The Observer Pattern Using Aspect Oriented Programming

Jacob Borella (jborella@it.edu)
IT-University of Copenhagen

VikingPLOP september 19-22 2003

Abstract

Usually only the solution of the 23 original design patterns, first proposed by the “gang of four”, are considered reusable, whereas the implementation is not. Using Aspect Oriented Programming this paper provides a new solution to and a reusable implementation of the Observer pattern. The implementing languages are AspectJ and AspectS respectively. A consequence of using this new implementation in an application is added complexity, why a simpler and not reusable realization of the pattern might be preferable in some cases.

1 Introduction

Since the concept of design patterns were introduced in the article [4] written by Gamma, Helm, Johnson, and Vlissides, the authors later to be known simply as “the gang of four”, much work has been devoted to investigate design patterns. A lot of the work has been used to identify other design patterns than the original 23 introduced in [3], books has been written about how to use and implement patterns in different programming languages, and a lot has been written about domain specific patterns. In spite of all the work done, the design patterns are still considered reusable mostly at the conceptual level. As a consequence the programmer still must implement the patterns for each application he is building.

In [2] it is noted that some architectural abstraction will sooner or later become part of the language, suggesting that the best solution to the missing reusability is to make the patterns part of the language. The drawback of this approach is more complicated languages.

Another approach to the problem of reusability, is made possible by the introduction of Aspect Oriented Programming (AOP), since this kind of programming enables one to introduce code into an existing codebase and change behavior of existing methods.

In general there seems to be two approaches for creating AOP implementations of the patterns. To understand the difference between the two approaches it is important to distinguish between problem, solution, and implementation of a pattern. The problem identifies a design problem and explains when to use a given pattern to solve this problem. The solution descripes in nonimplementation-specific terms a solution to the problem. This is done by identifying main elements of the pattern and how these elements relate and collaborate. The implementation is a realization of a solution.

The first approach is to use AOP to implement the solutions that are given in [3]. The other approach is to create genuinly new solutions to the problems of the patterns in the light of AOP and implement these solutions.

What is new in this paper is, that a new solution to, and implementation of, the Observer pattern is provided. This approach seems reasonable, because the solutions to the problems of the patterns listed in [3] are not considering AOP technologies. The implementing languages are AspectS [7] for Smalltalk and AspectJ [8] for Java. The reason for choosing AspectS is that aspects can be applied at runtime and because the language has a very different instantiation policy than AspectJ, which is choosen mainly because it is the most popular language.

2 An example of AOP

To introduce some important concepts of AOP a very simple, but still usefull and genuinly new, implementation of the Observer pattern is provided. Assume

```
public class Subject {
    private int state;

    public void setState(int newState) { state = newState; }
    public int getState() { return state; }
}
public class Observer {
    public void showMessage(String msg) { System.out.println("[Observer] " + msg); }
}
public aspect SubjectAgent {
    private java.util.LinkedList observers = new java.util.LinkedList();

    public void Observer.update(Subject s) {
        showMessage("subject changed state to " + s.getState());
    }

    public void addObserver(Observer o) { observers.add(o); }
    public void removeObserver(Observer o) { observers.remove(o); }

    after(Subject s) : set(int Subject.state) && target(s) {
        java.util.Iterator i = observers.iterator();
        while (i.hasNext()) {
            ((Observer) i.next()).update(s);
        }
    }
}
```

Figure 1: The observer pattern.

that it doesn't matter whether the implementation is reusable or how nice it looks. The programmer simply want to get the job done. Each time a subject changes state, some action must happen in its related observers. In the case of this example a subject can set an instance variable called *state*. Each time *state* is changed, all observers must show the new state on the screen. This can be realized by the code listed in figure 1.

The new construct in this example is the aspect **SubjectAgent**, which in this example consists of a pointcut, an advice, an introduction, and two methods. The introduction, which is located in line 13-15, introduces (read inserts) a method *update(Subject)* in the **Observer** class. The pointcut in figure 1 is the part in line 20 after the ':'. It defines some well defined place in the executing code, in this case the assignment of a new value to *state*. The advice denotes what to do when a pointcut is reached. Code can usually be executed before, after, and/or around the pointcut. In the example the code is to be executed after the above mentioned pointcut. The code to execute is within the brackets in line 21-24, and simply calls *update* on all registered observers. The methods for adding and removing observers are as in Java.

