Introducing Variability into Aspect-Oriented Modeling Approaches

Philippe Lahire[2], Brice Morin[1], Gilles Vanwormhoudt[3], Alban Gaignard[2], Olivier Barais[1], and Jean-Marc Jézéquel[1]

[1] IRISA Rennes, Projet Triskell, IRISA - Campus de Beaulieu F-35042 Rennes Cedex
[2] I3S Nice-Sophia Antipolis, Equipe Rainbow, I3S-UNSA
Les algorithmes, 2000 route des lucioles BP 121 F-06903 Sophia-Antipolis Cedex
[3] GET Telecom-Lille 1/ LIFL, Université de Lille 1, F-59655 Villeneuve d'Ascq Cedex

Abstract. Aspect-Oriented Modeling (AOM) approaches propose to model reusable aspects, or cross-cutting concerns, that can be composed in different systems at a model or code level. Building complex systems with reusable aspects helps managing software complexity. But in general, reusability of an aspect is limited to a particular context. On the one hand, if the target model does not match the template point-to-point, the aspect cannot be applied. On the other hand, even when it is actually applied, it is woven into the target model always in the same way. In this paper[1], we point out the needs of variability in the AOM approaches and introduce seamless variability mechanisms in an existing AOM approach to improve reusability. Our aspects can fit various contexts and can be composed into the base model in different ways. Introducing variability into AOM approaches will turn standard aspects into highly reusable aspects.

1 Introduction

The Aspect Oriented Software Development (AOSD) paradigm first appeared at the code level a decade ago [7] with the most famous AOP language AspectJ [6]. The aspect paradigm offers a new way to construct complex systems by composing crosscutting concerns with the base system. In the earlier stages of the software life-cycle, several Aspect-Oriented Modeling approaches (AOM) already exist [1,2,4,16], with various levels of abstraction (requirement, design, architecture). In general, these approaches decrease the complexity of systems by composing models that represents the different concerns of the system (business, security, persistence ...). To help developers saving time designing systems and therefore reduce the time-to-market of these systems, models should be reusable.

Currently, AOM approaches provide some means to design reusable and flexible aspects. But, reusability and flexibility are often limited. In general, they

[1] This work was partially supported by the French National Research Agency (RNTL FAROS Project).

G. Engels et al. (Eds.): MoDELS 2007, LNCS 4735, pp. 498–513, 2007.
© Springer-Verlag Berlin Heidelberg 2007

describe one possible variant of an aspect and propose one possible way to integrate it. For example, a designer cannot model a design pattern in its full genericity with these approaches: he can only model one specific implementation choice for this design pattern. Consequently, aspects are only reusable in similar or very related contexts. In this paper, we argue that aspects must be reusable in various contexts. Designing context independent aspects requires seamless variability mechanisms for specifying the weaving, the pointcut expression, *etc...* . Such mechanisms will turn standard aspects into highly reusable and flexible pieces of models. The contribution of this paper is to point out the needs of variability in the AOM approaches, to provide some mechanisms to support variability in one particular AOM approach and to illustrate these new mechanisms on a concrete example. To address variability in software development, Software product lines (SPL) offer some mechanisms to support functional variability[2] and to derive products that match the user's needs. However, this variability only concerns the software module specifications. In the case of AOM approaches, variability should also be applied onto the composition mechanisms.

The remainder of this paper is organized as follows. Section 2 points out the needs of variability in the AOM approaches with a motivating example. Section 3 presents an overview of an AOM approach. This approach is extended in the section 4 to support variability mechanisms. Section 5 describes a metamodel for this approach and the implementation of a modeling tool. Section 6 presents related works and section 7 concludes and discusses future work.

2 Motivating Example

To illustrate the needs of variability in the AOM approaches, we use the example of a mobile phone device. Figure 1 shows a simplified class diagram presenting the main functionalities of an accountancy package for a mobile phone.

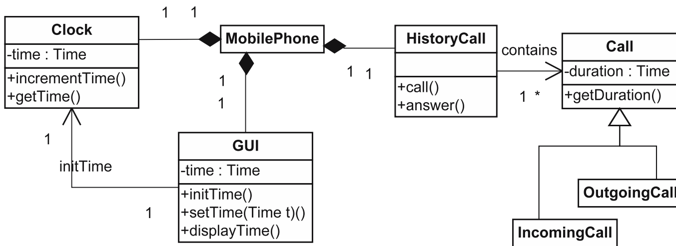

Fig. 1. A Simplified class diagram of the mobile phone

When the user is calling (resp. is called by) someone, the *HistoryCall* class creates a new outgoing (resp. incoming) call and saves the duration. The class *GUI* can display its local variable *time* which is initialized when switching on the phone. The class *Clock* only contains a variable *time* which is incremented every minute.

[2] see Software Product Line Conferences : http://www.splc.net

2.1 Matching Variability

Two optional requirements, **total calls** and **total outgoing calls**, can be added to our mobile phone in order to compute the total duration of the (outgoing) calls. We will use the Counter pattern [11] to realize these two requirements.

