Aspectual Separation of Feature Dependencies for Flexible Feature Composition

Kwanwoo Lee*, Goetz Botterweck† and Steffen Thiel‡
*Department of Information Systems Engineering
Hansung University, Seoul, Korea 136–792
Email: kwlee@hansung.ac.kr
†Lero
University of Limerick, Limerick, Ireland
Email: goetz.botterweck@lero.ie
‡ Department of Computer Science
Furtwangen University of Applied Sciences, Furtwangen, Germany
Email: steffen.thiel@hs-furtwangen.de

***Abstract*—Aspect-oriented programming (AOP) provides effective mechanisms for improving the modularity of feature implementations. However, as features in general are not independent of each other, changes in the implementation of one feature may cause changes to or side effects in the implementation of other features. We address this challenge by separating feature dependencies from feature implementations using AOP techniques. Specifically, this paper contributes by providing aspect-oriented implementation patterns for feature dependencies (e.g., modification dependency and activation dependency). With the resulting clear separation of dependencies, this approach makes each feature implementation easier to understand and reuse. A product line of scientific calculator applications is used to demonstrate and evaluate the proposed approach.**

***Keywords*-software product line engineering, feature dependency, aspect-oriented implementation patterns;**

I. Introduction

A common way to handle and represent product configurations in software product line engineering are feature models [1], [2], which provide means for managing and representing the product configurations of a software product line (SPL) in terms of feature configurations.

To be able to take feature configurations and derive corresponding product implementations from the core assets, these assets must be designed in a way that allows to switch variable parts on and off depending on the selection of variable features in the product configuration. However, a feature does not always correspond to exactly one implementation component. Hence, if a feature is related to several parts of multiple components, it can become difficult to select and compose corresponding code fragments according to a desired feature configuration. A first step toward flexible feature composition are approaches that modularize feature implementation.

Aspect-oriented programming (AOP) [3] is a good candidate for modularizing feature implementation, as it provides effective mechanisms for encapsulating crosscutting concerns into modular units. There have been several attempts to modularize features using AOP [4], [5], [6]. These approaches allow to *modularize the implementation* of features into separated components, called aspects.

However, this is not sufficient. Since features in general are not independent of each other, changes in the implementation of one feature will cause or side effects in the implementation of other features. This problem is mainly caused by the fact that dependencies between features are *embedded* into feature implementation modules, resulting in tangled code. In this paper, we strive to separate dependencies from feature implementations using AOP techniques.

The remainder of the paper is structured as follows: Section II defines the problem that this paper addresses, and presents an overview of dependencies between features. Section III describes how the feature dependencies can be clearly separated using aspect-oriented implementation patterns. Section IV demonstrates the proposed approach with a calculator product line example and evaluates the approach with some observed metrics. Our approach is compared with other related work in Section V. Section VI concludes the paper.

II. Background

A. Problem

This section describes the core problem that we are addressing in this paper. As a running example we use a product line of calculator applications. We will later use a related case study to discuss and evaluate our approach (see Section IV).

AOP provides an effective way of incorporating variable features into a product, as it can isolate variable features into aspects and integrate them in an additive way. That is, based on the initial base structure implementing the common features, variable features can be implemented using aspects. Then, an aspect weaver creates a product by weaving aspects (implementing variable features) into the base modular structure.

For instance, a product line of calculators may have features such as `History` (to maintain a list of previously evaluated expressions), `Mode` (to switch between different modes for display and angles), and `Number Systems` (to switch between numeral bases such as decimal and binary).

The feature `History` can be implemented using AspectJ as shown in Figure 1. To simplify, we only show an excerpt and focus on the code that implements the upward navigation through the history list.

```
1. public aspect HistoryModule {
2.   HistoryModule() {
3.     uBtn = new UpButton();
4.   }
5.   after(CalculatorPanel p):
6.             SomeExtensionPoint(p) {
7.     p.add(uBtn);
8.   }
9.   class UpButton extends CalculatorButton {
10.    public void actionPerformed(..) {
11.      upHistory();
12.      ...
13.    }
14.  }
15.  public void upHistory() {
16.    // set the previous expression list to parser
17.    parser.setList(history.elementAt(
18.                         historyPosition).exp_list);
19.    /** ------- modification by NumberSystems -----
20.     * parser.setNumSys(history.elementAt(
21.     *                   historyPosition).numSys);
22.     */
23.    ...
24.  }
25.  private java.util.Vector<HistoryItem> history;
26.  int historyPosition;
27.  UpButton uBtn;
28.}
```

Figure 1. The Aspectual Implementation of the Feature `History`.

