New Perspective To Improve Reusability in Object-Oriented Languages

Philippe Lahire, Laurent Quintian

JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 2, No. 2, March–Aprile 2004

New Perspective To Improve Reusability in Object-Oriented Languages

Philippe Lahire, **Laurent Quintian**, I3S Laboratory, University of Nice-Sophia Antipolis and CNRS, France

Object-oriented languages provide insufficient answers regarding reuse of hierarchies of classes especially because mechanisms provided for separating application concerns are not sufficient. We propose to extend object-oriented languages, Java in the current implementation, to address this particular issue. The model, inspired by approaches dedicated to the separation of concerns, introduces a new concept called *adapter*. It enables to specify the *composition protocol* of a hierarchy of classes independently from the context of use. This composition protocol allows the programmer to benefit from the necessary guidance and controls when the adapter is customized to be integrated into applications.

1 INTRODUCTION

The reuse of the code written for one application to develop new ones is essential to be able to react quickly to the market needs but most of the time it is necessary to adapt the existing code in order to reuse it. Reusability represents one of the most important challenges of object-oriented languages [13] but we have to admit that, whatever the skills of existing programming languages are, this objective is far from being reached, in particular because the adaptation mechanisms (based on feature renaming or redefinition) are not powerful enough.

One of the keys of software reusability is to separate the various concerns of a software and to make them as much independent as possible from the future contexts of use. Depending on the nature of the concern which is addressed, it may be represented by a set of features (attribute or method), a set of statements, one class or a set of classes, one or several hierarchies of classes, etc. From a general point of view, a concern represents one facet of the problematic addressed by an application.

A concern is known as functional, non-functional or hybrid. Let us give some more explanation about this classification of concerns.

Generally an application is composed of several facets like:

- the extraction or storage of application data,
- the control of data consistency,

- the evaluation or computation of data,
- their display through the use of either a textual or graphical interface.

Each of these facets corresponds to a set of functionnalities which implements the associated objective. Such a piece of software is called a functional concern. For example, libraries of classes provided by object-oriented languages and dedicated to one purpose may correspond to a concern.

When applications need to extend existing functionnalities with additional processing (often called *services*), like persistence handling, distribution of data, support of various forms of security or method-call tracing [10], then it is a non-functional concern. In Section 2 we will address a hybrid concern which extend existing functionnalities but also add new ones.

When a concern is identified and described, whether functional or non-functional, an important issue is to get a sufficient expressiveness to recompose those concerns so that applications may be specified. Two different situations may occur: the contents of the concern may be located in a single location (for example within a class or a hierarchy of classes) or it has to be spread out into the system of classes.

If we focus on highly flexible applications, which are more and more numerous with the development of ubiquitous computing, we observe that the means provided by the object-oriented languages are insufficient. The main reason is that to implement separation and composition of hierarchies of classes, object-oriented languages mainly rely on various concepts of *class*, possibly generic, and on both *inheritance* and *aggregation* relationships. But the experiments that may be found in the litterature show that it leads to costly feature redefinitions and to a large duplication of code. In some situations object-oriented languages do not even offer any realistic solution for the composition of concerns. This is typically the case when a concern is spread out into the application or when the number of concerns to be composed evolves constantly and makes the resulting hierarchy too complex very quickly.

Following these observations other paradigms emerged to address the needs of modern applications. Among others we may mention the separation of concerns [12], programming by components [21] or programming by models [5]. We think that object-oriented languages should accept the challenge and provide mechanisms supporting, in some form, the separation of concerns. These mechanisms should fit nicely with object-oriented concepts, in particular reuse, and applications using those facilities should be as robust and reliable as applications built upon pure object-oriented mechanisms. To go in this direction, we propose a generic approach (independent of existing object-oriented languages), to extend the object-oriented languages with a set of adaptation operators applying on classifier diagrams[1].

This approach respects the following hypotheses: *i)* describing and separating the various concerns relying strongly on the object-oriented language which is chosen to program the application, *ii)* enabling the specification of a composition protocol

[1]This is the term used in UML to deal with the various kinds of class.

which is documented and comprehensive enough to guide and control the reuse of each reusable concern.

Our model is inspired by the existing approaches dedicated to the separation of concerns (ASoC - *Advanced Separation of Concerns*). Approaches for the separation of concerns include aspect-oriented programming (AOP) [12], subject-oriented programming (SOP) [17], role or point-of-view oriented programming [9], composition filters [1] or, to a lower extent, mixins [2] and metaprogramming [4]. The state of the art and the detailed study proposed in [20] show that the languages offering the most interesting expressiveness are AspectJ [11, 10] (AOP) and Hyper/J [18] (SOP).

Section 2 presents our choices for the description of concerns and sets the bases of an example which is particularly representative and on which the remaining parts of this document rely. Section 3 provides an overview of both the composition model and the type of adaptations supported in our approach. Section 4 relies on the previous sections to describe how to equip a concern in order to improve its reusability. Section 5 provides some more details about the semantics of adaptations. Section 6 gives an overview of the implementation and addresses the state of the art. In particular it compares the contribution of our approach to AOP and SOP. The last section draws conclusions and describes some future works.

2 DESCRIBING APPLICATION CONCERNS

A concern is made up of classifiers (*class*, *interface*...) or hierarchies of classifiers and it is reasonnable to assume that it is encapsulated in one container (a *package* for example) which may possibly contains itself other sub containers.

In the following example we use the Java syntax in order to define the two concerns that we consider but the composition model relies on a reification of object-oriented languages and not on a given syntax.

We now suggest to consider two possible concerns of an application: the graphical user interface (from now referred to as GUI) and the design pattern `Observer` [7]. In this section we describe each of these concerns independently from each other. Section 4 benefits from the information given in Section 3 about both the supported adaptations and the modelling of the composition protocol. It explains how to specify the composition protocol of the concern `Observer` and shows how to implement and control its integration within the concern GUI.