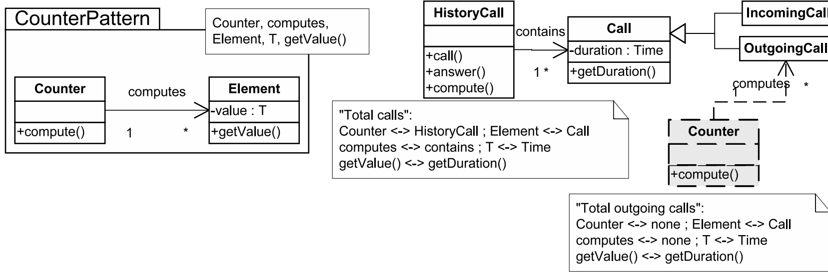

Fig. 2. The Counter pattern realizing the **total calls** requirement

In most of the AOM approaches [2,11,16], a template specifies the model elements of the concern that have to be bound with target model elements. Reusability is then limited to iso-structural target models because if the structure does not match the template point-to-point, the aspect cannot be applied.

Figure 2 shows the Counter pattern composition into the mobile phone model. In order to realize the **total calls** requirement, we use the existing *HistoryCall* and *Call* classes to respectively act as *Counter* and *Element*. We now want to realize the **total outgoing calls** requirement in a separate Counter class. This requires the creation of a new class acting as a Counter and the insertion of a new association between this class and the *OutgoingCall* class. Instead of modifying the base model to this end, it would be more efficient that the Counter pattern automatically introduces all these missing elements. However this is not possible with classic AOM approaches [2,11,16] because the weaving process of the aspect upon the base system can not vary depending on the bindings.

2.2 Adaptation Variability

One optional requirement, **display time**, can be added in order to display and update the time every minute, when the internal clock is updated. The Observer pattern will realize this requirement, notifying the *GUI* (Observer) that the *Clock* (Subject) has been updated.

In most of the AOM approaches [2,11,16], aspects are composed into the target model using one composition rule at a time, offering poor flexibility. Depending on the context, it would be very useful to easily switch between different composition rules. In the context of embedded systems it may be preferable to reduce the number of classes because of memory limitations, and completely merge the aspect while in some other cases, it may be preferable to compose the aspect

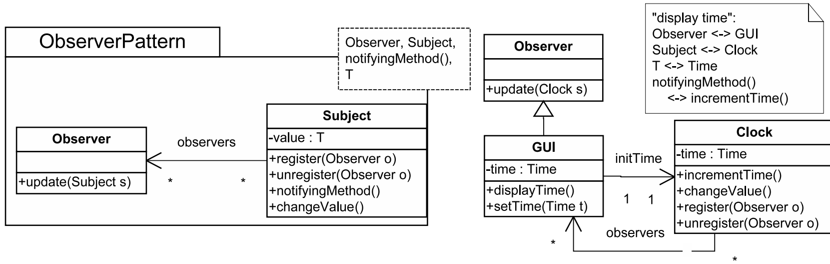

Fig. 3. The Observer pattern merged into the base model

by inheritance in order to improve readability. Figure 3 illustrates another composition rule where *Subject* is merged into *Clock* whereas *GUI* inherits from *Observer*.

This motivating example has shown the needs of variability in two contexts i.e. matching and adaptation. There is also a need for functional variability e.g. how to design many versions of the Counter pattern (total or average for example). Since Software Product Line approaches [17] can help modeling this kind of variability, we do not cover them into this paper.

3 An AOM Approach Overview

The approach which is presented in this paper is only one among many possible approaches for addressing AOM [3,11]. It focuses on providing capabilities for concerns (functional or extra-functional) to be reused. In this context, the expressiveness of the concern modeling is not a primary objective. For example, contrary to other non aspect-oriented approaches like [17], we do not offer more capabilities for expressing the variability of concerns than the one provided by the underlying metamodel used for the concern specification. The approach called SmartAdapters had been applied first to Java programs [8] and more recently to EMF models. It leverages the notions of subject [14] and aspect programming [6,7]. Its key concepts are **concerns**, **adapters**, **adaptations** and **adaptation target**. The main idea is the following: each concern identified as reusable should go with an adapter which specifies a **composition protocol**, that is a set of adaptations and adaptation targets describing how the concern should be composed with other concerns when it is reused. This protocol will guide the designer to identify the specific parts for reuse when composing a reusable concern into a target concern.

