Feature Dependency Analysis for Product Line Component Design*

Kwanwoo Lee[1] and Kyo C. Kang[2]

[1] Division of Information Engineering, Hansung University,
389 Samsun-dong 2-ga, Sungbuk-gu, Seoul, 136-792, Korea
kwlee@hansung.ac.kr

[2] Department of Computer Science and Engineering,
Pohang University of Science and Technology,
San 31 Hyoja-Dong, Pohang, 790-784, Korea
kck@postech.ac.kr

Abstract. Analyzing commonalities and variabilities among products of a product line is an essential activity for product line asset development. A feature-oriented approach to commonality and variability analysis (called feature modeling) has been used extensively for product line engineering. Feature modeling mainly focuses on identifying commonalities and variabilities among products of a product line and organizing them in terms of structural relationships (e.g., aggregation and generalization) and configuration dependencies (e.g., required and excluded). Although the structural relationships and configuration dependencies are essential inputs to product line asset development, they are not sufficient to develop reusable and adaptable product line assets. Other types of dependencies among features also have significant influences on the design of product line assets. In this paper, we extend the feature modeling to analyze feature dependencies that are useful in the design of reusable and adaptable product line components, and present design guidelines based on the extended model. An elevator control software example is used to illustrate the concept of the proposed method.

1 Introduction

Product line software engineering (PLSE) [1] is an emerging software engineering paradigm, which guides organizations toward the development of products through the strategic reuse of product line assets (e.g., architectures and software components). In order to develop product line assets, commonalities and variabilities among products must be analyzed thoroughly, as commonalities can be exploited to design reusable assets and variabilities can be used for the design of adaptable and configurable assets.

* This Research was financially supported by Hansung University in the year of 2003.

J. Bosch and C. Krueger (Eds.): ICSR 2004, LNCS 3107, pp. 69–85, 2004.
© Springer-Verlag Berlin Heidelberg 2004

A feature-oriented approach to commonality and variability analysis has been used extensively in product line engineering [2], [5], [6], [10], [16], [17]. This is mainly due to the fact that the feature-oriented analysis is an effective way of identifying and modeling variabilities among products of a product line. In addition, the analysis result, a feature model, plays a central role in configuring multiple products of a product line.

In the feature-oriented approach, variable features are considered units of variation (i.e., addition or deletion) in requirements. A variable feature, if not properly designed, may affect a large part of product line assets. If features are independent each other, each of them can be developed in isolation, and effects of a variation can be localized to the corresponding component. However, as features usually are not independent, a feature variation may cause changes to many components implementing other features. For example, in an elevator product line, the *Automatic Door Close* feature, which automatically closes doors if the predefined time elapses after doors have been open, can be active during the operation of *Passenger Service* feature. However, the *Automatic Door Close* feature must not be active while the *VIP Service*[1] feature is active, as the *VIP Service* feature allows doors to be closed only when the door close button is pressed. That is, activation or deactivation of *Automatic Door Close* depends on the activation state of *Passenger Service* or *VIP Service*. If the activation dependency is embedded with the component implementing the *Automatic Door Close* feature (i.e., the component checks activation of *Passenger Service* or *VIP Service* before executing its functionality), the inclusion or exclusion of *VIP Service* causes changes to the component for *Automatic Door Close*. Moreover, if a newly introduced feature *Fire Fighter Service*[2] does not also allow the *Automatic Door Close* feature to be active during its operation, the inclusion of the *Fire Fighter Service* feature also causes a change to the component for *Automatic Door Close*.

Feature dependencies have significant implications in product line asset development. In the product line context, products of a product line must be derivable from product line assets. Product line assets must be designed so that inclusion or exclusion of variable features causes little changes to components implementing other features. In order to achieve this goal, various dependencies that variable features have with other features must be analyzed thoroughly before designing product line assets. With this understanding, product line assets must be designed so that variations of a feature can be localized to one or a few components.

---

[1] *VIP Service* indicates a driving service an elevator provides for VIPs. When it is activated by the landing key switch, a car moves to a predefined floor and waits for a car call after opening doors. If a car call is entered and the door close button is pressed, the car will serve the car call. When no car call is entered within the preset time, the car will close its doors and resume *Passenger Service*.

[2] *Fire Fighter Service* may be activated using a key switch in a car and places an elevator under the control of a fire fighter. The behavior of this feature must be in compliance with the local laws. In this paper, we assume that the car starts traveling while a car call button is being pressed and stops at the requested floor with doors closed. The doors can be opened only while the door open button is being pressed. If the button is released, the doors are closed immediately.

Although understanding feature dependencies is critical in product line asset development, they have not been analyzed and documented explicitly. In this paper, we extend the feature analysis method to include a feature dependency analysis as well as a commonality and variability analysis. This paper introduces six types of feature dependencies and discusses what engineering implications each class of feature dependencies has in the design of reusable and adaptable product line assets. Based on this understanding, we present component design guidelines. An elevator control software (ECS) product line example is used throughout this paper to illustrate the concepts and the applicability of the proposed method.

An overview of the feature modeling in [2], [9], [14] is given in the following section.

2 Feature Modeling Overview

Feature modeling is the activity of identifying commonalities and variabilities of the products of a product line in terms of features and organizing them into a feature model. The structural aspect of a feature model is organized using two types of relationships: aggregation and generalization relationships. The *aggregation* relationship is used if a feature can be decomposed into a set of constituent features or a collection of features can be composed into an aggregate feature. In cases where a feature can be specialized into more specific ones with additional information or a set of features can be abstracted into a more general one, they are organized using the *generalization* relationship.