Although this example is very simple, it might be a very usefull implementation of the Observer pattern in case the code is unlikely to be changed. It is very simple to understand, but still separates the observer from the subject. The motivation for using a more complicated, but reusable, solution should be the same as always.

3 Analysis and requirements

The intent of the Observer pattern is to:

> *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.* [3, p. 293]

This definition is implementation specific, because it assumes that "dependents are notified". Such implementation is known as the push model, but it is also possible to let the dependent poll for state changes. In this case there is no notification. Using polling is considered inefficient for most applications, why this paper will not consider such a solution.

In the original solution, the subject has at least two methods; *addObserver(Observer)* and *removeObserver(Observer)*. This choice of interface is not the only possible one. Furthermore it must be decided where to place the methods of the interface. Assuming that it is possible to make the pattern reusable, there are two or three possibilities, depending on whether the pattern is realized as a class; at the observer, subject, or pattern. The interface will be slightly affected by where the methods are placed.

At the one extreme the original interface is kept. The resulting method names can be seen in table 1. The consequence of using the interface in table 1 is that it is only possible to introduce behavior at the class level (that is any observer class must do the same in response to a change in a subject class). The behavior must still be implemented (that is the observer must have some update logic, which can be called by the subject).

Placed on the subject.	
addObserver: anObserver	Indicate that *anObserver* now listens for changes in the subject.
removeObserver: anObserver	Indicate that *anObserver* no more listens for changes in the subject.
Placed on the observer.	
observe: aSubject	Indicate that *aSubject* now is observed by the observer.
stopObserving: aSubject	Indicate that *aSubject* is no more observed by the observer.
Placed on the pattern.	
startObserving: anObserver subject: aSubject	Explanation as for the observer case.
stopObserving: anObserver subject: aSubject	Explanation as for the observer case.

Table 1: Original interface depending on placement.

At the other extreme it is possible to let any observer instance do some action in response to a state change in any subject instance. The methods are listed in table 2. Remark that an agent is used in the table. The role of the agent is to do some actions in one or more observers whenever an event (that is a state change) occurs in one or more subjects. Using this definition the agent maps many observers to many subjects. Depending on the implementation it is beneficial to use a different mapping (one observer to many subjects or one subject to many observers).

Placed on the observer.	
do: anAction when: anEvent in: aSubjectInstance	Indicate that the observer must do *anAction* in response to *anEvent* in *aSubjectInstance*. Return *anId* for the agent.
stopDoing: anId	Remove the agent given by *anId* from the observer.
Placed on the subject.	
on: anObserverInstance do: anAction when: anEvent	Indicate that *anObserverInstance* must do *anAction* in response to *anEvent* in the subject. Return *anId* for the agent.
stopDoing: anId	Remove the agent given by *anId* from the subject.
Placed on the pattern.	
on: anObserverInstance do: anAction when: anEvent in: aSubjectInstance	Indicate that the *anObserverInstance* must do *anAction* in response to *anEvent* in *aSubjectInstance*. Return *anId* for the agent.
stopDoing: anId	Remove the agent given by *anId*.

Table 2: AOP interface depending on placement.

As opposed to the original interface, there is no control of whether the code introduced in *anAction* is correct. For instance the programmer might as an error specify the same action twice in response to a subject event. Furthermore *anAction* in table 2 cannot access private fields in the observer unless it is placed on the observer as an inner class or mechanisms are build into the language, which allows other objects to access internal state of the observer (for instance the keyword *privileged* in AspectJ or *friendly* in C++). The advantage of using inner classes is as noted above that they can access internal state. On the other hand placing them as inner classes leads to code scattering and worse the actions must be removed manually from every observer if some observer is to be used in a context where it doesn't play that role.

Placing *anAction* in separate classes avoids the above problems, but the approach might lead to a breach of encapsulation, because the observer is forced to expose the state that must be changed or methods to call when a subject change occurs.

If the methods are placed on the pattern, the interface is a little more complicated than in the two other cases, but besides from that there doesn't seem to be any logical choice of placement.