To add the feature `History`, the calculators must be extended with a GUI button for upward navigation. The aspect `HistoryMoudle` creates an `UpButton` instance (line 3) and adds it to the calculator GUI Panel (line 7), after the join point specified at the pointcut `SomeExtensionPoint(p)` (lines 5–8). The `UpButton` instance calls the method `upHistory()` (line 11), which implements the functionality for navigating through previous expressions in the history list and passing a selected expression to the calculator parser (lines 17–18), when the corresponding GUI button is pressed.

We can implement other features, e.g., `Number Systems`, in a similar fashion. However, we have to consider the interactions between features. For instance, since `Number Systems` can change the numeral base of an expression during its operation, the behavior of `HistoryModule` must be extended to maintain the numeral base for each expression in the list (lines 20-21). Thus, the inclusion or exclusion of `Number Systems` affects the aspectual implementation of `History`.

In addition, `Mode`, which allows to change the mode of the calculator, prevents `History` from being activated, as both features use the same shared resource (i.e., display panel). In other words, `History` can or cannot be active depending on the activation state of `Mode`.

Such dependencies are affecting the implementations of particular features, *varying* with the presence of the involved features. For instance, the dependency from `History` to `Mode` is relevant only when both features are configured for the same product. Consequently, if we embed the dependency into one of the feature implementations, a variation (i.e., a different choice for the product configuration) can cause undesired changes in the feature implementations. This corresponds to the observation in Liu et al. [6], "the implementation of a feature can vary from one program to another."

This invasive change problem mainly comes from lack of understanding of feature dependencies [7]. Without clear identification and separation of feature dependencies, code implementing feature dependencies may be scattered across multiple aspectual or modular components. Consequently, there is a need to better understand and handle feature dependencies. We will start in the next section by defining different types of dependencies.

B. Feature Dependencies

The *feature model* [1], [2] describes the capabilities of a product line including configuration options for products in terms of features. Configuration dependencies (e.g., *Configuration Inclusion* and *Configuration Exclusion*) between features constrain the configuration choices of the feature model. In addition to configuration dependencies (at design-time), there are operational dependencies (at run-time) that describe interactions between functional features. Within run-time dependencies, this paper focuses on *Modification* and *Activation* dependencies [7] that have crosscutting effects. We are interested in these dependencies since their variation may affect feature implementations.

Note that the *Usage* dependency [7], [8], which corresponds to *Inform* and *Data Flow* dependencies in [9], is the most common form of feature dependency. However, we are not interested in this dependency because *Usage* dependencies do *not* vary independent from their associated features. Instead, features involved in a *Usage* dependency must be always configured together [9].

A *Modification* dependency between two functional features means that the behavior of one feature is extended or modified in presence of the other feature. For instance, as explained earlier, the feature `History` has a modification dependency with `Number Systems`. This corresponds to similar the dependency types *Influences* [9] and *Intentional Interaction* [8].

Activation dependencies affect the activation of functional features. They can be classified into four categories: *Ex-*

cluded Activation, *Required Activation*, *Concurrent Activation*, and *Sequential Activation*.

An *Excluded Activation* dependency (known as *Excluded Dependency* in [8]) between two functional features describes the fact that one feature excludes the execution of the other. For example, `Mode` prevents the execution of `History`, as illustrated before. `Off`, which turns off the power of the calculator, can preempt the execution of `History`, as `Off` has the highest priority among all features in the calculator product line. Features in the *Excluded Activation* dependency usually access shared resources, which have to be used exclusively for their correct operation. Therefore, *Resource-Usage Interaction* in [8] and *Resource-Configure Interaction* in [9] are closely related to the *Excluded Activation* dependency.

A *Required Activation* (*Subordinate Activation* in [7]) dependency describes that one feature can be active only while the other is active. For example, `Boolean Operators` requires the activation of `Binary Number System` for its correct operation.

Concurrent Activation and *Sequential Activation* dependencies mean that one feature must be active with the other feature, concurrently or sequentially, respectively. For example, `STO`, which stores the result of a calculation in memory, has to be active immediately after `Evaluation`, which performs the calculation of a given expression.