Concern *Design Pattern* `Observer`

The design pattern involves the classifiers `Observer`, `Observable` and `ImplObservable`[2] (see Figure 1); they are independent from any application. The reader may notice in particular that the concern is encapsulated within the Java package

[2]Gamma prefers `Subject` and `ConcreteSubject` instead of `Observable` and `ImplObservable`.

```
01 package designpattern.observer;
02 public interface Observable {
03   public void addObserver(Observer o);
04   public void removeObserver(Observer o);
05   public void notifyObservers();
06 }
07 package designpattern.observer;
08 public interface Observer {
09   public void updateObserver(Observable o);
10 }
11 package designpattern.observer;
12 import java.util.ArrayList;
13 import java.util.Iterator;
14 import java.util.List;
15 public class ImplObservable implements Observable {
16   protected List observers = new ArrayList();
17   public void addObserver(Observer o) {
18      observers.add(o);
19   }
20   public void removeObserver(Observer o) {
21      observers.remove(o);
22   }
23   public void notifyObservers() {
24      for (Iterator iter = observers.iterator(); iter.hasNext();)
25        ((Observer) iter.next()).updateObserver(this);
26   }
27 }
```

Figure 1: Design pattern `Observer`

`designpattern.observer`. The interfaces `Observer` and `Observable` correspond to the two roles of the pattern. The class `ImplObservable` is a staightforward implementation of the interface `Observable`. It relies on the classes `ArrayList`, `List` and `Iterator` which are defined in the packages of the Java library (`java.util`).

At this stage, the developer may only notice that *i)* any class implementing the interface `Observer` must specify a body for the method `updateObserver` and *ii)* both interfaces are strongly coupled because each of them requires the other in the signature of their methods. To easily reuse this concern, and more generally all concerns, it is mandatory to add a more comprehensive documentation: *the composition protocol* which will be addressed in Section 4.

Concern *Graphic User Interface*

The concern in which we want to integrate the design pattern `Observer` deals with the implementation of GUI applications. It is obvious that the proposed GUI is very basic and fully dedicated to the example. But it is simple to understand and complex enough to explain our approach and to show its contribution.

The GUI concern is encapsulated in the Java package `application.ihm` (Figure 2). It describes two types of graphical objects: buttons (class `Button`) and labels (class `Label`). These two classes are based on classes from the Java `swing` library, `JButton` and `JLabel`. The redefinition of the method `fireActionPerformed` within

```
01 package application.ihm;
02 import java.awt.event.ActionEvent;
03 import javax.swing.JButton;
04 public class Button extends JButton {
05    public Button(String text) {
06        super(text);
07      }
08    protected void fireActionPerformed(ActionEvent event) {
09        super.fireActionPerformed(event);
10      }
11   }
12 package application.ihm;
13 import javax.swing.JLabel;
14 public class Label extends JLabel {
15    public Label(String text) {
16        super(text);
17      }
18   }
19 package application.ihm;
20 import java.awt.*;
21 import java.awt.event.*;
22 import javax.swing.*;
23 public class ApplicationInterface extends JFrame {
24    public ApplicationInterface() {
25        super("My application interface");
26        getContentPane().setLayout( new GridLayout(1,3));
27        Button button1 = new Button(" Click on me !");
28        getContentPane().add(button1);
29        Label label1 = new Label("- no text -");
30        getContentPane().add(label1);
31        JButton finish = new JButton(" Exit ");
32        getContentPane().add(finish);
33        finish.addActionListener(new ExitListener());
34        pack(); show();
35      }
36    public static void main (String args[]) { new ApplicationInterface(); }
37    public class ExitListener implements ActionListener {
38        public void actionPerformed(ActionEvent arg0) {System.exit(0);}
39    }
40 }
```

Figure 2: Description of the GUI concern

the class `Button` is not mandatory. Its only interest is to make the example more understandable.

The class `ApplicationInterface` contains the GUI initialisation and an entry point to launch the application. Figure 2 shows the full source code, but only the initialisation of the objects referenced through `button1`, and `label1` in the class constructor (lines *24* to *35*) will be used in Section 4 for the explanation dedicated to the composition of the two concerns.

3 MODELLING THE COMPOSITION OF CONCERNS

Before going further in the description of the example, it is necessary to address the composition model and the adaptations which may be applied to the concerns to weave them one another.

Composition Model

The adaptations which are involved in the implementation of the concern composition rely on concepts that belong to most languages and on one or several transformation or typing rules; thus it is reasonable to propose a common reification for the entire set of properties (of course it is possible to extend the reification at any time without breaking the model architecture). As we mentioned it earlier, a concern is encapsulated in one hierarchy of containers (*package*, *cluster*...)[3] which contains classifiers. A classifier (*class*, *abstract class*, *deferred class*, *interface*...) may inherit from one or several other classifiers. A classifier belongs to a container and it has modifiers (*abstract*, *deferred*, *final*, *frozen*, *public*...). It contains attributes (constants, instance or class variables), methods (procedure, function or constructor) which need or not the creation of class instances. The methods have a signature (name, parameters, return type...), modifiers and a body which corresponds to a set of statements. Most of the time it is not useful to consider the semantics associated with the language reification. We have to consider it only when this semantics influences the description of an adaptation.

An adaptation is located in one or several *adapters*, that is to say, outside the class to adapt; these adaptations are *non intrusive*. Figure 3 presents the main entities that enable to describe concern composition and thus the building up of an application. This composition is represented by one or several *adapters*. At the present time the mechanisms dedicated to the composition of adapters are very limitated and only guarantee that adapters which may create new concerns are handled first (further they are called *ex situ* adapters).

