Chapter 21
Variability and Aspect Orientation

Kwanwoo Lee

What you will learn in this chapter

- *The relationship between variability and aspect orientation*
- *How variability is realized using aspect orientation*

1 Introduction

Variability is an inherent property of software product lines. In software product line engineering, variability must be systematically described and managed throughout all development activities. Variability in a software product line is often analyzed and modeled in terms of features. Optional or alternative features in a feature diagram [1, 2] represent units of variations in requirements. However, realizing a feature may affect several parts of core assets (e.g., architectures or implementation components) instead of being localized.

There are two reasons for this: the first reason is a unit of features does not always correspond to that of components, i.e., the code implementing a particular feature may be scattered across multiple components. Second, features are not independent entities. If dependencies or interactions among features are hard-coded in several components implementing their related features, variations of feature dependency caused by feature variation (i.e., addition or deletion) may cause significant changes to many components. The above reasons make it difficult to realize the variability of a software product line in terms of features.

Aspect-oriented programming (AOP) [3] is a good candidate for handling the crosscutting problem, as it provides effective mechanisms for encapsulating

K. Lee (✉)
Department of Information Systems Engineering, Hansung University, Seongbuk-gu, Seoul, Republic of Korea
e-mail: kwlee@hansung.ac.kr

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_21,

crosscutting concerns into separate modular units called aspects. This chapter describes how aspect orientation can help realizing variability and presents areas of practice that are relevant to the topic with discussion of benefits and possible problems.

2 Relationship Between Variability and Aspect Orientation

Variability identified in terms of features can be classified into two categories: modular features and crosscutting features, depending on the impact on their implementation.

Definition 21.1. Modular feature
A feature is modular if its implementation can be confined to a single modular component.

Definition 21.2. Crosscutting feature
A feature is crosscutting if its implementation spans multiple modular components.

Modular features can be implemented as modular components such as classes in object-oriented programming. Suppose, for example, diesel and gasoline engines are alternative features of an automobile product line. Each of them can be implemented independently from the other. On the other hand, crosscutting features have widespread impacts on multiple modular components. For example, safety policies employed by automobile products can have widespread impact on multiple control components, such as *Engine*, *Brake*, and *Airbag* components.

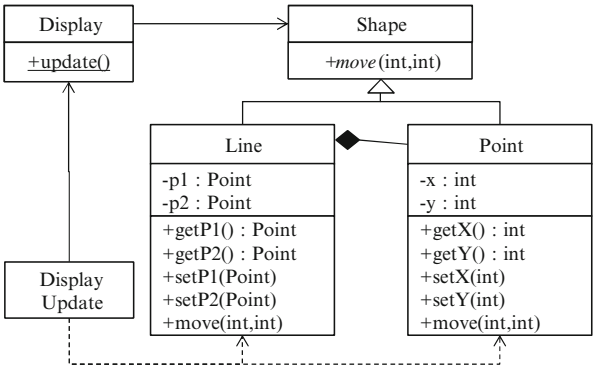
AOP supports separation of crosscutting features, whose implementation results in modification of several modular units (e.g., classes), from features that can well be encapsulated into modular units. This separation of concerns improves adaptability and configurability of product line assets, as the concerns that affect multiple modular units can be encapsulated into separate modular units, called aspects.

Definition 21.3. Aspect
An aspect is a separate modular unit encapsulating any crosscutting concern, which would otherwise be scattered across multiple components.

AOP languages, such as AspectJ [3] which is an aspect-oriented extension to Java, support the encapsulation of crosscutting features into new modular units—the aspects—through new composition mechanisms, such as pointcut advice. The *pointcut* mechanism is used to capture points where crosscutting concerns need to be inserted. A crosscutting concern to be inserted is defined through the *advice* mechanism.

Example 21.1. Aspectual implementation of a crosscutting feature
Suppose for example a simple drawing tool has a crosscutting feature, i.e., *UpdateDisplay*—the *Display* in the figure editor must be updated whenever the

state of each *Shape* instance changes. Implementing this feature in an object-oriented style leads to scattered *Display.update*() calls throughout the *set* and *move* methods of the *Line* and *Point* classes. Using AspectJ, the crosscutting concern can be effectively modularized into a single modular unit. That is, the *DisplayUpdate* aspect modularizes the scattered *Display.update*() calls using the pointcut-advice mechanism of AspectJ. The pointcut *ShapeUpdated* (lines 2 and 3) captures the call to methods of *Shape* subclasses (i.e., the *Line* and *Point* classes), where the method name starts with "set" or is "move." Whereas, the after advice (lines 4–6) inserts *Display.update*() after the join points specified at the pointcut *ShapeUpdated*.




```
1. public aspect DisplayUpdate {
2.     pointcut ShapeUpdated(Shape s):
3.         target(s) && (call(* Shape+.set*(..)) || call(* Shape+.move(..)));
4.     after(Shape s): ShapeUpdated(s) {
5.         Display.update(s);
6.     }
7. }
```

Realizing crosscutting features using AOP makes it easy to trace between features and their implementation units. If features are independent of each other, their variations (i.e., inclusion or exclusion) do not cause problems. However, if they are not, their variation may cause changes to the implementation or side effects in the behavior of other features.