Based on this structure, commonalities among all products of a product line are modeled as common features, while variabilities among products are modeled as variable features, from which product specific features can be selected for a particular product. Variable features largely fall into three categories: alternative, OR, and optional features. Alternative features indicate a set of features, from which only one must be present in a product; OR features represent a set of features, from which at least one must be present in a product; optional features mean features that may or may not be present in a product. In the feature model, selection of variable features is further constrained by "required" or "excluded" configuration dependencies. The "*required configuration dependency*" between two features means that when one of them is selected for a product, the other must also be present in the same product. An "*excluded configuration dependency*" between two features means that they cannot be present in the same product.

The commonality and variability information manifested in the feature model serves as a basis for developing product line assets. However, the commonality and variability information is not sufficient to develop reusable and adaptable product line assets. The next section presents a feature dependency analysis, which is indispensable for the development of reusable and adaptable product line assets.

3 Feature Dependency Analysis

Feature modeling in [2], [9], [14] mainly focuses on the structural relationships and the configuration dependencies between features. Although structural relationships and configuration dependencies are essential inputs to the development and configuration of product line assets, operational dependencies among features also have significant implications in the development of product line assets. By operational dependency, we mean implicitly or explicitly created relationships 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. Thus, this section introduces operational dependencies that have significant influences on product line asset development. Each of these is described below.

Usage dependency: A feature may depend on other features for its correct functioning or implementation. For example, in the elevator control software, the *Direction Control* feature depends on the *Position Control* feature, as it requires current position data from the *Position Control* feature to compute the next movement direction of an elevator. Moreover, the *Position Control* feature depends on the *Position Sensor* feature, as it requires sensor inputs from the *Position Sensor* feature to calculate the current position of an elevator. If one feature (a client) requires other feature (a supplier) for its correct functioning or implementation, we define that the first feature depends on the second feature in terms of *Usage* dependency.

Modification dependency: The behavior of a feature (a modifyee) may be modified by other feature (a modifier), while it is in activation. For example, in the elevator control software, the *Registered Floor Stop* feature determines the landing floors of an elevator. The *Load Weighing Bypass* feature, however, changes the behavior of the *Registered Floor Stop* feature by ignoring hall calls when a car is loaded to a predetermined level of capacity. *Modification* dependency between two features means that the behavior of a modifyee may be modified by a modifier, while it is in activation. If the modifier is not active, it does not affect the behavior of its related modifyee features.

A feature must be active before it can provide its functionality to users. An activation of a feature may depend on that of other feature. Activation dependency can be classified into four categories: *Exclusive-Activation*, *Subordinate-Activation*, *Concurrent-Activation*, and *Sequential-Activation*, each of which is described below.

Exclusive-Activation dependency: Some features must not be active at the same time. For example, in the elevator control software, the *Passenger Service* feature and the *Fire Fighter Service* feature must not be active simultaneously, as only one of them can be provided to users at a time. *Exclusive-Activation* dependency between two features means that they must not be active at the same time.

Subordinate-Activation dependency: There may be a feature that can be active only while other feature is active. For example, in the elevator control software, the *Passenger Service* feature consists of several operation features (e.g., *Call Handling*[3],

[3] Call Handling is an operation for registering or canceling passenger's call requests.

Door Control[4], and *Run Control*[5]). These features can be active while the *Passenger Service* feature is active. *Subordinate-Activation* dependency between two features means that one feature (a subordinator) can be active while the other feature (a superior) is active, but must not be active while its superior is inactive. Note that a subordinator may not be active during the operation of its superior. Also, subordinators may be in *Subordinate-Activation* dependency with more than one superior. For example, *Call Handling*, *Door Control*, and *Run Control* can also be active during the operation of *Fire Fighter Service* as well as *Passenger Service*. In case subordinators depend on activation of more than one superior, they can be active while at least one superior is active.

Subordinators may further depend on each other in terms of concurrency or sequence.

Concurrent-Activation dependency: Some subordinators of a superior may have to be active concurrently with each other while the superior is active. For example, *Call Handling* and *Run Control* must be active concurrently while *Passenger Service* is active, as an elevator must be able to register call requests from passengers and control the run of the elevator concurrently during the operation of *Passenger Service*. Therefore, *Concurrent-Activation* dependency defined for subordinators of a superior means that the subordinators must be active concurrently while the superior is active.

Sequential-Activation dependency: Some subordinators of a superior may have to be active in sequence. For example, *Call Handling* and *Run Control* must be active in sequence while *Fire Fighter Service* is active, as an elevator in *Fire Fighter Service* can register a call request only before its movement and start traveling immediately after registering a call request. *Sequential-Activation* dependency between two subordinators of a superior means that one subordinator must be activated immediately after the completion of the other during the operation of the superior.

Usage dependencies among features provide important information that is useful for designing reusable product line assets. If multiple *Usage* clients use the same *Usage* supplier, the *Usage* supplier indicates commonality among *Usage* clients. Although the feature model captures commonalities among products (i.e., common features), *Usage* dependency analysis helps asset designers identify additional commonality (i.e., commonalities among features). This commonality information (i.e., commonality among products and commonality among features) is an important input to the development of reusable components.

