Aspect-Oriented Patterns for the Realization of Flexible Feature Binding

Kwanwoo Lee
Department of Information Systems Engineering
Hansung University
Seoul, Korea
kwlee@hansung.ac.kr

***Abstract*—Feature selection is the process of determining features that should be included in a product to satisfy the requirements for the various stakeholders. Feature binding time refers to the time at which variable features are selected for a product and their implementations are bound into the product. A feature may have different binding times for different products. In this paper, we present an aspect-oriented approach to supporting flexible feature binding time.**

***Index Terms*—component; formatting; style; styling;**

I. Introduction

In software product line engineering, a product is derived by selecting some of product line features that satisfies the product requirements. Feature selection is the process of determining features that should be included in a product to satisfy the requirements for the various stakeholders.

The time at which features are selected for a product may vary depending on marketing strategies. For instance, one marketing strategy may be to deliver products to customers by prepackaging product specific features, as customer needs in this market rarely change. In this case, product specific features may be selected for a product during product build time. On the other hand, another marketing strategy may be to allow customers to start with a product with core features and then grow to a bigger one by adding new features at load time or run time. In this case, product specific features may be selected for products at product load time or run time.

Feature binding time refers to the time at which variable features are selected for a product and their implementations are bound into the product. Feature binding time has significant influences on the way a feature is implemented. There are many variability mechanisms for realizing feature binding times. Some of those include conditional compilation, macro processing, virtual dispatch tables, reflection, dynamic class loading, etc.

These mechanisms, however, are strongly tied to a particular choice of binding time [1]. The problem may occur when a feature may have different binding times for different products. That is, the variability of feature binding time affects the way a feature is implemented. To support multiple feature binding times effectively, code for feature binding times needs to be separated from feature implementations.

Aspect-oriented programming (AOP) provides effective mechanisms for separating crosscutting concerns from modular components. Since the code for multiple feature binding times may affect multiple feature implementation components, this paper uses AOP mechanisms to achieve flexible feature binding time. That is. The approach makes it possible to choose among compile-time, load-time and run-time binding for selected features.

For better understanding of this paper, the next section presents the concept of feature binding. Based on this understanding, aspect-oriented patterns for supporting flexible feature binding time are presented in section 3. Section 4 discusses areas requiring further research and concludes this paper.

II. Feature Binding

A feature has to be bound into a product to provide its capability to users. As shown in Fig. 1, an unbound feature must be first included into a product to provide its capability. However, it is considered that a feature is not bound into a product, if it is not available to users although included in the product. Therefore, feature binding means that a feature is included into a product and become available to users. It is important to note that available features that are bound into a product can provide their capabilities to users only when they are active.

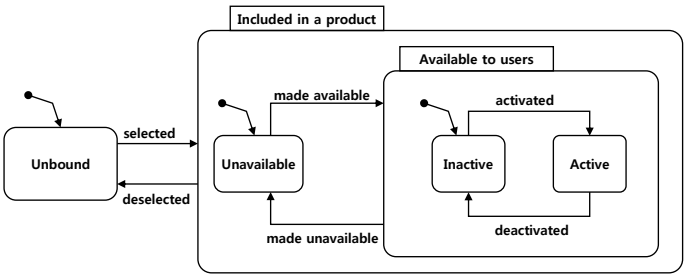

Fig. 1. Feature Binding Context.

For feature binding to occur, the inclusion and availability of a feature may be decided simultaneously at compile time, load time, or run time. Or, the inclusion of a feature is decided at compile time, but its availability may be postponed until load time or run time.

Feature binding at compile time: In this case, the code related to the selected features is included into a product and becomes available to customers at compile time, while the code for deselected features is excluded from the product. This results in different packaging of the product.

Feature binding at load time: There are two cases for load time feature binding to occur. The first case is that features are included into a product at compile time and become available at load time. The second case is that the inclusion and availability of features are decided at load time. The main difference between the compile feature binding and the load time feature binding is that the software derived from the compile time feature binding must be compiled, whereas that from the load time feature binding needs not.