Each adapter has a unique name that allows to identify it. It may be concrete or abstract and it may inherit from another adapter. This inheritance mechanism is basic and allows only two of the usages of inheritance defined by B. Meyer [13] : *reification inheritance* and *functional variation inheritance*. An adapter contains *adaptation targets* that are concrete or abstract[4] depending on whether the entities on which the adaptation applies are fully specified by the declaration or not. An adaptation target is typed and deals with either a classifier, a method or an attribute. When an abstract adaptation target is declared, it is possible to set some constraints to be satisfied. A concrete adaptation target contains either an explicit list of identifiers (depending on its type, names of classifier, method or attribute), or a regular expression whose evaluation will generate the list of identifiers; its expressiveness is quite similar to the regular expressions of AspectJ [10]. An adapter also contains *adaptation operators* typed with the adaptation target. If the adapter is defined *ex situ*, then a new *container* (one *package* in current implementation) is created and the concerns related to the adaptation are inserted into it. If it is *in situ*, then the container already exists and the result of the adaptation modifies the concern associated with it.

[3] We use terminology of different programming languages (Java, Eiffel...) to highlight that our approach is not restricted to one particular language.

[4] If it contains itself an adaptation target which is abstract, then the adapter is abstract.

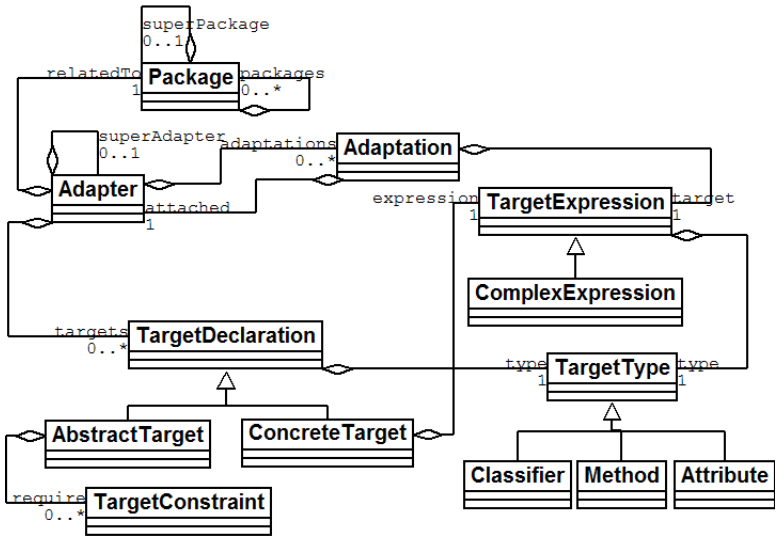


Figure 3: Reification of concern composition

Supported Types of Adaptations

Our model supports five categories of adaptation and each of them addresses several types of adaptation. This list can be extended in the future if new capabilities are found useful. Figure 4 shows that the design choices applied to the model enable to increase at any time the list of adaptations without questioning the model or the implementation architecture. The five categories of adaptation are presented in table 1. In particular it indicates if an adaptation introduces a new functionality (functional adaptation) or if it extends the description of existing functionalities in order to associate them with some new services (non-functional adaptation).

The adaptations supported by our model fit into this classification as follows (the order is the same as in table 1):

- implementation of new interfaces or insertion of a new super-class (*SuperClassifierIntroduction*);
- fusion of methods (*MethodMerging*) with several variants enabling to specify that the body of the first method is located before (*MergingBefore*), after (*MergingAfter*) or if one of the method is abstract (*MergingWithAbstract*), and fusion of classes (*ClassMerging*) with the ability, either to only add methods or attributes (*FeaturesOnlyadded*), or to merge the body of the methods (*CustomizedMerging*);
- insertion (*MethodIntroduction*) or redefinition of methods (*MethodRedefinition*);
- insertion of new instance or class variables into classes (*AttributIntroduction*);

N	Description	Type of adaptation	Target of adaptation
1	Implementation of new interfaces	functional	Class and set of classes
2	Fusion of classes and methods	functional	Class and set of classes
3	Addition and redefinition of methods within classes	functional	Class and set of classes
4	Addition of class/instance variables to classes	functional	Class and set of classes
5	Interception *before*, *after*, *around* and *on exception* of methods	non-functional	Method and set of methods
6	Interception of the accesses to class/instance variables	non-functional	Variable and set of variables

Table 1: Adaptation categories suitable for the reuse of concerns

- interception before (*BeforeInterception*), after (*AfterInterception*), around (*AroundInterception*) or when an exception is triggered on methods (*OnExceptionInterception*), whatever they apply to (instance or class);

- interception of the accesses to the instance or class variables OnGet/OnSet). In this case we also enable the integration of treatments before or after retrieving the information (*BeforeGet* and *AfterGet*) or before and after updating the variable (*BeforeSet* and *AfterSet*).

Each type of adaptation contains two methods: the first one is *check*, it controls that the constraints required by the execution are satisfied; the second one is *execute*, it describes the behaviour of the adaptation.

Section 4 will give a preview of some of the adaptations supported by our model through the continuation of the examples started in Section 2. Section 5 will give more details about the semantics of adaptations.

4 EQUIPPING A CONCERN TO BE REUSED

As a summary of what was written in the previous sections, we claim that the implementation of the approach must rely on these two key-ideas :

- Integration of concerns into the programming language used. A first experimentation uses the Java language but the previous sections show that the approach is not dedicated to a particular language.

- Having a set of adaptation operators which is expressive enough to enable the composition of functional and non-functional concerns.