4 Solution

A solution can be seen in figure 2. It has two classes (**Subject** and **Observer**) and an aspect (**Agent**) not counting subclasses. No specific interface for the Observer pattern behavior is chosen, but it is decided to place the methods on the **Observer**. The role of the **Agent** is to observe the **Subject** and execute some code on behalf of the **Observer** each time a subject change occurs. Which kind of subject change one can listen for is specified by subclassing the **Agent**. The **Observer** is composed of zero or more **Agent**'s depending on which subject

Figure 2: The observer using composition.

changes it listens for. The composition can be changed at runtime, making it possible to change the subject changes an observer reacts to. Remark that this is not the same as the old callback protocol, since the **Subject** is left unchanged. The **Observer** has the responsibility of installing **Agent**'s and as a consequence controls actions taken on itself.

The methods described in table 1 or 2 can be introduced in the **Observer** class by introduction or by hardcoding them into the class. If using introduction the code is separated into three components; the observer, the subject, and the observer pattern.

5 A Smalltalk implementation

An implementation of the observer pattern using introduction in the observer of the methods listed in table 2 is provided in appendix A. Smalltalk (Squeak) and AspectS are used as implementing language.

The implementation is highly reusable. The only thing which is required is to write a new agent per type of subject event and registrering it under some name in the pool dictionary *SubjectEventConstants*. By looking at the implementation of an **Agent** subclass in figure 3 it is easy to realize that the implementing work is minimal. The only thing required is to supply the pointcut which denfines

```
getPointcut
    ^ OrderedCollection
            with: (AsJoinPointDescriptor targetClass: Subject targetSelector: #price:)
```

Figure 3: The only method of the **SubjectNameChangeAgent**.

when a subject change occurs (in this case when the message *price:* is sent to a **Subject**). The **Subject** and **Observer** classes are unaffected by the Observer pattern only defining their own methods.

The **Agent** class is doing the actual method calls when a subject event occurs. It is listed in figure 4 and is a subclass of **AsAspect** meaning that it can introduce

```
adviceSubjectChange
    ^ AsBeforeAfterAdvice
        qualifier: (AsAdviceQualifier attributes: { #receiverInstanceSpecific. })
        pointcut: [self getPointcut]
        afterBlock: [:receiver :arguments :aspect :client :result |
            doBlock copy fixTemps valueWithArguments: {receiver. self observer.}]

getPointcut
    self subclassResponsibility

doBlock
    ^ doBlock

doBlock: aBlock
    doBlock := aBlock

observer
    ^ observer

observer: newObserver
    observer := newObserver
```

Figure 4: Methods of the **Agent** class.

advice. In this case one advice is introduced, which activates a block of code in an **Observer** each time a pointcut defined by its subclass (for instance the one in figure 3) occurs. The **ObserverPattern** is not listed here, but it introduces methods for managing all the agents in the **Observer**.

```
o := Observer new.
s := Subject new.
pattern := ObserverPattern new.
pattern addObserverClass: Observer.
pattern install.
a := o do: [:subject :observer | observer showMessage: 's name change to: ' , subject name]
      when: SubjectNameChange
       in: s.
s name: 'apple'.
o stopObserving.
s name: 'want be noticed'
pattern uninstall.
```

Figure 5: Example of using the observer pattern.

Running the example in figure 5 results in the observer writing: "[observer] s name change to: apple" to the transcript when the subjects changes name to 'apple'. The second name change goes unnotified, since the observer stops observing before. Remark that the pattern is instantiated and installed at runtime, making it possible to change the implementation of it and assign roles at runtime. In a more advanced version it could also be possible to create agents at runtime.

The programmer is required to deregistrer all agents prior to dereferencing the observer (in the implementation of the original solution this results in some calls to *removeObserver(Observer)*), otherwise it will still be active. There is no workaround for that in languages using garbage collection. Calling uninstall on the pattern results in the agents being removed as well.

6 An AspectJ implementation

In appendix B an implementation using AspectJ is provided. Due to the more detailed pointcuts of AspectJ it is possible to listen not only for method calls as in AspectS, but also the assignment of variables, instantiation of objects etc. This gives AspectJ an advantage over AspectS.