In addition, *Usage* and *Modification* dependencies among features have significant implications in the design of reusable and adaptable product line assets. Suppose a feature (*f1*) requires one of alternative features (*f2* and *f3*) for its correct implementation. If components implementing *f1* directly use components implementing *f2* or *f3*, components for *f1* are changed whenever one of *f2* and *f3* is selected for a product. This implies that presence or absence of *Usage* suppliers may affect many components implementing their dependent *Usage* clients. To improve reusability and adaptability of product line assets, variations of *Usage* suppliers should be hidden from

[4] Door Control is a control operation to open and close doors.

[5] Run Control is a control operation for acceleration and deceleration of the elevator.

components implementing their *Usage* clients. Similar to features in *Usage* dependency, as presence or absence of a modifier affects the behavior of its dependent modifyee, components implementing modifyee features must be designed so that they can be easily adapted or extended without significant changes.

Activation dependencies also have influences on reusability and adaptability of product line assets. As illustrated above, the operation features (i.e., *Call Handling* and *Run Control*) may be active either concurrently or sequentially depending on the service mode (i.e., *Passenger Service* or *Fire Fighter Service*). As *Fire Fighter Service* is a variable feature, its presence causes changes to activation dependencies among the operation features. If these activation dependencies are hard-coded in the components implementing the operational features, variations of feature dependency caused by feature variation (i.e., addition or deletion) may cause significant changes to many components. To address this problem, activation dependencies among feature must be separated from components implementing the features.

With this understanding of engineering implications feature dependencies have in the design of reusable and adaptable product line assets, the next section presents detailed guidelines for designing product line asset components.

4 Product Line Component Design

This section describes component design guidelines on how feature-oriented analysis results (i.e., feature commonalities, feature variabilities, and feature dependencies) described in the previous sections can be used to develop reusable and adaptable product line components.

Separating commonality from variability: Commonality in a product line can be looked at from two perspectives: commonalities among products and commonalities among features. If features are common across all products in a product line, they represent commonalities among products. If features are used by more than one feature, they indicate commonalities among features. In order to improve the reusability of product line assets, components implementing commonalities must be decoupled from variabilities so that they can be commonly reused ‘as-is’ for production of multiple products.

Hiding variable features from their Usage client features: As discussed earlier, variation of a *Usage* supplier may affect many components implementing its *Usage* clients. In order to localize effects of a variation, the presence or absence of variable features must be hidden from components implementing their *Usage* client features. As variable features are classified into alternative, OR, and optional ones, detailed component design guidelines for each type of variable features are described below.

In order to hide alternative features from *Usage* client features, generic aspects among alternative features must be modeled as an interface component, while specific aspects must be encapsulated in concrete components. As shown in Fig. 1, the factory component instantiates and returns one instance of each concrete component to the client component through the getInstance method, and then the client component can

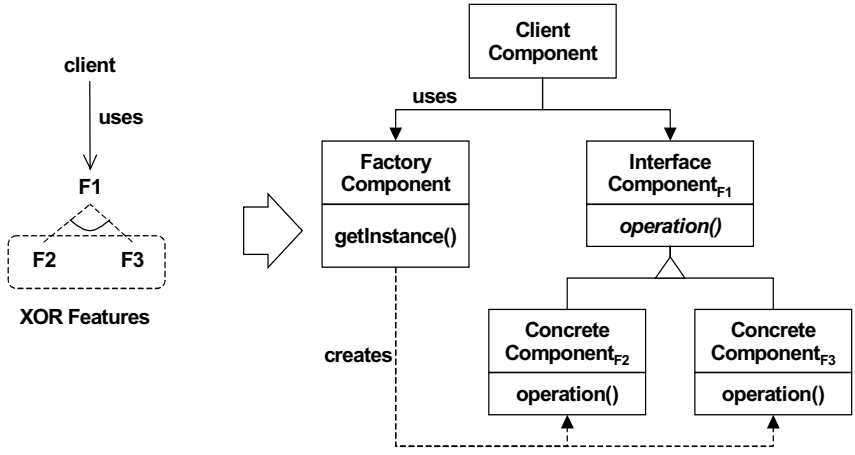

Fig. 1. Hiding XOR Variation

access the concrete component through the abstract interfaces of the interface component. This approach allows run time binding of alternative features into a product without affecting other parts of the product. For build time binding to occur, the interface component can be designed as a parameterized component, which takes a concrete component as an actual parameter.