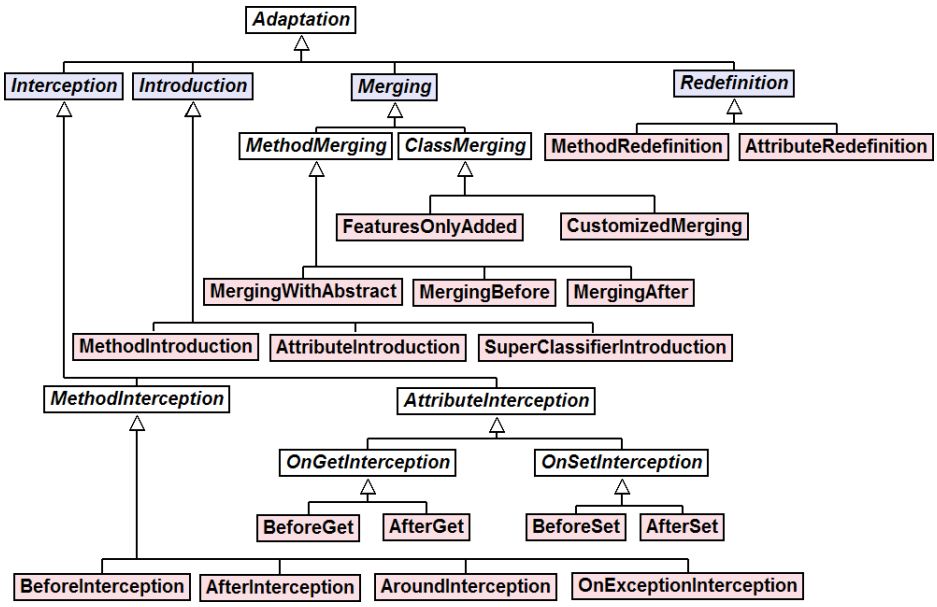


Figure 4: Reification of the adaptations which are supported

In this section we aim to demonstrate that reusability means to propose an approach for the composition of concerns which relies on the following fundamental principles:

- Being able to delay the specification of the entities on which the adaptation applies, whatever the adaptation operator is.
- Encapsulating the definition of the composition protocol (or pattern) which is dedicated to a given concern in order to make it independent from the context of the future uses.
- Making the composition protocol expressive enough to guide and control the reuse of the concern.
- Enabling the concretisation of a composition pattern to adapt it to the context of use.
- Providing two kinds of composition: if we consider two concerns to be composed, the type *in situ* composes one of the concern into the other one while the type *ex situ* creates a new concern which contains the composition of the two. The second kind of composition is particularly suitable when a concern must be composed several times in various locations of the application.

Forthcoming subsections constitute a direct continuation of the examples started in Section 2. We use the hierarchy of adaptations described in Section 3 in order to describe successively the composition protocol of the concern `Observer` and a specialization of it to take into account specificities of the concern GUI.

We decided to make only one concern reusable (the pattern `Observer`); the concern GUI is described for the specific needs of the example. This is why this paper only describes one composition protocol. But if work is done in order to make the GUI concern reusable, then it would be interesting to define a composition protocol for it.

Composition Protocol of the Concern `Observer`

As mentioned earlier, the composition protocol of the design pattern `Observer` (Figure 5) has two main objectives:

- describing the methodology associated with the use of the concern to assist the programmer during the reuse process,
- providing the necessary material to control that the programmer applies the given methodology when he adapts the composition protocol to a specific context (for example the one of the concern GUI).

The syntax proposed hereafter[5] is essential because it corresponds to what is given to the programmer of the design pattern `Observer` to specify its methodology of reuse. However, it is only a particular view of the model of which Sections 2 and 3 gave an overview. Still, we initially wanted to make the syntax as close as possible to the Java language in order to consider the entity *adapter* as a special kind of classifier [20]. But the specificity of its contents led us to prefer an *ad hoc* syntax guaranteeing a better readability and making it possible to consider the extension as a language dedicated to the adaptation of classes (*Domain Specific Language*). Only the pieces of code describing the behaviour to be inserted or modified are written with the application programming language. Currently we are thinking of some syntax refinement to better isolate the corresponding statements.

While examining the contents of Figure 5 we notice that it is independent from its future contexts of use. The line *01* indicates the concern to which the composition protocol (the design pattern `Observer`) is attached. The entity describing a composition protocol (ligne *02*) is an adapter (keyword **adapter**) and has a name (here `ObserverAdapter`). The keyword **abstract** appears before the keyword **adapter** because the adapter contains some abstract adaptations. Adaptations are introduced by the keyword **adaptation**. Each of them has a name and relies on entities (class, method, attribute), which represent the adaptation targets (keyword **target**). The keyword **abstract** precedes each abstract adaptation or abstract adaptation target and it is mandatory to specify a comment. We consider that the documentation of an adapter is crucial because it improves the reusability of concerns. The description of the composition protocol proposed hereafter is made up with three parts: *i)* the observable entities, *ii)* the entities which observe and, *iii)* the concern which uses the design pattern.

[5]The examples of adapters presented in this paper only use a part of this syntax.

```
01 package designpattern.observer
02 abstract adapter ObserverAdapter {
03
04   abstract Class target ``class(es) being used as an observable'' : observableClass
05   abstract Method target `` method(s) triggering observer's changes'' : notifyingMethod
06        require notifyingMethod in observableClass.*
07
08   adaptation becomeObservable ``Modify class in order to make it observable'' :
09       extend class ImplObservable with observableClass
10   adaptation notifyingObserver
11        `` Alter notifyingMethods to tell observers about modification'' :
12       extend method notifyingMethod( ... ) with after { notifyObservers(); }
13 ------------------------------------------------------------------------------------
14   abstract Class target ``class being used as an observer'' : observerClass
15
16   adaptation becomeObserver ``Modify class to make it an observer'' :
17       inherit Observer in observerClass
18   abstract adaptation observerUpdate ``Introduce the updateObserver method in the observer class'' :
19       introduce method public void updateObserver(Observable o) in observerClass
20 ------------------------------------------------------------------------------------
21    abstract Class target
22      ``Class(es) initializing observable or observers objects'' : applicationInitClass
23    abstract Method target
24         ``Method(s) creating observable or observer objects'' : applicationInitMethod
25        require applicationInitMethod in applicationInitClass.*
26    abstract Attribute target
27         ``Attributes(s) pointing out observable objects'' : observableInstance
28        require observableInstance in applicationInitMethod.*
29    abstract Attribute target
30         ``Attributes(s) pointing out observer objects'' : observerInstance
31        require observerInstance in applicationInitMethod.*
32
33    adaptation initApplication
34         ``Alter applicationInitMethod to insert observers in the list of observables'' :
35        extend method ApplicationInitClass.applicationInitMethod( ... ) with after {
36          ObservableInstance.addObserver( ObserverInstance );
37         }
38  }
```