These dependencies between functional features must be analyzed before implementing features to prepare the implementation for variations. To this end, we will now discuss how each type of the dependencies can be clearly separated from feature implementations using aspects.

III. Dependency Aspects

As discussed earlier, we suggest that dependencies should be handled as separate entities. To avoid improvised ad-hoc solutions, repeatable well-known patterns for the implementation of dependencies would be helpful. To this end we now describe aspect-oriented implementation patterns for feature dependencies.

A. Modification Dependency Aspects

A modification dependency from a feature `F1` to a feature `F2` implies the functional behavior related to `F1` can be divided into two parts: the functional core and the interaction. The functional core indicates the main functionality of `F1`, while the interaction represents the behavior extended by `F2`. The interaction behavior represents the modification dependency from `F1` to `F2`. Since the interaction behavior may or may not be present depending on the presence of `F2`, separating it from the implementation of the functional core allows the implementation of the functional core to be reused 'as-is' in various product contexts.

We can implement the interaction part (i.e., modification dependency) of `F1` using an AspectJ aspect (See Figure 2.). The pointcut `modification-point` specifies the join points at which the interaction behavior created by `F1` and `F2` is applied. In the advice body, the interaction behavior is defined. The pointcut specification or advice body may use the functionality or data defined in the modules (`CoreFuntionF1` and `CoreFunctionF2`) implementing the core functionality of the features. Note that the stereotype `module` represents the modularized unit of feature implementation. Thus, the modules `CoreFuntionF1` and `CoreFunctionF2` can be realized as either Java classes or AspectJ aspects.

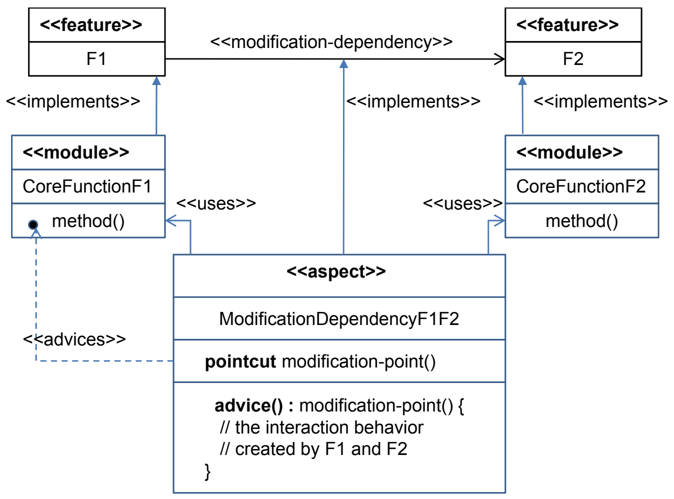


Figure 2. Modification Dependency Aspect.

Since the modification dependency can be defined as a separate aspect, the aspect can be included whenever both involved features are selected.

B. Activation Dependency Aspects

The activation dependencies described in Section II-B can be implemented as separate *generic* aspects, which declare *generic* types with type *parameters*. Each generic aspect provides the abstract interfaces that must be concretized in its concrete sub-aspects and defines the template behaviors, which the concrete sub-aspects inherit.

The importance of using generic aspects is twofold. First, they are reusable artifacts among the aspects implementing the activation dependencies, as they abstract out common interaction patterns between features in the activation dependencies. Second, this way the overall code size of all dependency implementations can be reduced, as each dependency can simply be implemented by extending the abstract interfaces of a generic aspect and inheriting the template behaviors of the abstract aspect.

Features in an **Excluded Activation Dependency** must not be active at the same time. Figure 3 shows how this requirement can be implemented in a generic way:

```
1. public abstract aspect ExcludedAct<M1,M2> {
2.   M1Interface source;
3.   M2Interface target;
4.
5.   interface M1Interface {
6.     public void terminate();
7.   }
8.   interface M2Interface {
9.     public boolean isActive();
10.  }
11.  declare parents: M1 implements M1Interface;
12.  declare parents: M2 implements M2Interface;
13.
14.  abstract pointcut M1allExeJP();
15.  void around() : M1allExeJP() {
16.    if ((target!=null) && target.isActive())
17.      source.terminate();
18.    else
19.      proceed();
20.  }
21.}
```

Figure 3. Excluded Dependency Aspect.