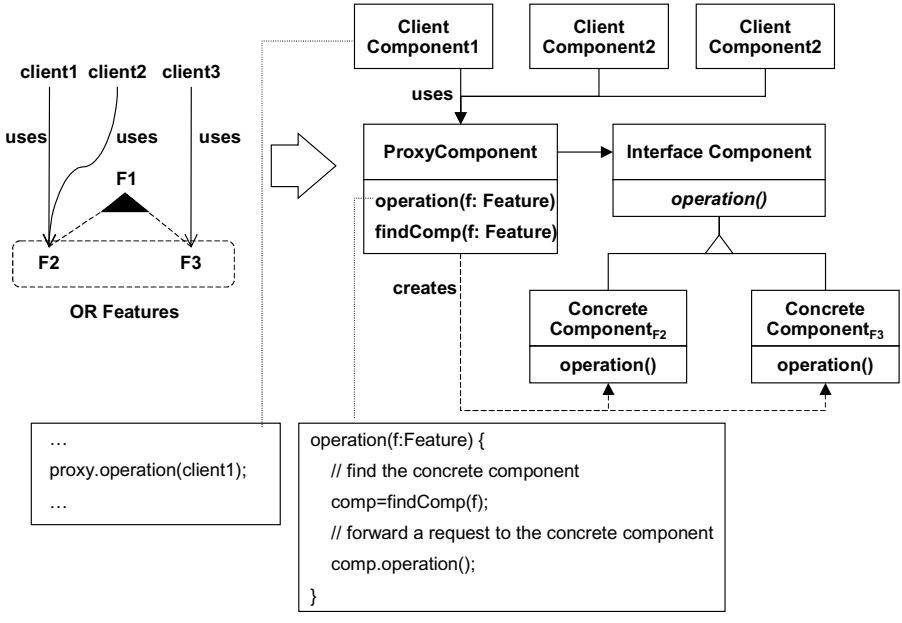

Fig. 2. Hiding OR variation

OR features can be hidden from *Usage* client features by introducing the *ProxyComponent*, which encapsulates the *Usage* dependencies to OR features and forwards requests from components implementing *Usage* client features to corresponding concrete components implementing OR features. For example, as shown in Fig. 2, *ClientComponent1* uses the *operation* method of the *ProxyComponent* by giving its associated feature (i.e., *client1*) as a parameter. The operation method then finds the concrete component implementing the *Usage* dependent feature related to the *client1* feature through the *findComp* method, and forwards the request to the concrete component. As client components do not know which concrete components implementing OR features will respond to their requests, any feature variation does not affect the client components.

In case the component implementing an optional feature is directly used by the components implementing its *Usage* client features, the presence or absence of the former component cannot be fully hidden from its client components. In order to hide optional variation from its clients, the presence or absence of an optional feature must be decoupled from the components implementing its clients. As shown in Fig. 3, the *ClientComponent* makes use of the *ProxyComponent*, which forwards the request from the *ClientComponent* to the $Component_{F1}$, if the $Component_{F1}$ exists. If the $Component_{F1}$ does not exist, the *ProxyComponent* does nothing.

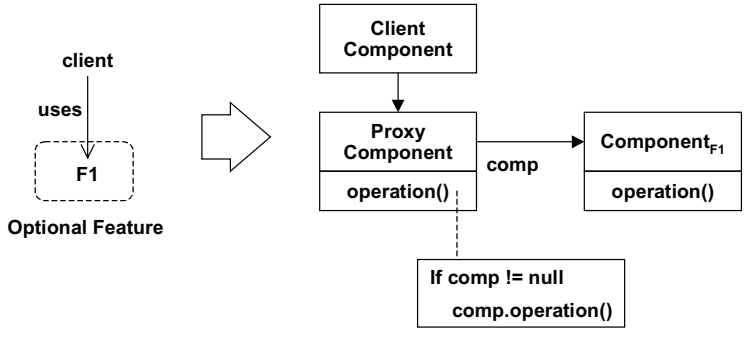

Fig. 3. Hiding Optional variation

Although this approach can confine any feature variation into a single component, this may lead to inefficient systems, as variable features are accessed through abstract interfaces. Moreover, there may be a situation where it is very difficult to encapsulate variable features in a separate component due to other kinds of feature dependencies (e.g., modification and activation dependencies).

Separating Modification dependency from components implementing modifyee features: The presence or absence of modifier features affects components implementing their related modifyee features. In order to make components for modifyee features not changed even with presence of modifier features, modification dependency must be decoupled from components for modifyee features. Inheritance based methods (e.g., class inheritance or mixin [15] can be used to separate modification dependency from components implementing modifyee features. As shown in Fig. 4, the operation of the component implementing the *Modifier* feature extends or over-