Figure 5: Composition Protocol of the Pattern `Observer`

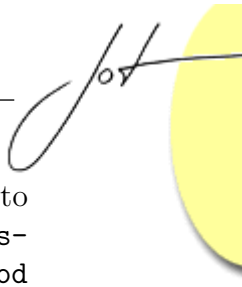
The lines *04* to *13* describe the targets and adaptations related to the observable entities. The adaptation targets `observableClass` and `notifyingMethod` respectively deal with the observed instances and the methods that must notify the entities observing them. At the time the composition protocol is specified, these methods and classes are not known yet, this is why the targets are preceded by **abstract**. They will become concrete when the programmer who wants to use the design pattern composes it with the chosen concern (GUI in our example). Nevertheless because it is possible to specify abstract adaptation targets, it is already possible to describe the corresponding adaptations. Two adaptations were specified: one of type *FeaturesOnlyAdded* (keyword **extend class**) and another of type *AfterInterception* (keyword **extend method**).

The first adaptation indicates that the contents of class `ImplObservable` must be inserted into the classes whose instances are observed. For the languages with multiple inheritance, such adaptation could be replaced by an adaptation of type *SuperClassIntroduction*. The second adaptation describes the insertion of a call of the method identified by `notifyObservers` at the end of the methods referred to by `notifyingMethod`. These methods must belong (keyword **require**) to the observed classes (`observableClass`). The call to the method `notifyObservers` enables the instances of the classes to play the role of observers to be warned of some state changes and to trigger the execution of a set of statements.

The lines *14* to *20* describe the adaptation target and two adaptations dealing with classes whose instances observe those addressed by the lines 4 to 13. The adaptation target `observerClass` refers to the classes chosen for playing this role. These two adaptations are of type *SuperClassIntroduction* (keyword **inherit**) and of type *MethodIntroduction* (keyword **introduce method**). The first adaptation adds the interface `Observer` to the classes attached to the adaptation target `observerClass`[6]. Let us mention that the interface `Observer` contains a method `update` which is of course abstract so that the second adaptation is abstract too. This indicates that an implementation must be attached when the context of use is known. It corresponds to the action to be performed on the observing instances.

The lines *21* to *37* describe the initialisation required to introduce the design pattern into the user concern. This initialisation fills the contents of the collection of instances which observe. It is located in the observed instances. The collection corresponds to the attribute `observers` and insertion of instances is done through method `addObserver`, both of them defined in the class `ImplObservable`. Four adaptation targets are specified to implement this initialization: *i)* `applicationInitClass` and `applicationInitMethod` correspond to the classes and methods where the attachment between the observed instances and the ones which observe is set, *ii)* `observableInstance` points out to the attributes referring to the observed instances and, *iii)* `observerInstance` records the attributes referring to the instances which observe. The insertion of the attachment is done through an adaptation of type *AfterInterception* (keyword **extend method**).

[6]Everything works as if those classes were specifying the declaration *`implements Observer`*



We may also note that the composition protocol described in Figure 5 permits to ensure that the attributes pointed out by `observableInstance` and `observerInstance` are declared within the method which correspond to `applicationInitMethod` (keyword **require**). The same approach is used (lines *06* to *12*) to ensure that the methods pointed out by the adaptation target `notifyingMethod` belong to the observed classes.

GUI-specific Adaptation

Let us now compose the concern related to the design pattern with the concern GUI. This task, which is assigned to the programmer of the application, is specified within a concrete adapter (Figure 6). Its contents seems *a priori* very simple but we show that inheritance of the abstract adapter (Figure 5) allows to control that the content is consistent with the composition protocol. The reader may also notice that the abstract adapter contains enough information to produce a comprehensive skeleton of the concrete adapter.

Lines *01* and *02* indicate that the concern *Observer* is composed (keyword **compose**) with the concern GUI. Because the package of the adapter (package `application.IHM`) is the same as the one of the concern GUI, the composition is then *in situ*. The adaptations will be performed directly within the classes `Button`, `Label` and `ApplicationInterface` which are in this package. Choosing a composition *ex situ* would produce the result of the composition in another container. It would also facilitate a new composition of the design pattern `Observer` with, for example, a transparent handling of the storage of the modified objects.

The adapter `ApplicationIHM` makes the adapter `ObserverAdapter` concrete. Its role is to complete its declarations. It is the adaptation name or the adaptation target that enable to bind the adapter with its homonym in the abstract adapter. The major part of these declarations is made with abstract adaptation targets (lines *04* to *10*). It is worth mentionning that if one of the adaptation targets is missing `ApplicationIHM`, this will be detected, because it is not possible to define a concrete adapter if some of its caracteristics remain abstract[7].

The concretization of the abstract adaptation targets implicitly completes the declaration of the corresponding adaptations. One may notice that some of the adaptation targets are concretized by an explicit enumeration of identifiers (`observableClass`, `ObserverClass`, `notifyingMethod` or `applicationInitMethod`), whereas others are concretized by a regular expression (`observableInstance` and `observerInstance`). Note that the concretization of an adaptation target may rely on another one declared within the adapter itself or in one of its parents (lines *08* to *10*). To make the description of adaptation targets easier we should not restrict ourselves to

[7]Of course nothing prevents us from building a new abstract adapter that inherits from an existing one, either to add some new adaptations or to make some of its adaptations or adaptation targets concrete.

```
01 package application.ihm
02 compose designpattern.observer.* with application.ihm
03 adapter ApplicationIHM extends ObserverAdapter {
04   target observableClass = application.ihm.Button
05   target observerClass = application.ihm.Label
06   target notifyingMethod = application.ihm.Button.fireActionPerformed()
07   target applicationInitClass = application.ihm.ApplicationInterface
08   target applicationInitMethod = applicationInitClass.applicationInterface()
09   target observableInstance = applicationInitMethod.button*
10   target observerInstance = applicationInitMethod.label*
11
12   adaptation observerUpdate :
13       introduce method public void updateObserver(Observable o) {
14            setText(``I have been notified of a button click'');
15        } in observerClass
16 }
```

Figure 6: Concretization of the composition protocol for the GUI

classical regular expressions[8] but we should extend their expressiveness to be able to point out, without enumerating them, the whole set of leaves of a hierarchy or all ancestors of a class. At the end, `ApplicationIHM` contains the description of the body of method `update` which should handle the specific needs of the concern GUI[9]. If you forget to concretize this method, it will be detected when the composition is performed.

