
4.                cflow (execution(* AutoService.service()));
5.
6.        after(CallManager manager, Call call) returning:
7.                anti-nuisanceVP() {
8.                    // perform anti-nuisance functionality
9.         }
10.  }
```

Fig. 5. Aspectual implementation of the Anti-Nuisance feature

Fig. 5 shows an aspectual implementation of the *Anti-Nuisance* feature. The pointcut captures the execution join point of the *handleCall* method, occurring within a given control flow during the execution of the *start* method in the *AutoService* class. After returning from the matched join point, the aspect executes the functionality of *Anti-Nuisance*. The lines 3 and 6 in Fig. 5 are the aspectual implementation of the *Sequential-Activation* dependency, whereas the line 4 is for the *Subordinate-Activation* dependency. Since all dependencies related to the *Anti-Nuisance* feature

are encapsulated into the *Anti-Nuisance* aspect, the addition of the *Anti-Nuisance* feature does not affect the rest of the system. In this way, we can incorporate additive features without modifying existing components.

This approach (i.e., encapsulating dependencies related to an additive feature into the aspectual component for the feature) is applicable when the additive feature has dependency relationships with only core features. In other words, it may have a problem when an additive feature has dependency relationships with other additive features. This problem and its solution are described in the following section.

4.2 Separating Feature Dependency

As illustrated in section 3, the *CallCancellation* feature depends on the *VIPService* feature in terms of feature activation. If the activation dependency is embedded in the *Cancellation* aspect, the aspectual code in Fig. 3 must be further modified like Fig 6 when the *VIPService* feature is added. This implies that the *Cancellation* aspect may have to be changed whenever features that the *CallCancellation* feature depends on are added.

```
1.   pointcut cancellationVP( CallManager manager, Call call) :
2.       call(* exist(Call)) &&
3.       target(manager) && args(call) &&
4.       if (manager.exist(call)) &&
5.       cflow (execution(* AutoService.start()) ||
6.             execution(* VIPService.start()));
```

Fig. 6. Modified aspectual code for the *CallCancellation* feature

In order address this problem, dependencies between features need to be separated from components implementing core functionality of the features. Fig. 7 shows how dependencies related to the *CallCancellation* feature can be separated from the *Cancellation* aspectual component. The *Cancellation* aspect (Lines 1-8) is defined as an abstract aspect, which defines only the cancellation behavior (Line 6) at a specified variation point (Lines 2-3). The concrete aspect (Lines 10-16) that extends the abstract aspect overrides the abstract pointcut (Lines 2-3) by taking dependencies between the *CallCancellation* features and the other features into account. Different sets of dependencies between features can be defined as different concrete aspects by extending the abstract aspect. For example, the *CancellationDependency1* aspect (Lines 10-16) represents the code implementing dependencies related with *CallHandling* and *AutoSerivce* features. The *CancellationDependency2* aspect (Lines 18-25) represents the code implementing dependencies related with *CallHandling*, *AutoSerivce*, and *VIPService* features.

This approach makes it possible to incorporate the *VIPService* feature without modifying the *Cancellation* aspect, as dependencies related to the *CallCancellation* feature is decoupled from the *Cancellation* aspect and defined as a separate aspect.

```
1.    abstract aspect Cancellation {
2.        protected abstract pointcut
3.            cancellationVP( CallManager manager, Call call);

4.        after(CallManager manager, Call call) returning:
5.                    cancellationVP(manager, call) {
6.                    manager.delete(call);
7.         }
8.    }
9.
10.   public aspect CancellationDependency1 extends Cancellation {
11.       protected pointcut CancellationVP():
12.           call(* exist(Call)) &&
13.           target(manager) && args(call) &&
14.           if (manager.exist(call)) &&
15.          cflow (execution(* AutoService.start()));
16.   }
17.
18.   public aspect CancellationDependency2 extends Cancellation {
19.       protected pointcut CancellationVP():
20.           call(* exist(Call)) &&
21.           target(manager) && args(call) &&
22.           if (manager.exist(call)) &&
23.          cflow (execution(* AutoService.start()) ||
24.                   execution(* VIPService.start()));
25.   }
```

Fig. 7. Separating dependencies

5 Conclusion

Feature dependency analysis provides essential design drivers for incremental software development, while AOP provides effective mechanisms for encapsulating crosscutting abstractions into modular units and integrating the units without changing the rest of the system. In this paper, we have identified the problems that may occur in incremental software development using AOP. To address the problems, we have proposed guidelines on how feature dependency analysis and AOP (especially AspectJ) can be combined to support incremental software development. Our experience showed that separating dependencies from aspectual components was critical for improving flexibility and evolution of software. We could easily add more features without changing other parts of software.

Although we have demonstrated how various dependencies between features can be handled using an AOP technique (i.e., AspectJ) to support incremental software development, we do not argue that AspectJ is the best AOP technique. More advance implementation method or techniques may be combined with feature dependency analysis to improve the proposed method.

References

1. AspectJ Team, "AspectJ Project", http://www.eclipse.org/aspectj/.
2. Lee, K. and Kang, K. C.: Feature Dependency Analysis for Product Line Component Design. Lecture Notes in Computer Science, Vol. 3107, Springer-Verlag, Berlin (2004) 69-85
3. Ferber, S., Haag, J., Savolainen, J.: Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. Lecture Notes in Computer Science, Vol. 2379, Springer-Verlag, Berlin (2002) 235-256
4. Fey, D., Fajta, R., Boros, A.: Feature Modeling: A Meta-model to Enhance Usability and Usefulness. Lecture Notes in Computer Science, Vol. 2379, Springer-Verlag, Berlin (2002) 198-216