Definition 21.4. Operational feature dependency

Operational feature dependencies are implicitly or explicitly created relationships between features in such a way that the behavior or implementation of one feature affects that of other features.

Operational feature dependencies [4] have significant implications on variability. If the code for dependencies between features is embedded into feature implementation modules, a feature variation will affect the modules implementing

other features. This problem, known as optional feature problem [5], mainly comes from a lack of understanding of operational feature dependencies and scattering dependency-related code across feature implementation modules.

Example 21.2. Operational feature dependency between *ShapeColor* and *UpdateDisplay*

Suppose for example the optional feature *ShapeColor* in the figure editor example extends the *Shape* class with a *color* attribute and corresponding getter and setter methods. However, introducing this feature affects the existing *DisplayUpdate* aspect that implements the *UpdateDisplay* feature to reflect the proper update of a display when a Shape's color changes. Line 9 indicates the code realizing the dependency between *ShapeColor* and *UpdateDisplay*. This implies the *DisplayUpdate* aspect has to be changed according to the selection of the *ShapeColor* feature.

```
1. public aspect ShapeColor {
2.       private Color Shape.color;
3.       public Color Shape.getColor() {return color;}
4.       public void Shape.changeColor(Color c) {color=c;}
5. }
6. public aspect DisplayUpdate {
7.       pointcut ShapeUpdated():
8.           target(s) && (call(* Shape+.set*(..)) || call(* Shape+.move(..)) ||
9.           call(* Shape+.changeColor(..)));
10.    after(Shape s): ShapeUpdated(s) {
11.       Display.update(s);
12.    }
13. }
```

One effective way of handling the variability issue related to operational feature dependency is separating dependency aspects from the implementation of features. AOP can help isolating dependency aspects between feature implementation modules as it provides effective mechanisms for extending a noninvasive way of crosscutting issues.

Example 21.3. Aspectual separation of the operational feature dependency between *ShapeColor* and *UpdateDisplay*

The code snippet below shows how the operational dependency between *ShapeColor* and *UpdateDisplay* can be separated from the *DisplayUpdate* aspect. The *DisplayUpdate* aspect (lines 1–6) defines only the core functionality (line 4) of the *UpdateDisplay* feature, which will be inserted at the join-points specified by the abstract pointcut *ShapeUpdated* (line 2). The *DependencyWithShapeColor* aspect (lines 7–11) defines the operational dependency between *ShapeColor* and *UpdateDisplay* by overriding the abstract pointcut. The *NoDependencyWith-ShapeColor* aspect implements the default dependency between *DispayUpdate*

and *Shape* instances. During the application engineering phase, one of the aspects can be configured according to the selection of the *ShapeColor* feature.

```
1. public abstract aspect DisplayUpdate {
2.       protected abstract pointcut ShapeUpdated(Shape s);
3.       after(Shape s): ShapeUpdated(s) {
4.          Display.update(s);
5.       }
6. }

7. public aspect DependencyWithShapeColor extends DisplayUpdate {
8.       protected pointcut ShapeUpdated(Shape s):
9.          target(s) && (call(* Shape+.set*(..)) || call(* Shape+.move(..)) ||
10.         call(* Shape+.changeColor(..)));
11. }

12. public aspect NoDependencyWithShapeColor extends DisplayUpdate {
13.       protected pointcut ShapeUpdated():
14.          target(s) && (call(* Shape+.set*(..)) || call(* Shape+.move(..)));
15. }
```

With the understanding of operational feature dependencies and AOP mechanisms, variability of a product line can be effectively handled.

3 Recommended Areas of Practice

This section describes two practice areas applying AOP to improve feature modularity and independence.

3.1 Modularization of Crosscutting Features

As described earlier, AOP by nature provides powerful mechanisms for encapsulating crosscutting concerns. With the help of AOP mechanisms, crosscutting features can be effectively modularized into aspectual components.

There have been several attempts to apply AOP in the development of industrial or non-trivial problems. Alves et al. [6] apply AOP in the development of mobile game product lines. They evaluate their approach in the context of an industrial-strength mobile game product line. Kästner et al. [7] refactor the embedded database system Berkeley DB into a software product line and evaluate the suitability of AspectJ for modularizing feature implementations. They report several limitations on the modularization of features when using the AspectJ language,

such as the increasing of coupling between aspects and classes due to the strong dependency of aspect pointcuts and implementation details of the base code. Zhang and Jacobsen conducted aspect-oriented refactoring of CORBA implementations [8]. Their results indicate that they were able to significantly reduce the complexity of the CORBA architecture with negligible performance overhead.

Quality attributes are the crosscutting concerns that have application-wide impact across modular components. Since separating and encapsulating them can help program understanding and improve adaptability, several efforts have been made to modularize quality attributes using AOP. Viega et al. [9] built an aspect-oriented extension to the C programming language to separate security policies from C programs. This approach allows developers to write the core functionality of the application, while a security expert focuses on specifying security properties. Szentiványi and Nadjm-Tehrani [10] proposed an approach to improve performance of fault-tolerant services using AspectJ. In this approach, an application-specific synchronization mechanism is defined as the separate aspects, which are alternatives of a general synchronization mechanism directly supported by the middleware. By using application-specific synchronization aspects, around 40 % of original overhead caused by a general synchronization mechanism could be reduced.