We propose to explain this approach through the reuse of the *Observer* design pattern. First we define its composition protocol (see Figure 4). For better readability, we use a concrete textual syntax in order to specify this composition protocol. Details in the concrete syntax are not important and the syntax might be slightly modified in the future.

```
01 concern designpattern.observer
02 abstract adapter ObserverAdapter {
03
04    abstract Class target "class(es) representing an observer" : observerClass
05    abstract Class target "class(es) representing a subject " : subjectClass
06    abstract Method target " method(s) notifying  changes " : notifyingMethod
07      require notifyingMethod in subjectClass.*
08
09    adaptation becomeObserver "Modify class to make it an observer" :
10       inherit Observer in observerClass
11
12    adaptation becomeSubject "Modify class in order to make it  a subject " :
13       merge class subjectClass  with Subject
14
15    adaptation  introduceLink "introduce an association (subject to observer)  " :
16          introduce Association observers (subjectClass  -¿ observerClass)
17
18    adaptation notifyingObserver
19       " Alter notifyingMethods to tell observers about modification" :
20       extend method notifyingMethod( ... ) with after { changeValue(); }
21
22     abstract adaptation updateObserver "add an update facility  to observers " :
23       introduce method public void update(subjectClass s) in Observer
24
25       ... Protocol includes also :object initialization,observers registration,...
26 }
```

Fig. 4. Snippet of the composition protocol for the Observer design pattern

Let us now detail this example illustrated in Figure 4. Line *01* specifies the concern to be reused. The adapter called *ObserverAdapter* describes its composition protocol (Line *02*). When the composition protocol is defined the concern(s) that may reuse it are not known so that we do not know the classes corresponding to the objects acting as *subjects* and those acting as *observers*. The only thing that we may assume is that there are classes that act as observers and subjects. They are represented by the two abstract targets of type **class**: *observerClass* and *subjectClass* (Lines *04* and *05*). Each of these targets may be associated to one or several classes at composition time.

Considering the design pattern *Observer* of Figure 3, any *subject* must inform an *observer* that its content has been modified by calling the method *changeValue*. For the same reasons that the classes mentioned above are not known the method(s) playing this role are also not known but they should exist and be declared in the *subjectClass* (Lines *06* and *07*). To ensure that the call to *changeValue* is performed by the method(s) *notifyingMethod*, the composition protocol specifies an adaptation of type ***interception*** which adds this call at the end of the corresponding method(s) (Lines *18* to *20*).

More generally this kind of adaptation deals with some actions to be taken when a classifier member (attributes, methods...) is accessed or called. These adaptations allow the designer to add behavior at the beginning, end or around some existing methods but also to add some treatment when an exception is triggered. For attributes, interception may occur when the attribute is read or modified.

Let us continue with our example. To be able to call *changeValue* or any other feature of class *Subject*, it is necessary to have access to it from within the classes corresponding to *subjectClass*. This means that we have to specify another adaptation. Two possibilities could be chosen: to merge all the features

of class *Subject* into *subjectClass*, or to make *subjectClass* inherit from *Subject*. Here we choose an adaptation of type ***Merging*** (Lines *12* and *13*).

Such adaptations deal mainly with packages, classifiers features and associations. Method merging is particularly interesting if there is a support for describing the behavior (programming constructs in Kermeta [13], Sequence diagrams in UML, etc.). At present time merging policies are mainly execution of one method before the other; the handling of interlaced method bodies could be inspired by approaches like [9]. Merging classifiers is either straightforward (no conflict, name of features to be merged are identical, feature appear only in one of the classes,...) or may need more information in order to relate the features of the classifiers that need to be merged [2,16].

All these adaptations were dealing with the *subjects*. It is then necessary to address *observers* and to also insert class *Observer* at the right place(s) in the target concern. We chose here to inherit from it (Lines *09* and *10*). Such adaptation is of type ***Introduction***. It deals not only with superclass introduction as it is the case here but also with adding classifier members (new attributes or methods), as well as association. It is also possible to add a classifier invariant or a method assertion (Precondition or postcondition).

We use the same type of adaptation to insert the association between *subject* and *observer* classes as specified in the design pattern. Depending on the association to be introduced we may provide additional information. For example, in the current case the association is unidirectional from *subjects* to *observers* (Lines *15* and *16*).

It only remains one thing to do: to add to the *observerClass* class(es) a method *update* (also an adaptation of type ***Introduction***), that reacts to the changes made in the subject object. At this time we do not know the content of this feature because we do not know what the purpose of the target concern is. This is why the adaptation is **abstract**. The advantage to plan this adaptation in the composition protocol is to guide and control the reuse of the design pattern.

This composition protocol continues with the description of the initialization and the registration of observers but for space reasons we do not include it.

Let us suppose now that this concern is reused by the concern described in Figure 1 (Section 2) dealing with mobile phones. So we need to compose these two concerns. The information which is imcomplete into the composition protocol (abstract targets and adaptations) is described into a concrete adapter *ApplicationPhone* which specializes the adapter *ObserverAdapter* as it is shown in Figure 5. Please note that, in this example, the insertion is *in situ*. It means that adaptations are performed within the concern *application.phone*. In some cases, it is better to make the composition *ex situ* that is to say to compose the two concerns into a new one.