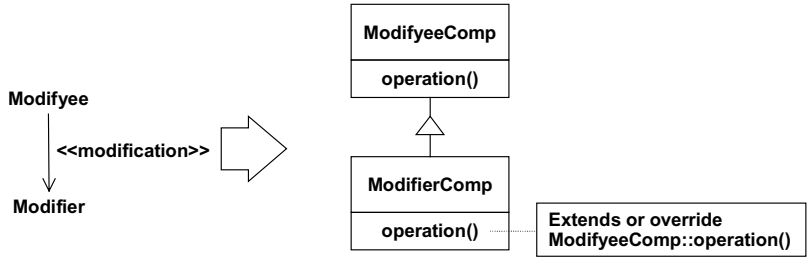

Fig. 4. Separating Modification dependency

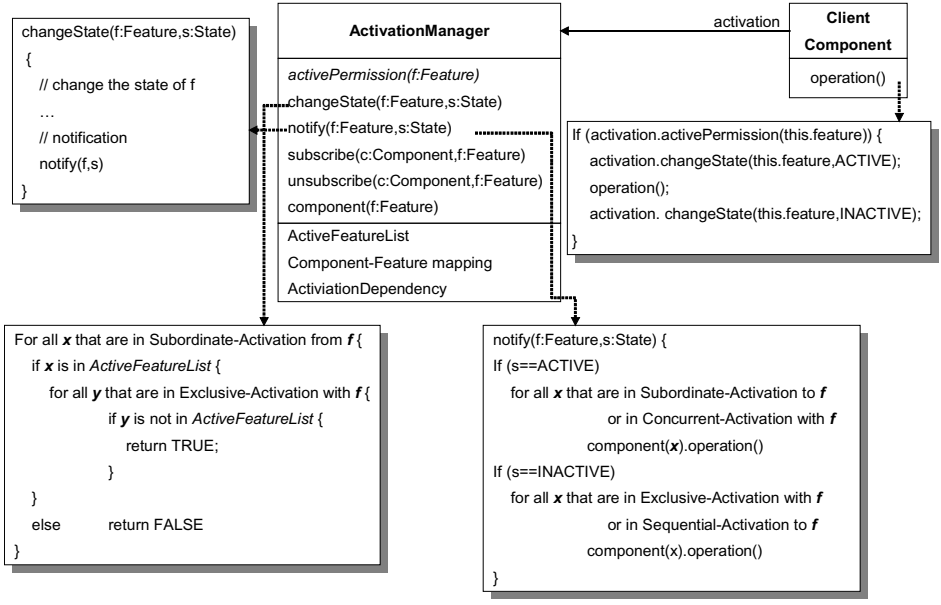

Fig. 5. Separating Activation dependency

rides the component for the *Modifyee* through the inheritance mechanism. An alternative approach could be to implement the *Modification* dependency using an "aspect" [11], which prescribes how components implementing *Modifyee* and *Modifier* features can be put together to form a complete behavior.

Separating Activation dependency from components: Activation dependencies between features must also be separated from components implementing related features, so that presence or absence of a feature will not affect components implementing other features. In order to separate activation dependencies from components, components must be specified in a way that they can execute their functionality only when they have permission to execute the functionality. As shown in Fig. 5, the client component determines the execution of its operation by asking a permission from the *ActivationManager* component. Once the client component has a permission to execute the operation, it reports the activation and deactivation status to *ActivationMan-*

ager before and after its execution so that other components that depend on the activation or deactivation of the feature are notified of the feature state. For instance, when a feature *f* is active, the components implementing the features that are dependent on *f* in terms of *Subordinate-Activation* or *Concurrent-Activation* are notified for execution. On the other hand, when a feature *f* is inactive, the components for the features that are dependent on *f* in terms of *Exclusive-Activation* or *Sequential-Activation* are notified for execution. As this approach makes components independent of each other by separating activation dependencies from components, any feature variation does not affect other components, but only affects the activation manager component.

The guidelines presented above can minimize changes to product line asset components, when variable features are included or excluded for production of a particular product. In the next section, we illustrate how feature dependencies can be analyzed and used to develop product line components using an elevator control software product line.

5 Feature-Oriented Engineering of Elevator Control Software

The example selected in this paper is an elevator control software (ECS) product line. For the purpose of illustration, we simplified the ECS product line by eliminating various indication features (e.g., current floor indication of an elevator), safety-related features (e.g., door safety), group management features (e.g., scheduling a group of elevators), etc.

5.1 Commonality and Variability Analysis of the ECS Product Line

Commonality and variability analysis is the first step towards the development of product line assets. Fig. 6 shows the result of a commonality and variability analysis of the ECS product line. Features that are common for all elevator products in the ECS product line are modeled as common features, while those that may not be selected for some products are modeled as variable features (denoted by "<<v>>"). *ECS Product Line* includes the *Passenger Service*, *VIP Service*, and *Fire Fighter Service* features. As *VIP Service* and *Fire Fighter Service* may or may not be present in a particular product, they are modeled as optional features. Similarly, *Weight Sensor* is modeled as an optional feature. However, as an elevator needs only one among digital and analog weight sensors, *Digital* and *Analog* are modeled as alternative features.

Configuration dependencies constrain selection of variable features for particular products. For example, as *Anti-Nuisance*[6] requires weight data from *Weight Sensor*, the selection of *Anti-Nuisance* requires the selection of *Weight* Sensor, and therefore the *required configuration dependency* is modeled as shown in Fig. 6.

---

[6] *Anti-Nuisance* cancels all car calls if an excessive number of car calls are registered for the passenger load determined by the weight sensor..