`ExcludedAct<M1,M2>` is a generic aspect which can be parameterized with the module types `M1`, `M2`. The `declare` mechanism is used to extend the actual type of `M1` with `M1Interface`. Then, the actual type of `M1` can have the method `terminate()`, which handles the abnormal termination of its instance. Similarly, the actual type of `M2` is extended with the method `isActive()`, which returns a boolean value representing whether or not the instance of that type is active. Note that these methods must be defined in a sub-aspect extending `ExcludedAct<M1,M2>`. These methods are used to manage the excluded activation dependency between the instances of `M1` and `M2`. The abstract pointcut `M1allExeJP` represents all execution join points related to an instance of `M1`. Therefore, the `around` advice (lines 15–20) means that each execution join point matched by the pointcut can proceed only when the instance `target` is not active, and otherwise the instance `source` must be terminated.

A **Required Activation Dependency** is similar to the excluded activation dependency in the sense that depending on the activation state of the `M2` type instance, the activation of the `M1` type instance is affected. However, here the consequence is the opposite as for the excluded activation dependency. As you can see in Figure 4, only the advice body (lines 16–19) is different from that of `ExcludedAct<M1,M2>`. That is, each execution join point matched by the pointcut can proceed only when the instance `target` is active, and otherwise the instance `source` must be terminated.

Features in a **Sequential Activation Dependency** must be active sequentially. `SequentialAct<M1>`, shown in Figure 5, implements the sequential activation dependency. The aspect extends the actual type of `M1` with `M1Interface` (line 7) which includes the abstract method `activate()`

```
1.   public abstract aspect RequiredAct<M1,M2> {
2-14. // the same as the ExcludedAct<M1,M2>
15.    void around() : M1allExeJP() {
16.      if ((target!=null) && target.isActive())
17.        proceed();
18.      else
19.        source.terminate();
20.    }
21.  }
```

Figure 4. Required Dependency Aspect.

(line 5). The method is called immediately after a `M2` type instance completes its execution (lines 10–12). Note that the abstract pointcut `M2lastExeJP` represents the last join point matched while a `M2` type instance is executing.

```
1.  public abstract aspect SequentialAct<M1> {
2.    M1Interface source;
3.
4.    interface M1Interface {
5.      public void activate();
6.    }
7.    declare parents: M1 implements M1Interface;
8.
9.    abstract pointcut M2lastExeJP();
10.   after() returning: M2lastExeJP(){
11.     source.activate();
12.   }
13. }
```

Figure 5. Sequential Dependency Aspect.

Features in a **Concurrent Activation Dependency** must be active together concurrently. `ConcurrentAct<M1>` in Figure 5 shows how to ensure that the `M1` type instance (`source`) starts concurrently with a `M2` type instance. To be able to execute instances of `M1` and `M2` concurrently, the aspect extends the actual type of `M1` with the interface `Runnable` and executes the instance (i.e., `source`) of that type in a separate thread before starting the execution of the `M2` type instance.

```
1.  public abstract aspect ConcurrentAct<M1> {
2.    Runnable source;
3.
4.    declare parents: M1 implements Runnable;
5.
6.    abstract pointcut M2startExeJP();
7.    before(): M2startExeJP(){
8.      new Thread(source).start();
9.    }
10. }
```

Figure 6. Concurrent Dependency Aspect.

It should be noted, that the generic aspects described above are not the only way of implementing such dependencies, especially if we have to take into account different contexts. The main point in this section was to demonstrate, how we can clearly separate feature dependencies from feature implementations with aspect-oriented patterns.