The solution is very reusable. To introduce an observerrole into some object it is only required to declare it in a subclass of **ObserverPattern**. For each kind of subject change, an agent must be written (which as before is a trivial task), and it must be specified in the subclass of **ObserverPattern**.

Since the implementation looks a lot like the one in Smalltalk/AspectS, only an example of binding the pattern is provided in figure 6. This file is compiled with the rest of the codebase, resulting in the **ConsoleLogger** being an observer and **SomeSubject** a subject. A consequence is, that the code must be recompiled

```
aspect ObserverPatternBinding extends ObserverPattern {

    public static final int SomeSubject.NAME_CHANGE = 0;
    public static final int SomeSubject.PRICE_CHANGE = 1;

    declare parents: ConsoleLogger implements Observer;
    declare parents: SomeSubject implements Subject;

    public Agent ConsoleLogger.getAgentFor(ActionIdentifier id) {
        switch(id.action) {
            case 0:
                return SomeSubjectNameChangeAgent.aspectOf(id.subject);
            case 1:
                return SomeSubjectPriceChangeAgent.aspectOf(id.subject);
            default:
                return null;
        }
    }
}
```

Figure 6: Example of bindng the observer pattern.

each time one wants to apply the pattern to a new class, remove the pattern from some class, or change the implementation of the pattern. This is a drawback of using AspectJ.

Since there are no blocks in AspectJ or Java, an **Action** class is introduced. It has the method **execute**, which is called on subject changes. Due to some lack of instantiation policy in AspectJ, the agent maps one subject instance to many actions. This approach is very similar to the AOP implementation of the original solution.

7 Related work

In [5] Hannemann and Kiczales give an implementation of the original solution to the Observer pattern. (At Hannemann's homepage an implementation of the 23 design patterns from [3] are provided.) Their implementation, which introduces methods corresponding to the ones listed in table 1 on the pattern, is also reusable only requiring the binding of roles and specification of subject changes. There are some problems using their implementation, which relates to their placement of the observer methods on the pattern implementation. The method *aspectOf()* is called on the implementing pattern class to access the pattern implementation. Changing the implementation will thus require the programmer to change all places in the codebase where the pattern methods are called, which is potentially a lot of places. A possible workaround is to use an abstract factory to access the pattern implementation, but this leads to code that are more complicated than necessary. Another problem is that they simulate state of the subject (the weak hashmap containing the observers per subject). Since there are introduction mechanisms in AspectJ it seems to be a better solution to use this mechanism.

Some critique of the implementation by Hannemann and Kiczales is given in [9], providing their own implementation of the Observer pattern in the language Caesar. One point is, that the methods are introduced as top level methods on the pattern. Besides from the above problem that one has to call the methods on the aspect, for which there is a workaround, it can be discussed whether such critique are justified, but an alternative not using top level methods are provided in this paper.

Another point is the lack of dynamic deployment. This critique seems somewhat unjustified, since it lies in the nature of the language. The AspectS implementation of the Observer pattern is an example of a dynamic deployable implementation.

Other contributions to implementing design patterns using AOP are [6], providing an implementation of the Decorator and Visitor patterns, and [1] providing an implementation of the Visitor pattern using AspectJ.

8 Conclusion

The primary goal was to obtain a reusable implementation of the Observer pattern using AspectS/Smaltalk and AspectJ/Java respectively. This requirement is met, since the only things left to do in the implementations is specifying the bindings. That is which objects are observers and which subject changes there are in the system. Furthermore the actions to do when a subject change occurs must be provided, but that seems reasonable. The reusable part of the implementation is provided in the appendices A.1 and B.1. There are differences between the two implementations, which are mostly due to the differences between the implementing languages.

The AspectS implementation is dynamic because of the dynamic nature of the language. At runtime it is both possible to replace the pattern implementation, assign roles, and introduce new pointcuts. Since the pointcuts can only catch message sends between objects and the throwing of exceptions, the granularity of the subject events that can be caught are somewhat limited.

The AspectJ implementation is static, which are mostly due to the fact that the advice are weaved into the codebase at compiletime. It is thus not possible, without an extra effort, to change pattern implementation at runtime or assign new roles. On the other hand the pointcuts that can be expressed are far more detailed than in AspectS.