Feature binding at run time: Feature binding in this class is similar to the load time feature binding. That is, features are included into a product at compile time and become available at run time or included at product load time and available at run time. Alternatively, both the inclusion and availability of features are decided at run time.

Depending on the inclusion and availability decisions of a feature, it may have to be implemented in different ways. In the next section, we present an aspect-oriented approach to supporting flexible feature binding times.

III. Aspect-Oriented Design Patterns

The current AspectJ weaver provides explicit support for compile-time and load time weaving. This implies that if we implement variable features with aspects, we do not need to change the aspects to support for either compile-time or load-time feature binding. However, when we want to implement a load-time binding feature that has to be included at compile-time and becomes available at load-time, we have to change the feature implementation to support the required feature binding decisions. Moreover, if we consider run-time feature binding, we may have to change the feature implementation as well. This implies a feature with multiple binding times may have to be implemented differently depending on which binding time decisions are decided for a product.

In this section, we present aspect-oriented design patterns for supporting flexible feature binding times.

A. Variable Inclusion Decisions

For feature binding to occur, the code implementing a feature must be included in a product and integrated with the other code for the product. Since the time at which a feature is included in a product may vary for different products in a product line, the variable inclusion times may require variable feature implementations.

To support flexible binding times effectively, the code implementing the decision on feature inclusion needs to be separated from the code implementing the core functionality of a feature. As shown in Fig. 2, `CompileTimeBinding` is an aspectual implementation for integrating the module (`VFModule`) implementing a variable feature with the module (`CFModule`) in the scope of a product at compile time. The pointcut `variationPoint` defines the join points in `CFModule` at which `VFModule` is bound into a product. The advice body in `CompileTimeBinding` defines actual binding between `CFModule` and `VFModule`. If `VFModule` and `CompileTimeBinding` are given to a AspectJ compiler at compile time, the compiler produce a weaved product.

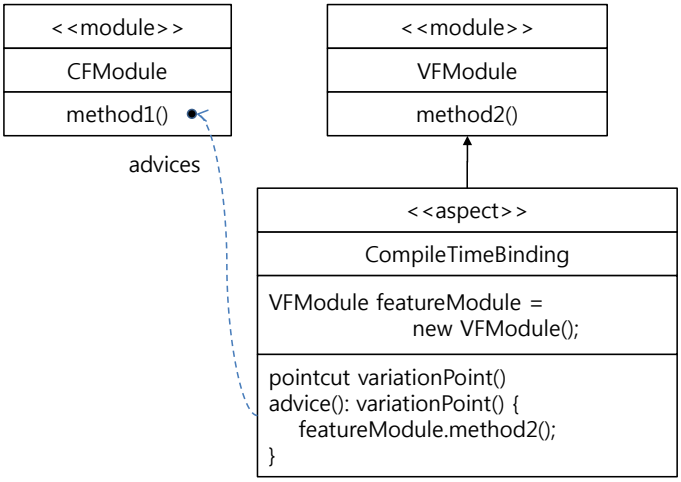

Fig. 2. Compile-Time Inclusion Pattern.

Note that AspectJ provides explicit support for load time weaving. Therefore, we can integrate `VFModule` and `CompileTimeBinding` with an existing product at load time, simply by including them before program execution.