To better differentiate the adapter and the behaviour attached to the concern, it may be interesting to isolate the pieces of code representing this behaviour (which is currently specified within the adapter; see line *12* of Figure 5 and lines *12* to *15* of Figure 6), into a specific class and to refer to it within the adapter.

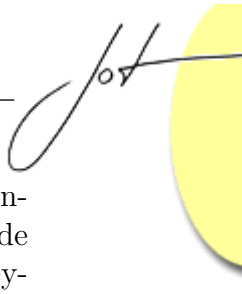
5 MORE ON THE SEMANTICS OF ADAPTATIONS

This section adds more details about the description of the set of adaptations supported by our approach. When it is suitable it refers to the example of composition of concerns presented in Section 4.

Providing the full reification of the adaptations would take too much space without bringing so much significant information. Therefore, we only mention the reification details and the controls influencing the understanding of the model[10].

There are eight variants of adaptation of type *Interception*. They apply at different steps of a method execution or of an attribute access. They are strongly influenced by AspectJ [11, 10]. They deal with attributes and method bodies. In the context of Java, these adaptations are performed more on classes than on interfaces because the latter may not contain method bodies and the attributes are only class constants (defined as *static final*).

[8] In the current implementation their expressiveness relies on the API `java.util.regex` of Java.
[9] Even if, in our example, the associated behaviour is very basic.
[10] The detailed reification of each adaptation appears in [20].



Four adaptations deal with methods. The adaptation *AroundInterception* enables executing a piece of code attached to the adaptation. Very often this code contains one condition *if-then-else* with, in one of the parts *then* or *else*, the keyword **proceed**, which means *to perform the original code of the method*. This type of interception enables choosing whether the method must be performed according to the context of execution.

The adaptation *BeforeInterception* (respectively *AfterInterception*) enables executing a piece of code before the first statement (respectively after the last statement, when it ends normally) of the original code of the method. If an exception is triggered and we need to modify the corresponding behaviour, then the adaptation *OnExceptionInterception* must be used to specify the additional statements that should be performed if an exception is triggered. Adaptations of type *AfterInterception* are showed in Figure 5 lines *10* to *12* and lines *33* to *36*.

Four other adaptations apply to attributes (whether they are shared by all occurrences of a class or belong to only one given occurence). These adaptations enable to specify a piece of code when the value of an attribute is read *OnGetInterception* or modified *OnSetInterception*. The execution of this piece of code is triggered either before this operation (*BeforeGet* or BeforeSet), or after (*AfterGet* or AfterSet).

There is another type of adaptation related to the body of a method: the *MethodRedefinition*. It allows the method body to be modified freely. Its counterpart for attributes also exists for programming languages supporting this feature (for example Eiffel).

The fusion of a class A with a class B is implemented by adaptations of type *ClassMerging*. They are inspired by facilities provided by Hyper/J [18]. Their semantics is the following: *i)* The super-classes of class A are inserted in the list of super-classes of class B. Of course, when the language supports single inheritance only, it is not possible to add any super-class if the class already has one (which is not inherited implicitly like *Object* in Java). *ii)* If people want that the fusion corresponds only to insertion of additional methods (with handling of possible name conflicts), they should use the adaptation *FeaturesOnlyAdded*, otherwise there is *CustomizedMerging*. For each method of class A, if one method with the same signature already exists in class B, then an adaptation of type *MethodMerging* is performed on the two methods (at present time, there are three kinds of method fusion and it is possible to choose the kind of fusion to be applied), otherwise the method is added to class B (adaptation of type *MethodIntroduction*). *iii)* Each attribute of class A, which does not already exist within B, is copied into it. An adaptation of type *FeaturesOnlyAdded* is proposed in the lines *08* and *09* of Figure 5.

The types of adaptation implementing the fusion of one method M with a method M' are a specialisation of *MethodMerging*. They have the following semantics:

- When both methods are abstract, the result of the fusion is also abstract.

- If only one of the two methods is concrete (adaptation *MergingWithAbstract*), then its body becomes the result of the fusion.

- If both methods are concrete then it is necessary to choose whether the body of method *M* must be performed before the one of method *M'* (adaptation *MergingBefore*) or after (adaptation *MergingAfter*).

From our point of view, the facilities provided by these last two adaptations are not sufficient and in most situations the code will have to be inserted somewhere else or conditionally. To cope with that, a first solution is to use an adaptation of type *MethodRedefinition* but it implies a duplication of code. Another possibility is to introduce a new type of adaptation that enables to make a fusion under some condition like in an adaptation of type *AroundInterception*.

In addition to the adaptations dedicated to the fusion, interception or redefinition, we propose adaptations enabling the insertion of super-classes, methods and attributes. The adaptation *MethodIntroduction* enables adding a new method to a classifier[11] when it does not already exist in the classifier, otherwise this is a redefinition *MethodRedefinition*). An example of this adaptation is proposed in Figure 5 (lines *18* and *19*).

The adaptation *AttributeIntroduction* allows to add an attribute to a class if it does not already exist in the classifier.