3.2 Separation of Feature Dependencies

Modularizing crosscutting features does not guarantee that feature implementation modules are independent. The optional feature problem may occur when optional features are not independent.

Kästner et al. [5] elaborated the impact of the optional feature problem in two case studies (i.e., Berkely DB and FAME-DBMS) and surveyed different solutions to the problem and their trade-offs. One effective way of handling the problem is to remove code implementing feature dependencies from the modular implementations of related features. The idea is to extract the code responsible for the dependency into a separate module, called derivative module [11]. The derivative module is included in the generation process to restore the original behavior if and only if both original implementation modules are selected.

Lee et al. [12] also addressed the problem by separating feature dependencies from feature implementations using AOP techniques. Specifically, they proposed aspect-oriented implementation patterns for feature dependencies, which are repeatable well-known patterns for the implementation of feature dependencies.

The optional feature problem is closely related to research in the field of feature interactions [13]. Feature interactions can cause unexpected behavior when two optional features are combined. For example, in a home integration system (HIS) product line, the Fire Control feature opens the water main and turns sprinklers on when a fire is detected. If the Flood Control feature, which shuts off the water main to a home in the event of a flood, is added to the HIS with the Fire Control feature,

the Flood Control and Fire Control features may cause an undesirable side effect (e.g., the Flood Control feature disturbs the Fire Control feature by shutting off the water main before the fire is under control). Handling feature interactions may cause significant changes to product line assets if interaction-related code is scattered across many implementation modules. Therefore, interaction related code should be separated from feature implementation for flexible feature composition.

4 Outlook

Variability may have crosscutting concerns. This chapter explained the relationships between variability and crosscutting concerns, and described how variability having crosscutting concerns can be effectively modularized using aspect-orientation mechanisms. Overall, aspect orientation becomes a valuable instrument in modularizing crosscutting variability.

However, AOP has several limitations, some of which include pointcut fragility and code readability. As pointed out in [7], the pointcut language of AspectJ has language limitations, such as the statement extension problem and pointcut fragility, which constrain the identification and definition of interaction points between class modules and aspect modules. These limitations can be alleviated by making the interaction points explicit in an abstract way. For example, crosscutting programming interfaces [14] can be used to clarify the separation of base and extensions.

In addition, aspects may be too small modular units. A fine-grained fragmentation of product line assets increases the complexity of managing variability. Nevertheless, substantial research addressing the above-mentioned limitations still is necessary before this relatively new paradigm can be applied on a broad scale in various industrial domains.

References

1. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, SEI, Carnegie Mellon University, Pittsburgh, PA (1990)
2. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: Proc. SPLC 2004, pp. 266–283 (2004)
3. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C. Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Proc. ECOOP 1997, pp. 220–242 (1997)
4. Lee, K., Kang, K.C.: Feature dependency analysis for product line component design. In: Proc. ICSR 2004, Madrid, Spain, pp. 69–85. Springer (2004)
5. Kästner, C., Apel, S., ur Rahman, S., Rosenmüller, M., Batory, D., Saake, G.: On the impact of the optional feature problem: analysis and case studies. In: Proc. SPLC 2009, pp. 181–190 (2009)

6. Alves, V., Matos, Jr., P., Cole, L., Borba, P., Ramalho, G.: Extracting and evolving mobile games product lines. In: Proc. SPLC 2005, pp. 70–81 (2005)
7. Kästner, C., Apel, S., Batory, D.: A case study implementing features using AspectJ. In: Proc. SPLC 2007, pp. 223–232 (2007)
8. Zhang, C., Jacobsen, H.-A.: Refactoring middleware with aspects. IEEE Trans. Parallel Distr. Syst. **14**(11), 1–16 (2003)
9. Viega, J., Bloch, J.T., Chandra, P.: Applying aspect-oriented programming to security. Cutter IT Journal **14**(2):31–39, February 2001
10. Szentiványi, D., Nadjm-Tehrani, S.: Aspects for improvement of performance in fault-tolerant software. In: Proc. PRDC 2004, pp. 283–291 (2004)
11. Liu, J., Batory, D., Lengauer, C.: Feature-oriented refactoring of legacy applications. In: Proc. ICSE 2006, pp. 112–121 (2006)
12. Lee, K., Botterweck, G., Thiel, S.: Aspectual separation of feature dependencies for flexible feature composition. In: Proc. COMPSAC 2009, pp. 45–52 (2009)
13. Calder, M., Kolberg, M., Magill, E.H., Reiff-Marganiec, S.: Feature interaction – a critical review and considered forecast. Comput. Netw. **41**(1), 115–141 (2003)
14. William, G., Shonie, M., Sullivan, K., Song, Y., Tewari, N., Cai, Y., Rajan, H.: Modular software design with crosscutting interfaces. IEEE Softw. **23**(1), 51–60 (2006)