In the above composition protocol (Figure 4) we made several assumptions about the target concern. For example, we suppose that the association does not yet exist between the classes *GUI* (the observer) and *Clock* (the subject). This is a drawback because if the composition does not deal with a concern which satisfies these assumptions, it will be impossible to reuse the composition

```
01 concern application.phone
02 compose designpattern.observer with application.phone
03 adapter ApplicationPhone extends ObserverAdapter {
04
05    target typeOfValue = Time
06    target subjectClass = application.phone.Clock
07    target observerClass = application.phone.GUI
08    target notifyingMethod = application.phone.Clock.incrementTime()
11
12    adaptation observerUpdate :
13      introduce method public void update (subjectClass s) in observerClass {
14         setTime(time++)
14         displayTime()
15      }
16 }
```

Fig. 5. Reuse of Design Pattern Observer for a mobile phone

protocol in another context. Thus, we reach the conclusion that we need to introduce some variability within the composition protocol. This is the purpose of section 4.

4 Extension to Support Variability

In Section 3 we proposed an overview of the SmartAdapters approach. We now consider the needs of variability pointed out in Section 2. Our objective in this section is to introduce matching and adaptation variability into the composition protocol in order to make it more reusable and as consequence to make the concern itself more reusable. SmartAdapters is a support for explaining our approach but we plan to address other AOM approaches. Variability mechanisms introduced are inspired by Software Product Lines approaches, especially [17].

Figure 6 shows what we should introduce in an adapter to better customize the composition protocol. In Section 5 we will describe the metamodel containing the capabilities that are suggested here.

We may note first that adapter *ObserverAdapter* is now preceded by the keyword **derivable** (Line *02*). This means that it may present several alternatives to implement the composition and may consider some adaptation targets or adaptations as optional. This adapter acts as a template where some information should be given in order to choose between possible variants or options.

A first possible customization is dealing with the insertion of the features provided by classes *Subject* and *Observer*. Depending on the target concern or more generally on the context of reuse, it may be interesting to have the choice between inheriting from those classes or merging their features into *observerClass* and *subjectClass*. In Figure 4 a choice is made *a priori*. In Figure 6, the choice is described by the Lines *09* to *24* through a clause **Alternative** *InsertionChoices* which specifies here two variants (more variants could be defined if needed). A variant may contain several adaptation target declarations and adaptations. Implicitly this means that these targets and adaptations are dependent from each others.

Now, we can introduce the *update* method. If we merge the *Subject* and the *Observer*, we need to introduce the *update* method in the class where the *Observer* is merged i.e., *observerClass* (Lines *22* and *23*). *Subject* is also merged in a target

```
01 concern designpattern.observer
02 derivable adapter ObserverAdapter {
03
04     abstract Class target ‘‘class(es) representing an observer’’ : observerClass
05     abstract Class target ‘‘class(es) representing a subject ’’ : subjectClass
06     abstract Method target ‘‘ method(s) notifying changes ’’ : notifyingMethod
07       require notifyingMethod in subjectClass.*
08
09     Alternative InsertionChoices ‘‘ Choice between inheritance and merging’’ {
10       [Vinheritance] ‘‘ Inheritance variant ’’ :
11       adaptation becomeSubject ‘‘Modify class in order to make it a subject ’’ :
12          inherit class Subject  in subjectClass
13       adaptation becomeObserver ‘‘Modify class to make it an observer’’ :
14          inherit Observer in observerClass
15       abstract adaptation updateObserver ‘‘add an update facility  to observers ’’ :
16          introduce method public void update(Subject s) in Observer
17     or else [Vmerge] ‘‘ Merging variant ’’ :
18       adaptation becomeSubject ‘‘ Modify class in order to make it a subject  ’’ :
19          merge class subjectClass  with Subject
20       adaptation becomeObserver ‘‘Modify class to make it an observer’’ :
21          merge  class observerClass with Observer
22       abstract adaptation updateObserver ‘‘add an update facility  to observers ’’ :
23          introduce method public void update(subjectClass s) in observerClass
24      }
25
26     Alternative NotificationTime ‘‘ Choice of notification time’’  {
27       [Vbegin] ‘‘ Method beginning variant’’ :
28       adaptation notifyingObserver
29         ‘‘ Alter notifyingMethods to tell observers about modification’’ :
30         extend method notifyingMethod( ... ) with before { changeValue(); }
31     or else [Vend] ‘‘ Method ending variant’’ :
32       adaptation notifyingObserver
33         ‘‘ Alter notifyingMethods to tell observers about modification’’ :
34         extend method notifyingMethod( ... ) with after { changeValue(); }
35      }
36           ...Protocol  includes also :object initialization,observers registration,...
37 }
```

Fig. 6. Composition protocol for the Observer with variability

class, therefore the parameter of the *update* method has the type of this target class i.e. *subjectClass*. If the pattern is composed by inheritance, the *update* method is introduced in the *Observer* class itself, and the parameter has the type *Subject* (Lines *15* and *16*). The *update* method is very related to the composition variant, so we integrate its introduction in the InsertionChoices alternative. Depending on the chosen composition variant, the right *update* method will be introduced. In both cases the contents of this method is not already known, that is why this method is **abstract**.