The adaptation *SuperClassIntroduction* enables inserting a classifier within the list of super-classes. One example of this type of adaptation is given lines *16* and *17* of Figure 5. Let us mention that for the object-oriented languages supporting several kinds of classifiers such as classes (abstract or not) and interfaces, like Java, it is necessary to adapt the semantics of this adaptation. It is also the case when there are fundamental differences between inheritance mechanisms, for example, that inheritance is multiple in Eiffel and C++ and single in Java and C#. In our implementation for the Java language, we made the following choices: *i)* This adaptation applies to both classes and interfaces, *ii)* It is forbidden to add an interface if it already belongs to the list of direct ancestors and, *iii)* A super-class may be introduced if the class implicitly inherits from the class *Object* only.

6 IMPLEMENTATION AND RELATED WORKS

In this section we first give an overview of the implementation, then we compare our work with AspectJ and Hyper/J. Finally we address other interesting related works.

[11] In Java it applies to both classes and interfaces but for the latter, the method must be *abstract* and *public*.

Overview of the Implementation

The model presented in the previous sections was implemented as a prototype (*JAdaptor*), it allowed us to validate the example proposed in Sections 2 and 4. The remaining part of this section gives an overview of the prototype implementation. A more detailed description is available in [20].

JAdaptor was implemented with Eclipse [6] as an extension of the Java plugin that comes with the Eclipse delivery. *JAdaptor* was implemented as a precompiler which is executed before the Java compiler. The main tasks performed by the precompiler are:

- the retrieval of information corresponding to the source code and to the adapters, contained in the abstract syntax trees[12],
- checking of the information consistency, flattening of the hierarchy of adapters and composition of concerns,
- the generation of the source code after the composition is achieved.

We chose to use the XML technology and in particular the *XML-Schema* [22] as pivot model for the specification of the adapters associated with one concern for several reasons: *i)* the description of adapters becomes independent of any language, which facilitates reuse of the composition protocol, *ii)* people may benefit from the large set of tools dedicated to the XML technology like the XML editor of Eclipse or the JAXB library [15] which enables to generate Java classes automatically from the information contained in XML files, *iii)* the generation of a view of adapters with a dedicated and adequate syntax is straightforward (see Figures 5 and 6). Then it is quite easy to implement an analyser for this syntax that produces XML files.

It is important to underline that the implementation of this approach relies on both Model Driven Architecture (MDA) and generative programming. The general nature of concepts presented in Section 2 shows that concerns could be described using UML (even if in our examples we used Java). The know-how related to the composition of concerns (Section 3) is recorded in a model which is independent from the language to be extended and adaptations are handled by transformations of the reification of the source code, using generators.

Comparison with AspectJ and Hyper/J

As mentioned in the introduction, two of the main sources of inspiration of our approach are aspect-oriented programming and subject-oriented programming. We now have to compare it with those two programming styles and show our contribution.

[12] One corresponds to the adaptation specifications and another one to the concerns themselves.

Our approach does not aim to compete with the approaches of separation of concerns but to improve object-oriented languages. Thus it introduces new means to express the specification of adaptations, which are required for a better reuse of class hierarchies. This is why our approach may be seen as an extension of an OO programming language or as a domain-specific language, which is external to it. In other words, the techniques used come mainly from the *ASoC* but the object paradigm is dominant.

Let us now consider the expressiveness of our approach with respect to *AspectJ* and *Hyper/J* and compare the example described in Section 4 with its implementation in both AspectJ and Hyper/J [8, 20]. We showed above[13] that it is crucial to be able to define a composition protocol which is independent from its future contexts of use. But *Hyper/J* does not allow it and *Aspect/J* allows it only partially. For example, in the latter some adaptations such as the insertion of new interfaces, methods or attributes cannot be defined independently from the context of use. It is the same thing for the adaptation targets which refer to classes or attributes.

We also noticed some deficiencies for the supported adaptations. *Aspect/J* does not provide any adaptation to merge classes or methods. The adaptations provided by *Hyper/J* for the interception of method are not sufficient, therefore very few contextual information is accessible, so that it is not possible to get the parameters and return value of an adapted method. Moreover *Hyper/J* does not provide any adaptation for intercepting the access to an attribute.

Finally, even if our example does not put too much emphasis on the location of composition, *Hyper/J* and *Aspect/J* provide only one mode of composition: *ex situ* for the first one, and *in situ* for the second one.

We can see that *Aspect/J* has better results than *Hyper/J* (for the criteria we are considering). Still, AspectJ has deficiencies. Using it for our example would yield the following problem: *i)* It is not possible to specify the insertion of interface `Observer` within the composition protocol; *ii)* The contents of `ImplObservable` must be described directly within the abstract aspect (it may be compared to our abstract adapter); *iii)* None of the adaptation targets necessary for the implementation of the initialisation required by the integration of the design pattern may be described within the composition protocol.

Other Related Works

There are other languages or approaches dealing with AOP and SOP. Among them, ConcernJ [3] relies on component filters in order to implement the separation of concern. Caesar [14] mixes AOP with component-oriented programming in such a way that specifications of aspects (i.e. the interfaces) are separated from their implementation and from their deployment. JAC [19] is also inspired from components but uses a reification for the specification of an aspect and thus requires no

[13] The Ph.D Thesis of Laurent Quintian [20] provides a more detailed study.

language extension. Jiazzi [16] allows to program by roles or by points of view, but the separation of concerns may be performed only within the same class.

The fact that several approaches are influenced by component-oriented programming shows the interest of such an approach for achieving a separation of concerns and then reusability. We can mention some of the promising features provided by this paradigm: component substitutability (static and dynamic), hierarchical organization of components, separation of interface and implementation, separation of the functional and configuration concerns, etc.

7 CONCLUSION AND PERSPECTIVES

The object-oriented languages at the end of the eighties contributed to a better reusability of the software but the needs of applications which are greater and greater everyday point out their limits. This is why we propose a model and its implementation to improve the reusability of languages while preserving the robustness of application.