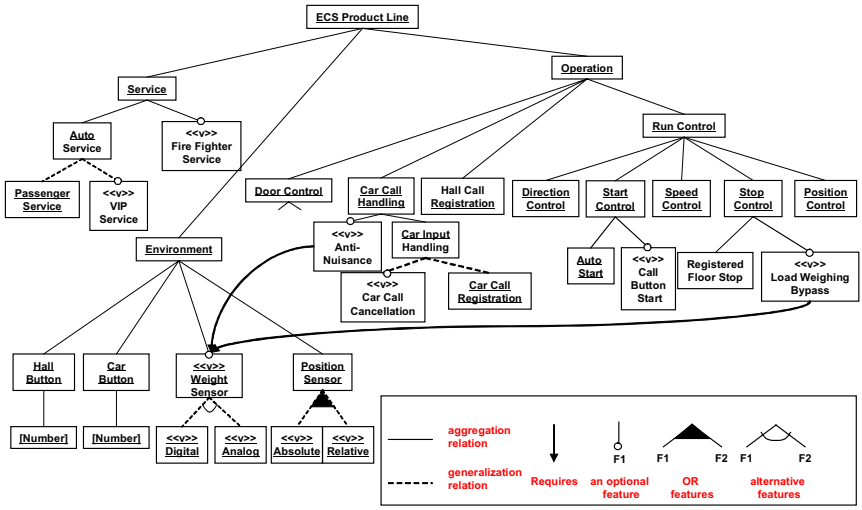

Fig. 6. The feature model of the ECS product line

5.2 Feature Dependency Analysis of the ECS Product Line

Once commonality and variability are analyzed and modeled in the feature model, dependencies among the features in the feature model must be analyzed. Fig. 7 represents a part of *Usage* dependencies between features in the ECS product line. As discussed earlier, if a feature depends on other features in terms of *Usage* dependency, this means that the former feature requires the latter features for its correct functioning or implementation. For example, *Passenger Service* in the ECS product line is a normal driving service of an elevator. During the operation of *Passenger Service*, the elevator registers calls from passengers in the halls or within the car; answers the calls by moving the car to the requested floors; and controls (i.e., opens or closes) doors when it stops. Therefore, as shown in Fig. 7, *Passenger Service* requires *Hall Call Registration*, *Car Call Registration*, *Start Control*, etc. for its implementation. *Start Control* requires correct functioning of *Direction Control*, as it controls movement of the elevator toward the direction decided by *Direction Control*. Moreover, *Direction Control* requires correctness of *Position Control*, as it determines the direction for next movement based on the current position of the elevator.

Modification dependency between two features means that the behavior of a feature is changed by the other. As shown in Fig. 7, *Load Weighing Bypass* changes the behavior of *Registered Floor Stop* by ignoring the registered hall calls when a car is loaded to the predetermined level of capacity. Also, *Car Call Cancellation* changes the behavior of *Car Call Registration* by deleting registered calls.

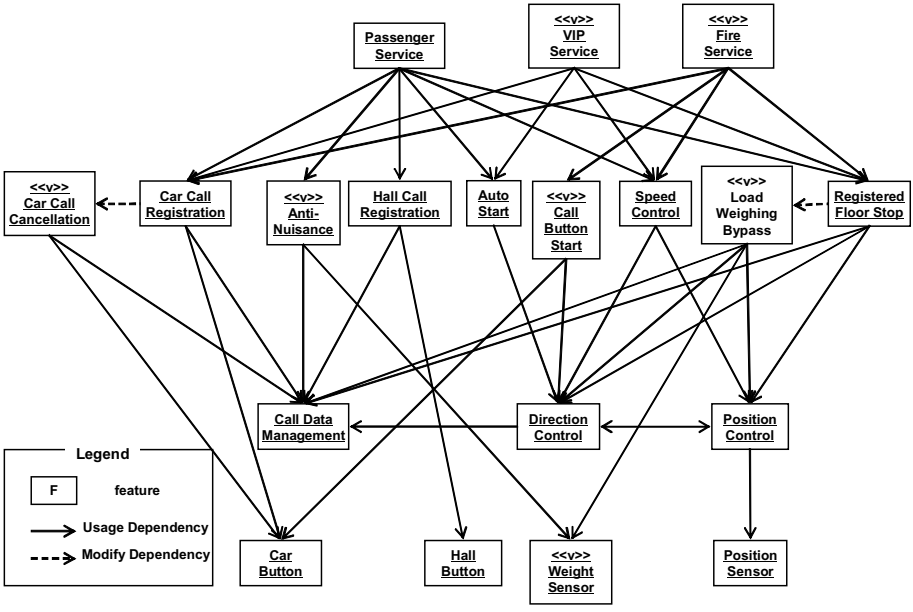

Fig. 7. A part of Usage and Modification dependencies in the ECS product line

In order to represent activation dependencies among features in an effective way, we use Statechart [7]. The rounded rectangle represents the active state of a feature. As shown in the bottom box of Fig. 8, a feature nested by another feature represents that the activation of the former depends on that of the latter. Two features connected by an arrow means that two features are active sequentially. Two features nested by a third feature indicate that the two features cannot be active at the same time. If a line divides two features nested by a third feature, this represents that the two features can be active concurrently. With these notations, we can model the *Subordinate-Activation*, *Concurrent-Activation*, *Exclusive-Activation*, and *Sequential-Activation* dependencies among features effectively.

As shown in the upper part of Fig. 8, *Passenger Service*, *VIP Service*, and *Fire Fighter Service* cannot be active at the same time. This means that an elevator in the ECS product line can provide only one of *Passenger Service*, *VIP Service*, and *Fire Fighter Service* at a time. While *Passenger Service* is active, *Car Call Registration* and *Car Call Cancellation* must not be active at the same time, and *Anti-Nuisance* can be active after the completion of *Car Call Registration*. In addition, *Hall Call Registration* must be active concurrently with *Car Call Registration*, *Car Call Cancellation*, and *Anti-Nuisance* during the activation of *Passenger Service*. Moreover, *Car-Call Cancellation*, *Anti-Nuisance*, and *Hall Call Registration* can be active only when *Passenger Service* is active. They must not be active while *VIP Service* or *Fire Fighter Service* is active.