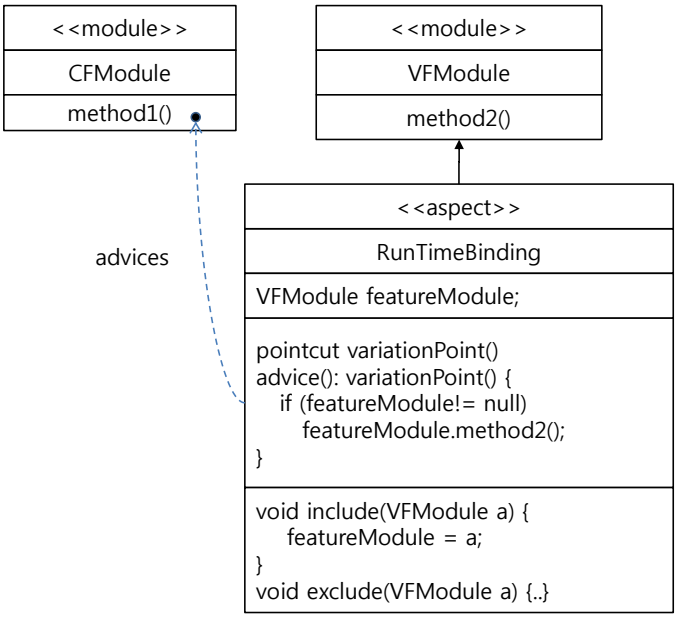

Fig. 3. Run-Time Inclusion Pattern.

Although AspectJ does not provide support for run time weaving, we can implement the run time feature binding which includes the implementation of a variable feature in a product

at run-time. As shown in Fig. 3, `RunTimeBinding` allow `VFModule` implementing a variable feature to be included in a product scope and integrated with `CFModule`, which is an implementation module in the product scope.

`RunTimeBinding` is similar to `CompileTime-Binding` in that it specifies the join points used to integrate `CFModule` and `VFModule` using pointcut definitions. However, their integration is deferred until the actual instance of `VFModule` is included in the product at run-time. A external configurator has the responsibility for including the instance using the `include` method based on the user's decision at run-time.

B. Deferring Availability Decisions

Although AspectJ weaver does not provide explicit support for run-time weaving, we can support runtime feature binding using AOP, in case inclusion is determined at compile-time but availability is decided at load-time or run-time. That is, availability of included aspects is decided by enabling or disabling advices in the aspects.

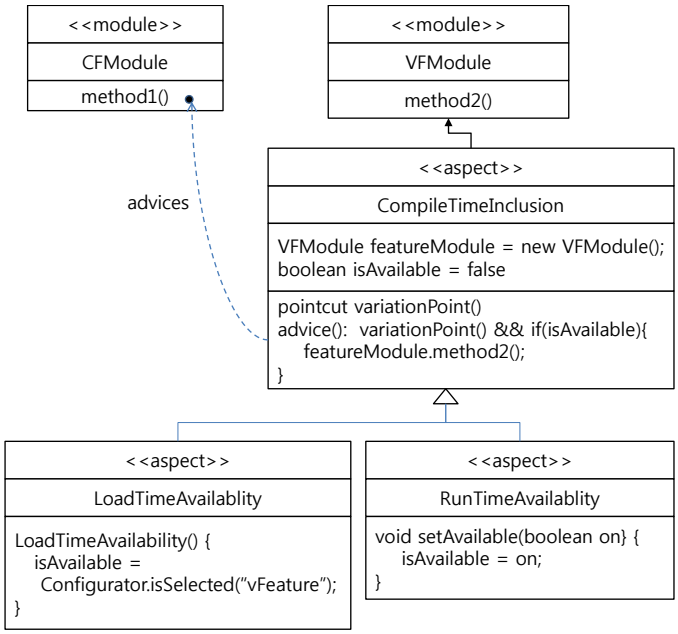

Fig. 4. Variable Availability Pattern.

Fig. 4 shows how to support the load-time or run-time binding of a variable feature, which is included at compile time but becomes available during load-time or run-time. `CompileTimeInclusionNotAvailable` is similar to `CompileTimeBinding` shown in Fig. 2 except that the boolean type variable `isAvailable` is used to allow the availability decision to be decided in its child aspects (`LoadTimeAvailability` and `RunTimeAvailability`).

`LoadTimeAvailability` determines the value of `isAvailable` when the aspect is created by consulting with an external configuration, which has responsibility for configuring a product after compilation. On the other hand, `RunTimeAvailability` sets the variable `isAvailable` to true using the method `setAvailable`. But the decision is made by an external configurator at run-time.

With these patterns, we can clearly separate binding time decisions from the code implementing the core functionality of a feature. This enables us to select different choices among multiple feature binding implementations when a feature has multiple feature binding times.