Our approach relies on the idea that to make a concern reusable implies to equip it with its *user manual*. One interesting point of our approach is that it is not only textual and static, it is a full entity (*the adapter*). It enables the specification of the entities that will be adapted (*the adaptation targets*) and the *adaptations* themselves. An adapter is used only at the beginning of the compilation process to weave the concerns involved in an application

An adapter provides *i)* an encapsulation entity supporting abstract (partial) declarations, *ii)* typing of both adaptations and adaptation targets, *iii)* an inheritance relationship (especially *reification inheritance* [13]) which implements dynamic binding between the adaptations and the adaptation targets, *iv)* an assertion mechanism in order to specify the dependencies between the adaptation targets and, *v)* a textual documentation.

The fact that this approach is independent from the chosen programming language is also important even if the description independency of the adapters is not strong enough from our point of view[14]. But we study some alternatives like locating this code outside the adapter, in a class. The model has true evolution capabilities. They rely on an object reification which benefits from the underlying language (modularity, inheritance...). The set of concepts addressed in Section 2 and the hierarchy of adaptations described by Figure 4 are extendible. The independence of the approach and the evolutivity of the model are favoured at the implementation level by the use of the XML technology and by the generative capabilities provided by the Eclipse platform.

Perspectives of our work include the improvement and extension of the model, and improvement of its implementation. On the implementation side, it would be

[14]The treatments to be inserted use the chosen language (for us, Java).

interesting to consider directly the Java byte-code in order, for example, to use our approach to reuse libraries provided without their source code. It would be also interesting to validate our model on other languages such as Eiffel or C++.

We aim to perform some improvements upon the suggested model in order to resolve its weaknesses and to increase the expressiveness of the composition model. First the hierarchy of adaptations should be extended to offer new adaptation capabilities. In particular it is necessary to provide the ability to modify the *import* clause of a class. The support of this adaptation is mandatory to be able to implement composition *ex-situ* within already existing applications and to specify them into the composition protocol. The fact that generic classes are supported by several languages suggests proposing adaptations dedicated to this kind of classifier which allow to take into account generic parameters. Other improvements include a better support for composition of adapters attached to the same concern, and improvement of the flexibility of the composition protocol by enabling to define alternative adaptations. It is important to be able to reuse the associated concern in an even greater number of contexts without being obliged to reduce the spectrum of the composition protocol and as a consequence, the programmer assistance and the controls which are performed. One improvement which derives from the latter is the implementation of optional clauses. This would enable to differenciate between an adaptation within the framework of an existing application and another one dealing with concerns to be composed in order to build an application. A more long term perspective is to adapt our model to apply it to component-oriented programming and to model-oriented programming.

REFERENCES

[1] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *Proceedings of ECOOP'92*, LNCS(615), pages 372–395. Springer-Verlag, 1992.

[2] G. Bracha and W. Cook. Mixin-based inheritance. In *proceedings of OOPSLA/ECOOP 90*, 1990.

[3] P. Caro. *Adding Systemic Crosscuting and Super-Imposition to Composition Filters*. Ph.D Thesis, University of Twente, Netherlands, 2001.

[4] P. Cointe. Reflective languages and metalevel architectures. *ACM Comput. Surveys*, 28(4es):151, 1996.

[5] K. Czarnecki and J. Vlissides. Domain-Driven Development. Special Track at OOPSLA'03 URL: http://oopsla.acm.org/oopsla2003/files/ddd.html, 2003.

[6] Eclipse fundation. Environnement eclipse. http://www.eclipse.org, 2004.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co., 1994.

[8] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *proceedings of OOSPLA'02*, 2002.

[9] E. Kendall. Role model designs and implementations with aspect-oriented programming. In *Proceedings of OOPSLA'99*, pages 353–369, Denver, Colorado, United States, November 1999. ACM Press.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Communications of the ACM*, 44(10):59–65, October 2001.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Proceedings of ECOOP'01*, LNCS(2072), pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Proceedings of ECOOP'97*, LNCS(1241), pages 220–242. Springer-Verlag, June 1997.

[13] B. Meyer. *Object-Oriented Software Construction*. Professional Technical Reference. Prentice Hall, 2^{nd} edition, 1997.

[14] M. Mezini and K. Ostermann. Conquering Aspects with Casear. In *Proceedings of the 2^{nd} International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, Massachusetts, USA, March 2003, pages 90–99.

[15] Sun microsystems. Java architecture for xml binding jaxb. http://java.sun.com/xml/jaxb/index.jsp, 2004.

[16] S. McDirmid and W. Hsieh. Aspect-Oriented Programming with Jiazzi. In *Proceedings of the 2^{nd} International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, Massachusetts, USA, March 2003.

[17] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of OOPSLA 95*, 1995.

[18] H. Ossher and P. Tarr. Hyper/j: Multi-dimentionnal separation of concern for java. In *Proceedings of ICSE'00*, 2000.

[19] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Dynamic Wrappers : Handling he Composition Issue with JAC. In *proceedings of TOOLS'01*, pages 56–65, 2001.

[20] L. Quintian. *JAdaptor : Un modèle pour améliorer la réutilisation des préoccupations dans le paradigme objet*. Ph.D thesis in Computer Science, University of Nice - Sophia Antipolis, France, July 2004.

[21] C. Szyperski, J. Bosch, and W. Weck. Component-oriented programming. In *Proceedings of the workshop on Component-Oriented Programming at ECOOP'99*, LNCS(1743), pages 184–192, Lisbon, Portugal, June 1999. Springer-Verlag.

[22] W3C. Xml schema. http://www.w3.org/XML/Schema, 2004.

ACKNOWLEDGEMENTS

We gratefully acknowledge Karine Arnout (ETH Zurich) for her valuable comments and feedback on the paper.

ABOUT THE AUTHORS

Philippe Lahire is Assistant Professor with an HDR (accreditation to supervise research), at the University of Nice-Sophia Antipolis. He can be reached at Philippe.Lahire@unice.fr. See also http://www.i3s.unice.fr/.

Laurent Quintian received his Ph.D in Computer Science from the University of Nice-Sophia Antipolis. He can be reached at Laurent.Quintian@unice.fr. See also http://www.i3s.unice.fr/.