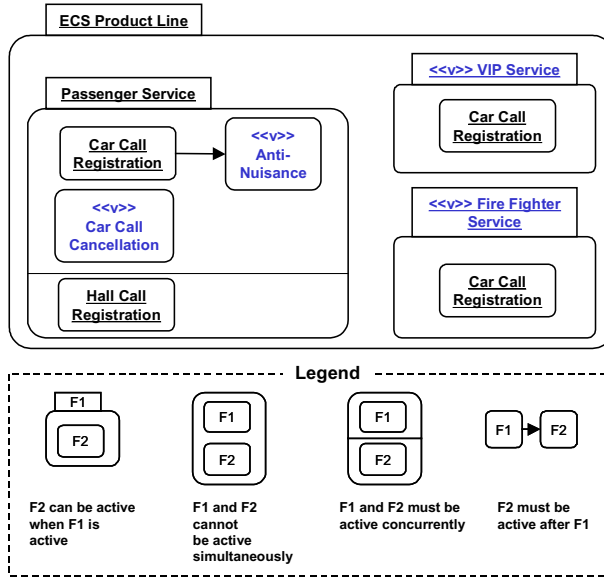

Fig. 8. A part of activation dependencies in the ECS product line

5.3 Component Design of the ECS Product Line

Product line assets must be commonly reusable for production of products and easily adaptable to product specific variations. In this section, we illustrate how the commonality and variability analysis results and the dependency information can be used to design reusable and adaptable product line components for the ECS product line.

As shown in Fig. 7, *Passenger Service* uses *Car Call Registration*, *Anti-Nuisance*, etc. for its implementation. As *Anti-Nuisance* is a variable feature, they must be hidden from the component implementing *Passenger Service* so that their presence or absence will not affect the component for *Passenger Service*. The *CarCallHandler* component in Fig. 9 has responsibilities for hiding the presence or absence of *Anti-Nuisance* from the component for *Passenger Service* and delegating requests from the component for *Passenger Service* to the component that implements *Anti-Nuisance*.

As *Car Call Cancellation* is a variable feature and modifies the behavior of *Car Call Registration,* the presence or absence of *Car Call Cancellation* affects the component for *Car Call Registration*. In order to make the component not changed for the presence or absence of *Car Call Cancellation*, the component for *Car Call Cancellation* (i.e., *CanHandler*) extends the component for *Car Call Registration* (i.e., *RegHandler*) through the inheritance mechanism.

In addition, as shown in Fig. 8, *Car Call Registration* and *Car Call Cancellation* must not be active at the same time, while *Anti-Nuisance* can be active after the completion of *Car Call Registration*. Moreover, *Anti-Nuisance* and *Car Call Cancellation*

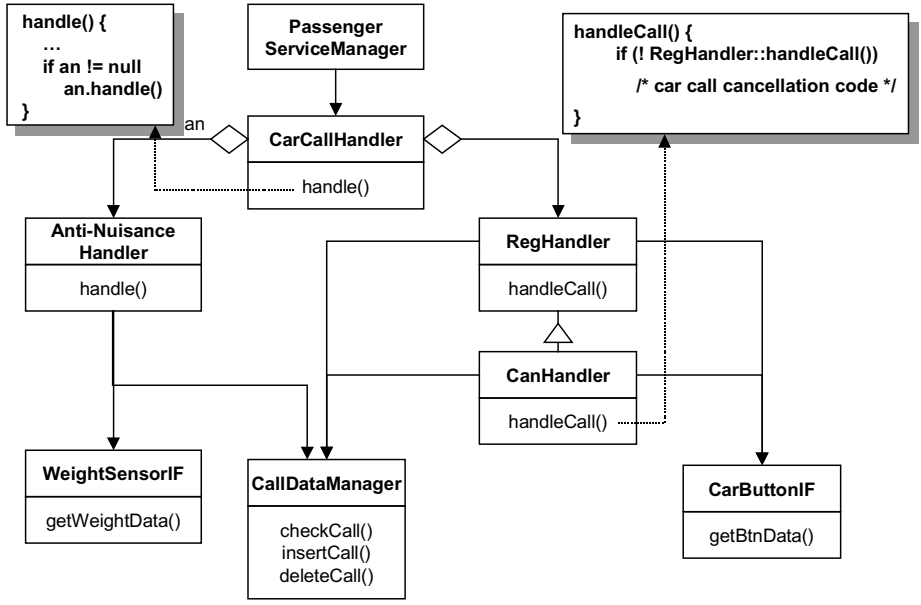

Fig. 9. Hiding variability and separating Modification dependency

must not be active while *VIP Service* or *Fire Fighter Service* is active. Because of these activation dependencies between features, the presence or absence of variable features (e.g., *Car Call Cancellation*, *VIP Service* or *Fire Fighter Service*) may affect other components. In order to handle this problem, we need to further separate activation dependencies from components. As shown in Fig. 10, the activation manager encapsulates activation dependencies between features and coordinates activation of components implementing the features.

6 Related Work

The feature-oriented domain analysis (FODA) [9] was first introduced in 1990. Since then, many product line engineering methods have adopted or extended the feature-oriented commonality and variability analysis as an integral part of product line engineering.