A second possible customization is related to the location of the call to method *changeValue* within *notifyingMethod*. It may be useful depending on the target concern to notify the subject changes to observers either at the beginning or at the end of the execution of *notifyingMethod*. The corresponding variants are described by the Lines *22* to *31* through a second clause **Alternative**. Each variant corresponds to a unique adaptation of type Interception.

In figure 7 we extend this protocol to experiment the combination of **optional** and **constraint** clauses. We now address the association between observers and subjects (called *observers* in the design pattern of Figure 2). It is very likely that

```
01 concern designpattern.observer
02 derivable adapter ObserverAdapter  {
03   ...
04
05   is optional AssociationExist  `` association (observers to subject)  may exist '' {
06        abstract  Association target `` handling association mapping'' :
07           subjectObserverAssociation
08        adaptation  mergeLink ``merge association with the Observer pattern one   '' :
09           merge association subjectObserverAssociation with observers
10           require subjectObserverAssociation ⊂  (subjectClass  -> observerClass)
11     }
12
13   is optional LinkModification1 ``  Existing association may be renamed ''   {
14        abstract adaptation  renameLink ``rename association-end of association   '' :
15           rename association  subjectObserverAssociation
16   is optional LinkModification2 ``  Existing association may be redefined '' :
17        adaptation  alterLink ``add an association-end to association   '' :
18           add association observers  (subjectClass  -> observerClass)
19          ...
20   constraint AssociationHandling ``working on association implies it exists ''   {
21      LinkModification1 depends on {AssociationExist}
22      LinkModification2 depends on {AssociationExist}
23      {LinkModification1, LinkModification2} are exclusive
24    }
25 }
```

Fig. 7. Options and matching variability

depending on the target concern this association may already exist in it. In order to authorize both situations we propose some optional adaptations (Lines *05* to *18*). A first optional clause assumes that the association exists in the target concern and is identified by the target *subjectObserverAssociation*; it must be merged with *observers*. Then it may be possible to specify a renaming adaptation because nothing can ensure that it has the same association-end name in the target concern. It is also possible to add an association-end when the association exists but in the opposite way in this concern.

The example developed in Figures 6 and 7 especially illustrates the needs for optional parts and variant definitions. In order to insure the consistency of the composition protocol, the user can define mutual exclusion and dependency constraints. These constraints restrict the number of possible combinations to sensible ones. In our example, we want to ensure that *i)* renaming and redefinition may not be performed if the association between observers and subjects does not exists in the target concern and, *ii)* renaming its association-end is incompatible with adding *observers*. These contraints are expressed (Figure 7 - Lines *20* to *23*) by introducing dependencies between *L*inkModification1, *L*inkModification2 and *A*ssociationExists options and a mutual exclusion between the first two options.

Now, we can compose the variable "Design Pattern Observer" into the mobile phone base model. In addition to the tasks described in figure 5 it is necessary to select options and variants (adaptation targets and adaptations) which are suitable for the concern "mobile phone". Of course the abstract adaptation targets and adaptations to concretize in the adapter *ApplicationPhone* depends on the variants and options which are selected (Figure 8).

The selection is made through a clause **derive** (Lines *05* to *08*). No association can match the *observers* association in the target model, so the optional clauses

```
01 concern application.phone
02 compose designpattern.observer with application.phone
03 adapter ApplicationPhone derives ObserverAdapter {
04
05    derive designpattern.observer with {
06       options: none
07       alternatives: InsertionChoices#[Vinheritance], NotificationTime#[Vend]
08    }
09
10    target typeOfValue = Time
11    target subjectClass = application.phone.Clock
12    target observerClass = application.phone.GUI
13    target notifyingMethod = application.phone.Clock.incrementTime()
14
15    adaptation observerUpdate :
16      introduce method public void update (Subject s) in observerClass {
17         setTime(time++)
18         displayTime()
19      }
20 }
```

Fig. 8. Reuse of Design Pattern Observer for a mobile phone

are not selected (note that an association exists in application.phone but in the opposite way so that it would be possible to keep only one association selecting *AssociationExist* and *LinkModification2*). We also select the two variants associated to the alternative clauses *InsertionChoices* and *NotificationTime*. Finally, we have to concretize the update method, specifying that the GUI has to increment its variable *time* and refresh the screen. Concretizing abstract methods in a concrete adapter is close to the mechanism defined in the AOP approach of Hannemann *et al.* [5]. Mandatory targets and adaptations of Figure 6 are processed normally in the same way as it is done in Figure 5.

Figure 9 shows two types of composition i.e, merging and inheritance, in order to realize the **display time** requirement. **Inheritance** corresponds to the adapter we have derived above, while **Merging** corresponds to another possible derivation provided by the protocol.