Both implementations raises the problem of breach of encapsulation. The actions that are defined can only access external methods and state on the observer.

As a consequence the observer might be forced to expose some state which were meant to be internal. Furthermore the pointcuts in Smalltalk can only intercept message sends and thrown exceptions. In order to be able to detect internal subject changes it might be required to access this state through message sends, which also leads to breach of encapsulation.

The conclusion is that AOP can make the Observer pattern reusable, providing an alternative to changing the language. A new consequence of using the new Observer pattern solution is added to the list in [3, p. 296]; the reusability adds complexity. For many applications the very simple solution provided in figure 1 might be as usefull in applications where the observer/subject relation is unlikely to change.

Whether other patterns can be made reusable are the subject of future work, some of which is already done in the articles listed in the references section. Still a lot of work lies ahead before all of the patterns are fully explored.

References

[1] D. Bardou and O. Hachani. Using aspect-oriented programming for design patterns implementation. In *Reuse in Object-Oriented Information Systems Design workshop. 8th International Conference on Object-Oriented Information Systems (OOIS 2002)*, 2002.

[2] B. B. Christensen. Architectural abstractions and language mechanisms. In *Proceedings of the Asia Pacific Software Engineering Conference*, 1996.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison-Wesley, July 2001.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.

[5] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[6] R. Hirschfeld, R. Lämmel, and M. Wagner. Design Patterns and Aspects — Modular Designs with Seamless Run-Time Integration. In *Proc. of the 3rd German GI Workshop on Aspect-Oriented Software Development, Technical Report, University of Essen*, 2003.

[7] Robert Hirschfeld. Aspect-oriented programming with aspects.

[8] http://www.eclipse.org/aspectj/.

[9] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, 2003.

A The observer pattern implementation in Smalltalk.

A.1 Reusable part

```
*** Agent ***
AsAspect subclass: #Agent
    instanceVariableNames: 'observer doBlock '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Design Patterns-Observer'

Instance methods:
adviceSubjectChange
    ^ AsBeforeAfterAdvice
        qualifier: (AsAdviceQualifier attributes: { #receiverInstanceSpecific. })
        pointcut: [self getPointcut]
        afterBlock: [:receiver :arguments :aspect :client :result |
            doBlock copy fixTemps valueWithArguments: {receiver. self observer.}]

getPointcut
    self subclassResponsibility

doBlock
    ^ doBlock

doBlock: aBlock
    doBlock := aBlock

observer
    ^ observer

observer: newObserver
    observer := newObserver

*** ObserverPattern ***
AsAspect subclass: #ObserverPattern
    instanceVariableNames: 'activatorsPerObserver observers '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Design Patterns-Observer'

Instance methods:
removeAllActivators
    activatorsPerObserver valuesDo: [:bagOfActivators |
        bagOfActivators do: [:activator | activator uninstall]]

init
    activatorsPerObserver := Dictionary new.
    observers := OrderedCollection new.

adviceAddTo
    ^ AsIntroductionAdvice
        qualifier: (AsAdviceQualifier attributes: { #receiverClassSpecific. })
        pointcut: [self getPointcut: #add:to:]
        introBlock: [:receiver :arguments :aspect :client |
            arguments second addReceiver: arguments first]

adviceDoWhenIn
    ^ AsIntroductionAdvice
        qualifier: (AsAdviceQualifier attributes: { #receiverClassSpecific. })
        pointcut: [self getPointcut: #do:when:in:]
        introBlock: [:receiver :arguments :aspect :client |
            | activator observerActivators |
            activator := arguments second new.
            activator observer: receiver.
            activator doBlock: arguments first.
            activator addReceiver: arguments third.
            activator install.
            observerActivators :=
                activatorsPerObserver at: receiver ifAbsentPut: [Bag new].
            observerActivators add: activator.
            activator]

adviceRemoveFrom
    ^ AsIntroductionAdvice
        qualifier: (AsAdviceQualifier attributes: { #receiverClassSpecific. })
        pointcut: [self getPointcut: #remove:from:]
        introBlock: [:receiver :arguments :aspect :client |
            arguments second removeReceiver: arguments first]

adviceStopObserving
    ^ AsIntroductionAdvice
        qualifier: (AsAdviceQualifier attributes: { #receiverClassSpecific. })
        pointcut: [self getPointcut: #stopObserving]
        introBlock: [:receiver :arguments :aspect :client |
            (activatorsPerObserver at: receiver) do: [:a | a uninstall].
            activatorsPerObserver removeKey: receiver.]

adviceUndo
    ^ AsIntroductionAdvice
        qualifier: (AsAdviceQualifier attributes: { #receiverClassSpecific. })
        pointcut: [self getPointcut: #undo:]
        introBlock: [:receiver :arguments :aspect :client |
            arguments first uninstall.
            (activatorsPerObserver at: receiver)  remove: arguments first]

getPointcut: theSelector
    | o |
    o := OrderedCollection new.
    observers do: [:observerClass |
        o add: (AsJoinPointDescriptor targetClass: observerClass targetSelector: theSelector)].
    ^ o

uninstall
    self removeAllActivators.
    super uninstall

addObserverClass: theClass
    observers add: theClass

removeObserverClass: theClass
    observers remove: theClass


Class methods:
new
    ^ super new init
```