FeatuRSEB [6] extended RSEB (Reuse-Driven Software Engineering Business) [8], which is a reuse and object-oriented software engineering method based on the UML notations, with the feature model of FODA. The feature model of FeatuRSEB is used as a catalogue of or index to the commonality and variability captured in the RSEB models (e.g., use case and object models). Generative Programming [2] also uses the feature modeling as the key to the development and configuration of components for a product family. Although these methods adopted feature modeling to ana-

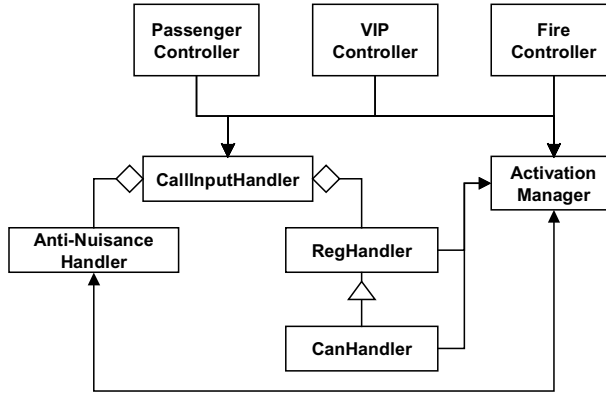

Fig. 10. Separating activation dependency

lyze commonality and variability among products, they do not analyze dependencies among features explicitly. With these methods, dependencies among features are taken into account implicitly during object or component design.

Recently, several attempts have been made to analyze and document dependencies among features explicitly. Ferber at el. [3] identified five types of feature dependencies or interactions (i.e., Intentional Interaction, Resource-Usage Interaction, Environment Induced Interactions, Usage Dependency, Excluded Dependency) in the engine control software domain and extended the feature model in terms of the dependency and interaction view for reengineering a legacy product line. Fey at el. [4] introduced another type of dependency, called "Modify" relation, in addition to "Require" and "Conflict" relations, which are the same as Required and Excluded dependencies in the original FODA method. Although these work identified several dependencies or interactions between features, they did not present how feature dependencies have influences on product line asset development. The primary contribution of this paper is to make explicit connection between a feature dependency analysis and product line asset design by showing how the analysis results can be used to design reusable and adaptable product line components.

7 Conclusion

The feature-oriented commonality and variability analysis method has been considered an essential step for product line asset development. In the past, we applied the method to the core part of elevator control software for development of reusable and adaptable assets. We have experienced that we could easily add new control techniques and new hardware devices without major modification of the existing assets [13]. However, when we tried to incorporate various market specific services of elevators into the assets, we had difficulties in evolving the assets without localizing effects of service additions to one or a few components. This is mainly due to the fact

that many services of elevators are highly dependent on each other but the assets were designed without consideration of various dependencies among services.

To address the difficulties, this paper has extended the method by including feature dependency analysis. In addition, we have made explicit connections between feature dependency analysis and product line component design. The explicit connections can help asset designers not only develop assets envisioned for a product line but also evolve the assets to support the future growth of the product line, as they can provide the rationale of design decisions made during asset development. Currently, we are reengineering the existing assets of elevator control software to validate the proposed method.

In addition to feature dependencies, feature interactions [12] also have significant influences on product line asset development. When products are developed with integration of components implementing various features, these features may interact with each other in unexpected ways. Handling feature interactions may cause significant changes to product line assets, if interaction related code is scattered across many components. To improve reusability and adaptability of product line assets, they must be designed in a way that interaction related code can be confined to a small number of components. Therefore, we are currently extending the method proposed in this paper by also taking feature interaction as a key design driver for product line component development.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns, Addison-Wesley, Upper Saddle River, NJ (2002)
2. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications, Addison-Wesley, Reading, MA (2000)
3. Ferber, S., Haag, J., Savolainen, J.: Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In Chastek, G (ed.): Software Product Lines. 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. In Chastek, G (ed.): Software Product Lines. Lecture Notes in Computer Science, Vol. 2379, Springer-Verlag, Berlin (2002) 198-216
5. Griss, M.: Implementing Product-Line Features by Composing Aspects. In: Donohoe, P. (Ed.), Software Product Lines: Experience and Research Directions. Kluwer Academic Publishers, Boston, 2000
6. Griss, M., Favaro, J., d'Alessandro, M.: Integrating Feature Modeling with the RSEB. In: Proceedings of Fifth International Conference on Software Reuse, (1998) 76-85
7. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8(3) (1987) 231-274
8. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. Addison Wesley Longman, New York (1997)
9. Kang, K. C., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. In: Technical Report CMU/SEI-90-TR-21. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)

10. Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering, 5 (1998) 143-168
11. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. Lecture Notes in Computer Science, Vol. 1241, (1997) 220-242
12. Kimbler, K., Bouma L. G.: Feature Interactions in Telecommunication and Software Systems V. IOS Press, Amsterdam (1998)
13. Lee, K., Kang, K. C., Chae, W., Choi, B. W.: Feature-Based Approach to Object-Oriented Engineering of Applications for Reuse. Software-Practice and Experience, 30 (2000) 1025-1046
14. Lee, K., Kang, K. C., Lee, J.: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In: Gacek C. (ed.): Software Reuse: Methods, Techniques, and Tools. Lecture Notes in Computer Science, Vol. 2319. Springer-Verlag, Berlin (2002) 62-77
15. VanHilst, M., Notkin, D.: Using C++ Templates to Implement Role-Based Designs. In: Proceedings of the 2nd JSSST International Symposium on Object Technologies for Advanced Software (1996) 22-37
16. Vici, A. D., Argentieri, N.: FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry. In: Proceedings of Fifth International Conference on Software Reuse (1998) 166-175
17. Zalman, N. S.: Making the Method Fit: An Industrial Experience in Adopting FODA. In: Proceedings of Fourth International Conference on Software Reuse (1996) 233-235