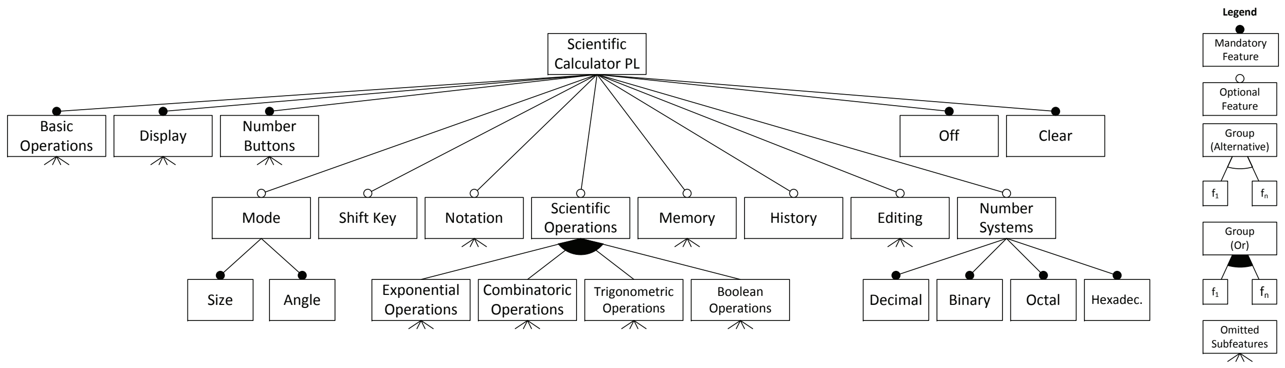
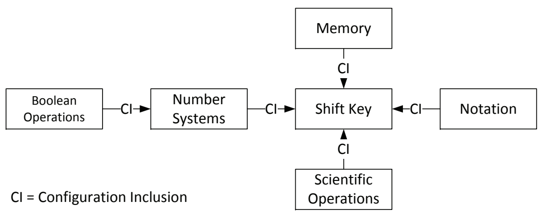
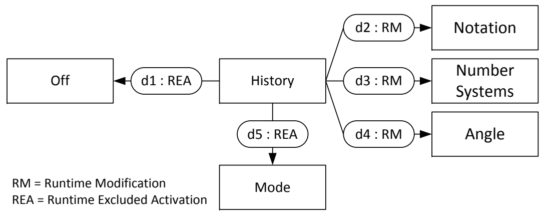


Figure 7. Feature Model (excerpt).



(a) Configuration Dependencies



(b) Operational Dependencies

Figure 8. Examples for Feature Dependencies (excerpts).

These patterns can help implementing feature dependencies with less efforts and less code redundancy. Moreover, the modularization of dependencies supports our goal of making feature implementations independent of each other.

IV. Case Study

To illustrate the application of the concepts introduced earlier we will now use a product line of calculator applications, which we created by refactoring the open source *Java Scientific Calculator* [10]. We chose calculator applications since most of the readers will be familiar with the functionality and to compare our concepts with other approaches (e.g., FOP [11]), which use similar examples. The activities performed during our approach can be roughly structured into Feature Analysis, Feature Implementation Mapping and Aspectual Implementation.

A. Feature Analysis

Feature analysis, which is the starting point for our approach, includes activities for identifying and modeling the commonalities and variabilities of a product line in terms of features and for analyzing dependencies between features.

Figure 7 shows the feature model of our calculator product line. It contains mandatory features representing commonalities, e.g., `Basic Operations`, `Display`, `Number Buttons`, and optional features indicating variabilities, e.g., `Mode`, `History`, `Number Systems`, etc.

During feature configuration, the software engineer decides which features will be included in the particular product. This process is constrained by the configuration dependencies as captured in the feature model (see Section II-B). For instance, the feature `Boolean Operations` requires the (design-time) inclusion of `Number Systems`, which allows to change the current numeral base of the calculator from `Decimal` to `Binary`.

Operational dependencies (at run-time) further constrain the execution of features. For instance, Figure 8(b) shows operational dependencies that are associated with the feature `History`. As described in Section II, the activation of `Mode` prevents `History` from being activated, and `Off` preempts the execution of `History`. In addition, `Number Systems` requires a modification in the behavior of `History`. Similarly, `Notation` and `Angle` also require an behavioral extension of `History`.

All these dependencies are explicitly modeled in the feature model, which is the first step towards clear separation of dependencies from feature implementations.

B. Feature Implementation Mapping

In this section, we describe how features and feature dependencies can be mapped into implementation components, such that they can be effectively configured based on the feature configurations of a SPL. We used the following techniques to organize the mappings:

Mapping non-crosscutting common features to a set of

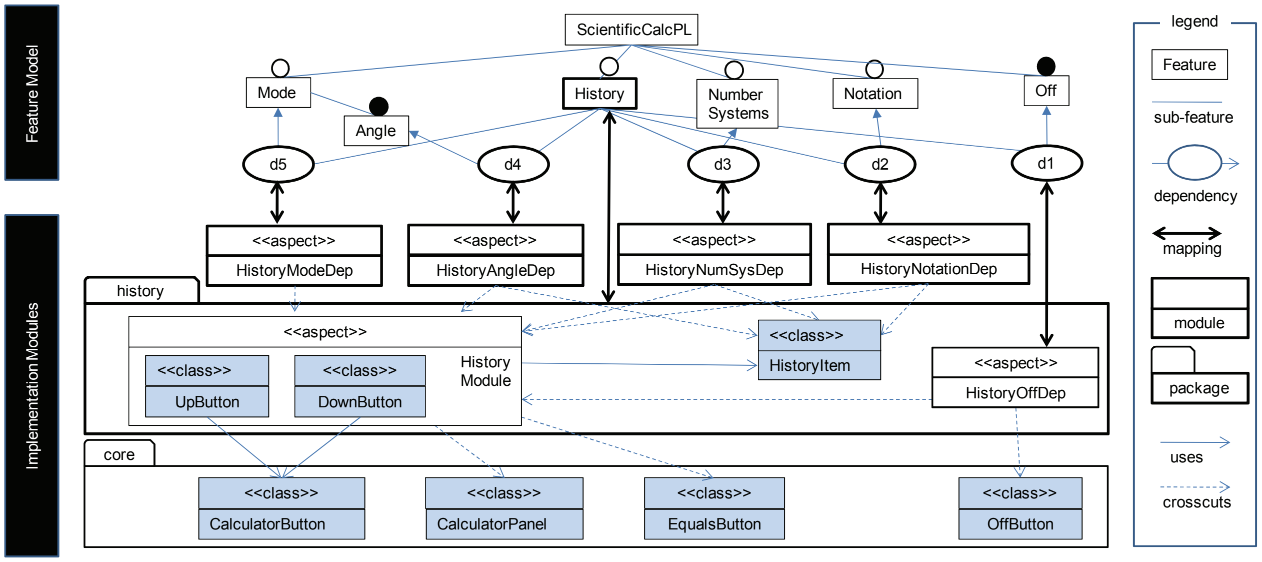


Figure 9. Feature-Implementation Mappings.

classes which constitute a base modular structure. As shown in Figure 7, the common features of the calculator product line (PL) include `Basic Operations`, `Display`, `Num Buttons`, `Off`, and `Clear`. Since these can be modularized into components, we can implement them as a set of Java classes forming the base structure of the PL. The lower part of Figure 9 shows some of the classes in the `core` package, which implement all common features.

Mapping each variable feature and its dependencies with common features to an aspectual component or a package. Dependencies related to a variable feature may also vary themselves, depending on the presence of the variable feature. One effective way of localizing the effect of a feature variation is to encapsulate all dependencies related to a feature into the components for the corresponding feature. For example, in Figure 7, the feature `History` is an optional feature. If it is deselected and removed from the implementation, `d1`, the dependency from `History` to `Off` (see Figure 8(b)) must be removed from the product as well. Therefore we encapsulate the components for `History` and `HistoryOffDep` implementing the dependency `d1` into the same package `history` (see Figure 9).

Mapping each variable dependency to a separate aspectual component. In case a variable feature has dependency relationships with other variable features, we need to identify variations of these dependencies and implement them as separate aspects. For example, since both `History` and `Number Systems` are optional features, the operational dependency between them is present if (and only if) both of them are present. Therefore, we separate the dependency from the components implementing the features and realize it as the aspectual component `HistoryNumSysDep`. This approach decouples the implementation of the feature `History` from implementations of other features (see Figure 9).

C. Aspectual Implementation

In this section, we link the concepts from two earlier sections by describing how dependencies (described in the previous section) can be implemented using the aspect-oriented implementation patterns (presented in Section III).

Figure 10 shows the aspectual implementation of the modification dependency between `History` and `Number Systems`. As shown in Figure 9, `HistoryNumSysDep` crosscuts the aspect `HistoryModule` and the class `HistoryItem`. The dynamic crosscutting mechanisms (i.e., *Pointcut* and *Advice*) of AspectJ are used to extend the aspect `HistoryModule`. Lines 5–7 of `HistoryNumSysDep` show how to specify the modification point (the call to the method `setList(..)` within the method `upHistory()`) of the aspect `HistoryModule` using the pointcut mechanism. After the modification point of the aspect `HistoryModule`, the behavior in lines 10–13 of `HistoryNumSysDep` is executed. On the other hand, the class `HistoryItem` is extended through the static crosscutting mechanism (i.e., *Intertype Declaration*) of AspectJ. Line 16 shows how to extend the class `HistoryItem` with the field `base`.