IV. Related Work

The concept of feature binding time was first introduced by Kang et al. [3]. Gurp et al. [5] elaborated it more precisely and provided a classification of many variability realization techniques. A broad range of mechanisms exist to implement different binding times, including the use of compiler directives, dynamic linking and loading, load tables, reflection, plug-ins, configuration files, etc. However these solutions are limited in that each supports only a specific binding time. They cannot be used effectiely in a situation where the binding time of a feature may vary depending on different product requirements.

Dolstra et al. [2] introduced the notion on the variability of feature binding time as *timeline variability*. However, they do not provide concrete mechanisms for realizing flexible feature binding time. They only suggested some future directions for the solutions.

There have been several attempts to realize flexible feature binding time. Hoek [6] proposed architecture-based approach to support any-time variability. The Koala component model [7] allows connection between components to be established either at compile time or at run time through *a switch*. Depending on the setting of a switch, the Koala compiler generates C code for connecting components either at compile time or at run time. Both of these approaches specify product line variabilities at design time, but resove them at any time thereafter. On the other hand, the approach presented in this paper uses aspect-oriented design patterns using AOP.

Similar to our approach, Edicts [1] uses AOP for flexible feature binding. However, it only supports run time feature biding which includes a feature at compile time and makes it available at run time. But our approach can add more flexibility of feature binding time from different choices of inclusion and avaiability decisions.

V. Research Issues and Conclusions

In this paper, aspect-oriented patterns are introduced to support flexible feature binding times. There are many issues that have to be addressed before this approach becomes useful. Some of these issues are summarized below:

- *Method*: We need methods for analyzing feature binding time, developing a software product line applying the patterns presented above, deriving a product based on a feature configuration, which can be determined at either compile-time, load-time, or run-time.

- *Feature model extension*: Although there have been many attempts to extend the original feature model [3], there has been no attempt to model the variability of feature binding time. Since the inclusion and availability decisions for feature binding are affected by resources available during run time or development environments such as programming languages or operating environments, when we model variabilities of feature binding time, we may take into account constraints or dependencies from various sources (e.g., available resources). Also we may have to consider finer classification of feature binding times.
- *Implementation mechanism*: In this paper, we illustrated the patterns using AspectJ. However, more advanced mechanisms such as Prose [4], which supports run-time weaving, can be used to support flexible feature binding time. Which one among current available technologies or mechanisms can be best utilized for achieving this goal? We need to analyze pros and cons for each technology or mechanism.

In this section, we have examined some of research issues that have to be addressed. Although we are at an early stage of research, most of research topics discussed above are being addressed.

ACKNOWLEDGMENT

This work was supported by the Korea SW Industry Promotion Agency (KIPA)

REFERENCES

[1] V. Chakravarthy, J. Regehr, E. Eide, *Edits: Implementing Features with Flexible Binding Times*, AOSD'08, 2008.

[2] E. Dolstra, G. Florijin, E. Visser, *Timeline Variability: The Variability of Binding Time of Variation Points*, Proceedings of the Workshop on Software Variability Management, 2003, pp. 119-122.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, *Feature-Oriented Domain Analysis (FODA) feasibility study*, Software Engineering Institute Technical Report CMU/SEI-90TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[4] A. Popovici, T. Gross, G Alonso, *Dynamic Weaving for Aspect-Oriented Programming*, Proceedings of the 1st international conference on Aspect-oriented software development, Enschede, The Netherlands, 2002, pp. 141-147

[5] M. Svahnberg, J. van Gurp, and J. Bosch. *A taxonomy of variability realization techniques*, Software-Practice & Experience, 35(8), pp. 705-754, 2005.

[6] A. van der Hoek, *Design-Time Product Line Architecture for Any-Time Variability*, Science of Computer Programming, Vol. 53, Issue 3, December 2004, pp. 285-304

[7] R. van Ommering, *Building Product Populations with Software Components*, ICSE'02, May 19-25, 2002, pp.255-264