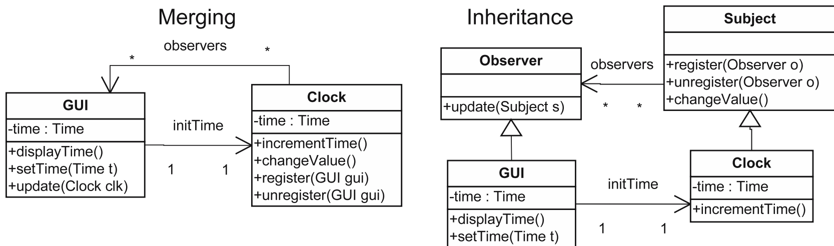

Fig. 9. Two possible compositions of the Observer pattern

In the motivating example, we were not able to realize the **total outgoing calls** with the standard Counter pattern because the template approach was not flexible enough. We can now realize the **total calls** and the **total outgoing calls** requirements using the same Counter pattern. Indeed, the Counter pattern now can be applied either if the class acting as *Counter* is present or not in the base model. For

space limitation, the derivable adapter and the concrete adapter are not shown but the principle is similar to the Observer protocol (Figures 6, 7 and 8).

Finally, it is interesting to note that introducing variability did not affect the guidance and the controls when reusing a derivable concern. On the contrary, the choices induced by the addition of variability is also controlled and guided thanks to the expressiveness of the composition protocol.

5 Metamodeling and Implementing AOM with Variability

This section proposes a metamodel of concerns that includes concepts for adapters and variability illustrated in sections 3 and 4. This metamodel aims at giving a precise formulation of concerns and make it possible their integration into modeling tools. Figure 10 shows an excerpt of the metamodel where concepts introduced to handle variability are identified with a circle at the upper left. The key concepts of the metamodel are concern, adapter, target and adaptation.

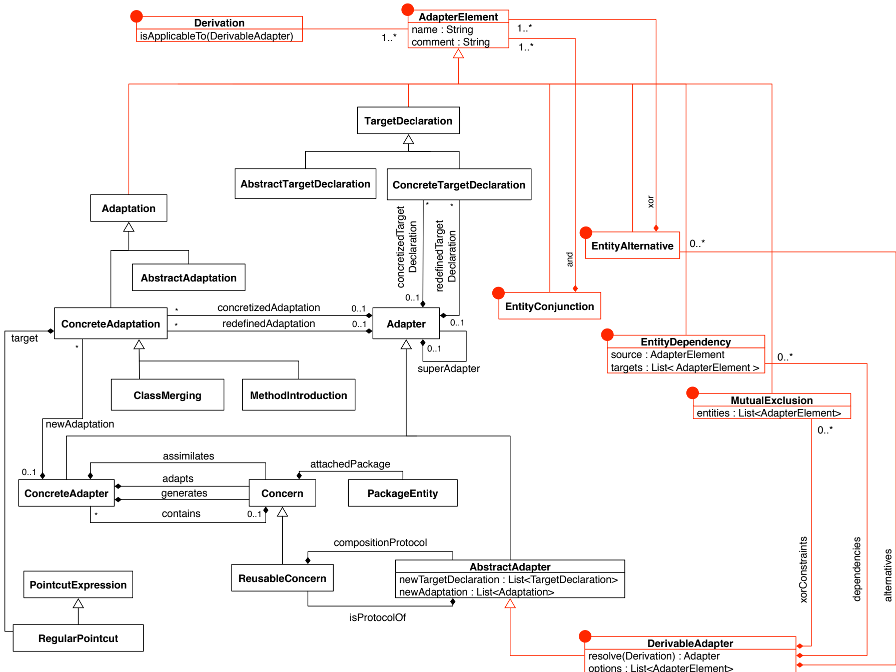

Fig. 10. Metamodel of SmartAdapter with variability

A reusable concern (*class ReusableConcern*) is associated to a package (*class PackageEntity*) which contains the concern description[3] and its protocol of composition (*class AbstractAdapter*). Concerns are not always reusable (*class Concern*).

[3] We assume that a concern is described by a package of classes similarly to a UML class diagram.

For example the concern which describes the GUI of an application is rather specific and may not be reusable; such concerns do not have a composition protocol but could be composed with other concerns. A concern refers to as many concrete adapters (*ConcreteAdapter*) as there are concern to be integrated with it.

An adapter (class *Adapter*) is identified by a name and may inherit (i.e. specialize) from another adapter. An adapter may be abstract (class *AbstractAdapter*), concrete (class *ConcreteAdapter*) or derivable (class *DerivableAdapter*). Each adapter contains adaptations (class *Adaptation*) and adaptation targets (class *TargetDeclaration*). A derivable adapter is an abstract adapter which supports variability: it allows the designer to customize the set of adaptations or/and the set of adaptation targets by expressing options, alternatives, dependencies and exclusions. Such an adapter is not intended to be used directly for composing concerns but serves to derive an adapter. A derived adapter is obtained using the method *resolve* of class *DerivableAdapter* which takes a derivation (class *Derivation*) parameter to select the adaptations and the adaptation targets among the options and variants. This adapter may be concrete, abstract or derivable depending on what is resolved by the derivation parameter.