A.2 Example code

```
*** Main ***
Object subclass: #Main
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: 'SubjectEventConstants '
    category: 'Design Patterns-Observer'
```

```
Class methods:
runExample
    | o s0 s1 a pattern|
    o := Observer new.
    s0 := Subject new.
    s1 := Subject new.
    pattern := ObserverPattern new.
    pattern addObserverClass: Observer.
    pattern install.
    a := o do: [:subject :observer | observer showMessage: 's name change to: ' , subject name]
          when: SubjectNameChange
           in: s0.
    o add: s1 to: a.
    s0 name: 'apple'.
    o stopObserving.
    s1 name: 'pear'.
    pattern uninstall.
    s1 name: 'want be noticed'

createSubjectEventTable
    SubjectEventConstants
        at: #SubjectNameChange put: SubjectNameChangeAgent ;
        at: #SubjectPriceChange put: SubjectPriceChangeAgent.

*** Observer ***
Object subclass: #Observer
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Design Patterns-Observer'

Instance methods:
showMessage: theMessage
    Transcript show: '[observer] ' , theMessage ; cr.

*** Subject ***
Object subclass: #Subject
    instanceVariableNames: 'price name '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Design Patterns-Observer'

Instance methods:
name
    ^ name

name: newName
    name := newName

price
    ^ price

price: newPrice
    price := newPrice

*** SubjectNameChangeAgent ***
Agent subclass: #SubjectNameChangeAgent
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Design Patterns-Observer'

Instance methods:
getPointcut
    ^ OrderedCollection
```

```
            with: (AsJoinPointDescriptor targetClass: Subject targetSelector: #name:)

*** SubjectPriceChangeAgent ***
Agent subclass: #SubjectPriceChangeAgent
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Design Patterns-Observer'

Instance methods:
getPointcut
    ^ OrderedCollection
            with: (AsJoinPointDescriptor targetClass: Subject targetSelector: #price:)
```

B The observer pattern using AspectJ

B.1 Reusable part

Action.java

```
public abstract class Action {

    public abstract void execute();
}
```

Agent.java

```
import java.util.Iterator;
import java.util.Hashtable;

public abstract aspect Agent pertarget(target(ObserverPattern.Subject)){

    private Hashtable activators = new Hashtable();

    protected abstract pointcut action();

    after() : action() {
        Iterator i = activators.values().iterator();
        while (i.hasNext()) {
            ((Action) i.next()).execute();
        }
    }

    public void addAction(ObserverPattern.Observer key, Action a) {
        activators.put(key, a);
    }

    public void removeAction(ObserverPattern.Observer key) {
        activators.remove(key);
    }
}
```