The activation dependencies can be easily implemented using the generic aspects. As shown in Figure 11, the aspect `HistoryModeDep` extends the parameterized abstract aspect `ExcludedDep<HistoryModule,ModeModule>`, which defines the interface variables, the abstract interfaces and the template behavior for an excluded activation dependency. The interface variables and the abstract interfaces

```
privileged aspect HistoryNumSysDep {
  history = HistoryModule.aspectOf();
  numSys = NumSys.aspectOf();

  pointcut modification-point() :
    call(* setList(..)) &&
    withincode(* upHistory());
  ...
  after() :modification-point() {
     HistoryItem item
      = history.historylist.elementAt(
                history.historyPosition);
    numSys.setBase(item.base);
  }
  ...
  public Base HistoryItem.base;
}
```

Figure 10. Modification Dependency Aspect Example.

must be concretized in the aspect `HistoryModeDep`. That is, the variables `source`, `target` defined in the generic aspect `ExcludedDep` are initialized with the actual instances of `HistoryModule` and `ModeModule`, respectively (lines 3–10). The two methods `terminate`, `isActive` are implemented (lines 11–13), and the abstract pointcut `M1allExeJP` is specified (lines 16–18).

```
1.  privileged aspect HistoryModeDep
2.   extends ExcludedDep<HistoryModule,ModeModule> {
3.    before(HistoryModule h):
4.      execution(UpButton.new() && this(h) {
5.        source = h;
6.    }
7.    before(ModeModule m):
8.      execution(ModeButton.new()) && this(m){
9.        target = m;
10.   }
11.   public boolean HistoryModule.terminate() {}
12.   public boolean ModeModule.isActive() {
13.     return (getMode()!=0);
14.   }
15.
16.   pointcut M1allExeJP() :
17.     execution(void actionPerformed(..)) &&
18.     within(HistoryModule);
19. }
```

Figure 11. Excluded Activation Dependency Aspect Example.

D. Evaluation and Discussion

The *Java Scientific Calculator* [10] originally has 185 classes and 2 interfaces defined in 8K logical executable lines of code (KLOC). We refactored a considerable number of features in the *Java Scientific Calculator* (to allow for configuration), but some code (1.7 KLOC) remained unrefactored.

Although we implemented the product line by refactoring the existing open source, this paper does not focus on the refactoring. Our concern is how variabilities and dependencies analyzed during the feature analysis can be utilized for structuring the product line implementation in a way that facilitates efficient product derivation.

As shown in the feature model (Figure 7), our calculator product line contains twelve major variable features and thirteen common features. The core of the product line, the common features, is implemented by 52 classes in 2.5 KLOC (38% of the code size), while 155 aspects and 98 classes in 4 KLOC (62% of the code size) implement variable features and dependencies. After refactoring, we have 6.5 KLOC in total for the product line, corresponding to 3% increase in code size compared to 6.3 KLOC for the original monolithic application without variability.

Of the implementation for variable features, about 77% (95 classes and 99 aspects) implement features and 23% (3 classes, 56 aspects) dependencies (17 for *modification*, 18 for *excluded activation*, 18 for *required activation*, and 3 for *sequential activation*).

Initially, when starting the research on the presented case study, we concentrated on clearly separating dependencies from feature implementations. During this work we then noticed that redundant code fragments reappeared in different dependency implementations. This inspired the development of the generic aspects presented in Section III. Using these aspects, we could further reduce the code size. In addition, experience during the refactoring seems to indicate that using the described generic aspects can reduce development effort for implementing feature dependencies. We intend to investigate this in more detail.

We also handled dependencies among dependency modules as separate modules. For example, some dependency modules interacted with each other at the same join point. In this case we defined a separate aspect which could prioritize the sequence of interactions. In general dependencies or interactions among dependency modules can also be modularized as separate aspects. This separation can increase the number of implementation modules. This complexity motivated us to develop the tools supporting product derivation.

V. Related Work

Aspect-oriented programming (AOP) techniques and languages were originally developed to modularize crosscutting concerns. Recently, there have been several efforts to use them for modularizing feature implementation. These include work by Griss [12], Alves et al. [13], Kästner et al. [5], Godil et al. [4] and Liu et al. [6].