A target declaration (class *TargetDeclaration*) specifies an adaptation target that matches the entities on which the adaptations relies on. An adaptation target may identify just one required element (class *AbstractTargetDeclaration*) (like the *observers* or the *subjects* in the design pattern *Observer*) or be fully specified (class *ConcreteTargetDeclaration*) by referencing the real element (class, method, ...) to adapt.

An adaptation (class *Adaptation*) specifies the action to be taken for an element of the reusable concern when it is composed. The metamodel includes a hierarchy of adaptation classes that are typed according to the types of target entities (package, classifier, method, attributes and association) and reflect the four kinds of adaptation currently proposed: interceptions, introduction, merging and redefinitions. Figure 10 shows two of the adaptation classes used in the previous examples (class *ClassMerging* and class *MethodIntroduction*).

To be able to take into account several variants for the integration of the concern, the metamodel includes the concept of alternative entity (*EntityAlternative*). An alternative entity may refer to several adaptations or adaptation targets (see *xor* link) but only one will be selected at composition time.

Adaptations, adaptation targets and even alternatives can be optional in a derivable adapter, that is to say that they are planned in the composition protocol but they could be retained or not when the concern is composed with another one. Optional elements of a derivable adapter are referenced by its *options* link.

Practically several adaptations or adaptations targets may be described in a given variant or be declared as an optional block. For this purpose we propose a way to group thoses entities (class *EntityConjunction*).

In a derivable adapter, classes *EntityDependency* and *MutualExclusion* allows designers to specify that an *AdapterElement* (variant or option) may not be selected with other ones or on the contrary must be selected if some others are selected. These classes define constraints that are checked before deriving a derivable

adapter, in order to insure the consistency of the derived adapter. If a derivation does not respect these constraints then an exception is raised that asks the user to modify the derivation.

The metamodel described above has been used to build a modeling tool integrated in the Eclipse environment. This tool currently provides two main functionalities: designing models of concerns and adapters; composing concerns from their models. This tool has been implemented using the Eclipse Modeling Framework (EMF) and the Kermeta language [13]. We have exploited EMF to define a Ecore version of our metamodel, reusing the Ecore metamodel for the description of concerns. The Kermeta language has been exploited to extend the Ecore version of our metamodel with operational behavior. This behavior performs several tasks related to the design and composition of concerns: it checks the consistency of adapters, computes derived adapters and compose elements of concerns from a set of adaptations. At this time, we are investigating the design of a concrete textual syntax for our metamodel like the one used in the previous section and we plan to build the concrete syntax tool using a meta-model centric approach as [12].

6 Related Work

There exists numerous AOM approaches but few of them support variability mechanisms at the composition level [4,16,1]. In [2], Clarke *et al.* model an aspect in a template package specifying the structure and the behavior of the aspect with a class diagram and sequence diagrams. The template is composed of model elements present in the concern's class diagram and specifies the elements that have to be matched in the target model. There is no functional or matching variability mechanism. The composition relationship authorizes multiple bindings i.e. it is possible to match several target model elements to the same concern model element. Adaptation lacks variability: concerns are indeed always merged into the target model. Note that it is possible to generate AspectJ code to postpone the weaving at code level. Our adaptation protocol allows the designer to define different variants of how the concern will be integrated in the target model. All the variability mechanisms we have identified may be adapted to Theme.

Muller *et al.* [11] also propose an approach to compose a reusable model expressed as a template package with an existing model. To express this composition, they introduce an apply operator that specifies the mapping between template parameters and elements of the target model. Their approach addresses variability at the composition level by giving the capacity to annotate the apply operator with different strategies such as "merge" or "view". Strategies are only provided to get different resulting models. Compared to our proposal, this solution does not offer any mechanism to express options and variants for the reusable model. It is also less flexible as it does not offer finer grain mechanisms to control how elements of reusable and target models must be composed.

France *et al.* [16] have developed a systematic approach for composing class diagrams in which a default composition procedure based on name matching can be customized by user-defined composition directives. These directives constrain

how class diagrams are composed. The framework automatically identifies conflicts between models that have to be composed and it solves them thanks to the composition directives. Contrary to Theme, composition directives address the weaving only from the structural point of view. They consider the composition as a model transformation. The variability can be addressed by designing several composition directives depending on the integration context. However, the definition of the composition directive would then become messy and error-prone. Besides, it is a symmetric AOM approach in which they do not differentiate between aspect model and base model. Consequently, they do not currently provide a pointcut language to manage the composition.

In [5], Hannemann *et al.* propose an AOP approach to implement design patterns with AspectJ. They propose up to seven different implementations for each design pattern. The only variability mechanism is the generalization relationship between an abstract aspect and an aspect. For example, the *update* method of the *Observer* is declared abstract in an abstract aspect and its contents will be specified in a concrete aspect. We also use this mechanism but the variability mechanisms we introduced allow a concern to be applied in multiple contexts whereas we would have to create a new aspect depending on the context with the Hannemann *et al.* approach. Option and variant notions do not exist, reducing the reusability of the aspects. Our concerns are adaptable and do not need modifications to be applied, but only customization. Introducing the same variability mechanisms at the code level code could enhance the expressiveness of AOP language such as AspectJ.