ObserverPattern.java

```
import java.util.Iterator;
import java.util.Hashtable;

public abstract aspect ObserverPattern {
    public static class ActionIdentifier{
        public int action;
        public Subject subject;
    }

    protected interface Subject{};//marker interface
    protected interface Observer{
        abstract Agent getAgentFor(ActionIdentifier id);
    };

    //introductions in observer
    private Hashtable Observer.installedActions = new Hashtable();
    private int Observer.nextId = 0;

    public int Observer.doAction(Action theAction, int when, Subject in){
        int idInt = nextId++;
        ActionIdentifier id = new ActionIdentifier();
        id.action = when;
        id.subject = in;
        getAgentFor(id).addAction(this, theAction);
        installedActions.put(new Integer(idInt), id);
        return idInt;
    }

    public void Observer.undoAction(int undoId){
        ActionIdentifier id = (ActionIdentifier) installedActions.get(new Integer(undoId));
        getAgentFor(id).removeAction(this);
        installedActions.remove(id);
    }

    public void Observer.stopObserving() {
        Iterator i = (Iterator) installedActions.values().iterator();
        while (i.hasNext()) {
            ActionIdentifier id = (ActionIdentifier) i.next();
            getAgentFor(id).removeAction(this);
        }
        installedActions.clear();
    }
}
```

SubjectObserverAction.java

```
public abstract class SubjectObserverAction extends Action {
    protected Object sub, obs;

    public SubjectObserverAction(Object subject, Object observer){
        obs = observer;
        sub = subject;
    }
}
```

B.2 Example code

ConsoleLogger.java

```
public class ConsoleLogger {
    private String name;

    public ConsoleLogger(String name) {
        this.name = name;
    }

    public void log(String msg) {
        System.out.println("[" + name + "] " + msg);
    }

}
```

ObserverMain.java

```
import it.edu.jborella.patterns.observer.SubjectObserverAction;

public class ObserverMain {
    public static void main(String[] args) {
        SomeSubject subject = new SomeSubject();
        ConsoleLogger logger = new ConsoleLogger("s logger");
        int id0 = logger.doAction(
            new SubjectObserverAction(subject, logger){
                public void execute(){
                    ((ConsoleLogger) obs).log("subject changed price to: " +
                    ((SomeSubject) sub).getPrice());
                }
            },
            SomeSubject.PRICE_CHANGE,
            subject
        );
        int id1 = logger.doAction(
            new SubjectObserverAction(subject, logger){
                public void execute(){
                    ((ConsoleLogger) obs).log("subject changed name to: " +
                    ((SomeSubject) sub).getName());
                }
            },
            SomeSubject.NAME_CHANGE,
            subject
        );
        subject.setPrice(123);
        subject.setName("apple");
        logger.stopObserving();
        //logger.undoAction(id0);
        subject.setPrice(3987);
        subject.setName("burger");
    }
}
```

ObserverPatternBinding.java

```
import it.edu.jborella.patterns.observer.Agent;
import it.edu.jborella.patterns.observer.ObserverPattern;

aspect ObserverPatternBinding extends ObserverPattern {

    public static final int SomeSubject.NAME_CHANGE = 0;
    public static final int SomeSubject.PRICE_CHANGE = 1;

    declare parents: ConsoleLogger implements Observer;
    declare parents: SomeSubject implements Subject;

    public Agent ConsoleLogger.getAgentFor(ActionIdentifier id) {
        switch(id.action) {
            case 0:
                return SomeSubjectNameChangeAgent.aspectOf(id.subject);
            case 1:
                return SomeSubjectPriceChangeAgent.aspectOf(id.subject);
            default:
                return null;
        }
    }
}
```

SomeSubject.java

```
public class SomeSubject {
    private int price;
    private String name;

    public String getName() {
        return name;
    }

    public int getPrice() {
        return price;
    }

    public void setName(String newName) {
        name = newName;
    }

    public void setPrice(int newPrice) {
        price = newPrice;
    }
}
```

SomeSubjectNameChangeAgent.java

```
import it.edu.jborella.patterns.observer.Agent;

public aspect SomeSubjectNameChangeAgent extends Agent {
    protected pointcut action() : call(public void SomeSubject.setName(String));
}
```

SomeSubjectPriceChangeAgent.java

```
import it.edu.jborella.patterns.observer.Agent;

public aspect SomeSubjectPriceChangeAgent extends Agent {
    protected pointcut action() : call(public void SomeSubject.setPrice(int));
}
```