Most of these approaches did not take into account dependencies or interactions between features during feature implementation. Liu et al. [6], however, observed that “the implementation of a feature can vary from one program to another.” In our view, the problem with varying feature implementation mainly comes from *embedding* relevant dependency code into the feature implementation. Although Liu et al. identified the modules (the derivative modules in [6]) implementing interactions or dependencies with other features, they regarded them as part of the feature implementation instead of separating them. Our main contribution (compared to [6]) is that we make an explicit connection between feature dependencies and dependency modules. Also we provide concrete implementation patterns (the dependency aspects) for implementing dependencies.

There have been several attempts to separate dependencies from feature implementation. Lee et al. identified problems with feature dependencies in SPL component development and proposed guidelines on how feature dependency information can be used for implementing features using object-oriented patterns [7] and using AOP [14]. In this paper, we extended this work by providing the aspect-oriented implementation patterns.

The dependency aspects were inspired by the idea of *Relationship Aspects* [15], which implement relationships between objects as separate aspects. In contrast to *Relationship Aspects*, our paper focuses on dynamic interactions rather than static relationships.

In this paper, we used AspectJ to isolate dependencies from feature implementations. As pointed out in [5], 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 [16] can be used to clarify the separation of base and extensions. The dependency aspects also provide abstract interfaces that decouple aspect code implementing dependencies from feature implementations, but does not provide semantic checking which prevents undesirable composition.

VI. Conclusions

The primary goal of this work is to make the process of product derivation more efficient. To this end, we have presented an approach to separating features dependencies from feature implementations for flexible feature composition. The approach was demonstrated and evaluated with a calculator product line.

Although, we have demonstrated the feasibility and applicability of separating the feature dependencies from feature implementation, we did not take into account all potential types of dependencies between features. For instance, a system with more dynamic characteristics would raise new concerns, both (1) in the way feature interactions would affect the design and implementations of individual features and (2) in the complexity of feature interactions.

The main contribution of this paper is to make an explicit connection between feature dependencies and their aspect-oriented implementation patterns, called dependency aspects, which in the end supports efficient product derivation.

Acknowledgment

This work was supported, in part, by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD, Basic Research Promotion Fund) (KRF-2008-013-D00114), also in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre (www.lero.ie).

References

[1] K. Czarnecki and U. W. Eisenecker, *Generative Programming*. Reading, MA, USA: Addison Wesley, 2000.

[2] K. C. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature Oriented Domain Analysis (FODA) feasibility study," Tech. Rep., 1990.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP 1997*, 1997, pp. 220–242.

[4] I. Godil and H.-A. Jacobsen, "Horizontal decomposition of prevayler," in *CASCON 2005*, 2005, pp. 83–100.

[5] C. Kästner, S. Apel, and D. Batory, "A case study implementing features using AspectJ," in *SPLC 2007*, September 2007, pp. 223–232.

[6] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *ICSE 2006*, 2006, pp. 112–121.

[7] K. Lee and K. C. Kang, "Feature dependency analysis for product line component design," in *ICSR 2004*. Madrid, Spain: Springer, July 2004, pp. 69–85.

[8] S. Ferber, J. Haag, and J. Savolainen, "Feature interaction and dependencies: Modeling features for reengineering a legacy product line," in *SPLC'02*, 2002, pp. 235–256.

[9] W. Zhang, H. Mei, and H. Zhao, "A feature-oriented approach to modeling requirements dependencies," in *RE05*, 2005.

[10] "Java scientific calculator," *http://jscicalc.sourceforge.net*, May 2008.

[11] D. Batory, "Feature-oriented programming and the AHEAD tool suite," in *ICSE '04*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 702–703.

[12] M. L. Griss, "Implementing product-line features by composing aspects," in *SPLC 2000*, August 2000, pp. 271–288.

[13] V. Alves, P. Matos Jr., L. Cole, P. Borba, and G. Ramalho, "Extracting and evolving mobile games product lines," in *SPLC 2005*, September 2005, pp. 70–81.

[14] K. Lee, C. K. Kang, and M. Kim, "Combining feature-oriented analysis and aspect-oriented programming for product line asset development," in *SPLC 2006*, 2006.

[15] D. J. Pearce and J. Noble, "Relationship aspects," in *AOSD 06*, Bonn, Germany, March 2006.

[16] G. William, M. Shonie, K. Sullivan, Y. Song, N. Tewari, Y. Cai, and H. Rajan, "Modular software design with crosscutting interfaces," *IEEE Software*, January/February 2006.