7 Conclusion

In this work, we propose an approach for introducing variability in aspect-oriented modeling (AOM). To achieve this goal, two important parts of such an AOM approach were needed: A concern model and a weaver that support variability. In this paper we mainly focus on the second one. Indeed, the variability in the concern specification depends on the expressiveness of the meta-model dedicated to concern modeling. Consequently, a reasonable solution to integrate variability in the concern model can be inspired by product lines researches and more precisely by [17].

To introduce variability in the weaving process, the composition meta-model of our AOM approach has been extended. These extensions concern the adaptations primitives and the pointcut specification. They are composed of a set of entities specifying optional parts, alternatives, dependencies and mutual exclusion constraints. These extensions allow the user to design a family of aspects at the design level that can be derived to be applied in a particular context.

One of the main benefits of building a composition protocol is the capability to control and guide the software architect when he designs new applications. The variability introduction does not affect the guidance and the control when reusing a derivable concern. On the contrary, the choices induced by the addition of variability are also controlled and guided thanks to the expressiveness of the composition protocol.

In the SmartAdapters platform, we plan to improve the pointcut language and the target identification. One possible solution is to describe the pointcut with a template model and to use pattern matching [15] to identify targets. We also want to generalize the SmartAdapters to various metamodels, not only class diagrams or Java programs. In [10], we have proposed and implemented a metamodel-driven approach to generate domain-specific AOM frameworks that uses the aforementioned pointcut language. Finally, AOM approaches can be used to manage variability in software product line. Our work can be merged to these approaches to show why variability is also needed in the aspects in order to use an AO approach to build software product line.

References

1. Aldawud, O., Elrad, T., Bader, A.: UML Profile for Aspect-Oriented Software Development. In: 3rd International Workshop on Aspect Oriented Modeling (In conjunction of AOSD'03), Boston, Massachusetts (March 2003)
2. Baniassad, E., Clarke, S.: Theme: An Approach for Aspect-Oriented Analysis and Design. In: ICSE '04. Proceedings of the 26th International Conference on Software Engineering, pp. 158–167. IEEE Computer Society, Washington, DC, USA (2004)
3. Barais, O., Le Meur, A.F., Duchien, L., Lawall, J.: Safe integration of new concerns in a software architecture. In: ECBS '06. Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, pp. 52–64. IEEE Computer Society, Washington, DC, USA (2006)
4. Elrad, T., Aldawud, O., Bader, A.: Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 189–201. Springer, Heidelberg (2002)
5. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and Aspectj. In: OOPSLA '02. Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 161–173. ACM Press, New York, NY, USA (2002)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of Aspectj. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
7. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
8. Lahire, Ph., Quintian, L.: New Perspective To Improve Reusability in Object-Oriented Languages. Journal Of Object Technology (JOT) 5(1), 117–138 (2006)
9. Mens, T., Kniesel, G., Runge, O.: Transformation Dependancy Analysis, a Comparison of two Approaches. In: Rousseau, R., Urtado, C., Vauttier, S. (eds.) Proceedings of LMO 2006, Langages et Modèles à Objets, Nîmes, France, pp. 167–182. Hermes-Lavoisier (Mars 2006)
10. Morin, B., Barais, O., Jézéquel, J.M., Ramos, R.: Towards a Generic Aspect-Oriented Modeling Framework. In: 3rd International Workshop on Models and Aspects (In conjunction of ECOOP'07), Berlin, Germany (2007)
11. Muller, A., Caron, O., Carré, B., Vanwormhoudt, G.: On Some Properties of Parameterized Model Applications. In: Proceedings of ECMDA'05: First European Conference on Model Driven Architecture - Foundations and Applications, Nuremberg, Germany (November 2005)

12. Muller, P.A, Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M: Model-driven analysis and synthesis of concrete syntax. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 98–110. Springer, Heidelberg (2006)
13. Muller, P.A., Fleurey, F., Jezequel, J.M.: Weaving Executability into Object-Oriented Meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, Springer, Heidelberg (2005)
14. Ossher, H., Tarr, P.: Hyper/J: Multi-Dimentionnal Separation of Concern for Java. In: Ghezzy, C. (ed.) Proceedings of ICSE'00, Limerick, Ireland, ACM Press, New York (2000)
15. Ramos, R., Barais, O., Jézéquel, J.M.: Matching model-snippets. In: MoDELS '07. Model Driven Engineering Languages and Systems, 10th International Conference, Nashville, Tennessee (2007)
16. Reddy, Y.R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., McEachen, N., Song, E., Georg, G.: Directives for Composing Aspect-Oriented Design Class Models. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, pp. 75–105. Springer, Heidelberg (2006)
17. Ziadi, T., Jézéquel, J.M.: Families Research Book. In: Product Line Engineering with the UML: Products Derivation. LNCS, pp. 557–588. Springer, Heidelberg (2006)