Observer Pattern using Aspect-Oriented Programming[*]

Eduardo Kessler Piveta
Laboratório de Banco de Dados e Engenharia de Software (LBDES)
Centro Universitário Luterano de Palmas
Universidade Luterana do Brasil
77054-970, Palmas, TO
piveta@ulbra-to.br

Luiz Carlos Zancanella
Laboratório de Segurança em Computação (LabSEC)
Universidade Federal de Santa Catarina
Campus Universitário
88040-900 Florianópolis, SC
zancanella@inf.ufsc.br

Abstract

This paper discusses the representation and implementation of the Observer design pattern using aspect-oriented techniques.

1 Introduction

Several object-oriented design techniques have been used to specify and implement design patterns efficiently. However there are several patterns that affect system modularity, where base objects are highly affected by the structures that the pattern requires. In other cases we want to apply a pattern to classes that already belong to a hierarchy. This could be hard depending on the pattern.

[*]Copyright © 2003. Permission is granted to copy for the SugarloafPLoP 2003 Conference. All other rights are reserved.

Several patterns crosscut the basic structure of classes adding behavior and modifying roles in the classes relationship. As an example you could see the Observer pattern. When you have to implement the pattern, you should provide implementation to the Subject and Observer roles. The implementation of this roles adds a set of fields (`Subject.observers`, `Observer.subject`) and methods (`Attach`, `Detach`, `Notify`) to the concrete classes or to its superclasses, modifying the original structure of the subject and observer elements. Another example is the Visitor pattern. It's main goal is to provide a set of operations to a class hierarchy without changing the structure of the underlying classes. In order to accomplish that task, the pattern adds to the `Element` class a method (`Accept`) to allow the `Element`instances to be visited.

Althougth the use of these patterns brings several benefits, they could "hard-code" the underlying system, making difficult to express changes in the code. To implement one of the patterns described above in an envolving system you may have to change several classes, affecting their relationships and the clients of these classes.

Aspect Oriented Programming [Kiczales et al., 1997], [Ossher and Tarr, 2001] can help on separating some of the system's design patterns, specifying and implementing them as single units of abstraction.

The main goal of this paper is to show how the Observer [Gamma et al., 1995] pattern could be implemented using aspect-oriented programming and how it could be specified using an aspect oriented design model. We are going to discuss the benefits and disadvantages on using this approach.

2 Observer Pattern

2.1 Intent

It allows the definition of a "one to many" relationship between a model (Subject) and its dependents (Observers) in a way that promotes low coupling. Using aspect-oriented programming you could also attach and dettach the design pattern in compile-time or runtime (depending upon the choosen language)

2.2 Motivation

The problem that the Observer pattern solves is how to maintain consistency among several objects that depends on a model data in a way that promotes reuse, keeping a low coupling among classes. In this pattern, every time the Subject's state changes, all the Observers are notified.

The main problem with the object oriented Observer pattern is that you should modify the structure of classes that participate in the pattern. So, it's hard to apply the pattern into an existing design as well as remove it from the system.

2.3 Context

The Observer pattern is used in the following situations, according to [Gamma et al., 1995]:

- When a change in the state of an object demands modification in unknown or variable objects

- When an object needs to notify others without knowing whose are the objects that are going to receive this notification.

2.4 Structure

The design of the Observer pattern is changed in order to represent it as classes and aspects. It could be seen in the Figure 1, represented as a class diagram. There are no `Observer` role neither a `Subject` one. Both structure and behavior of these two roles are expressed in the `ObserverPattern` aspect.

2.5 Participants

Subject Describes the interface that all the concrete subjects must be in accordance to (enforced by the `ObserverPattern` and `ConcreteObserverPattern`). When implemented, the subject will contain a reference to its observers, and allow the dynamic addition and deletion of observers.

Observer Describes the interface that all the concrete observers must be in accordance to (enforced by the `ObserverPattern` and `ConcreteObserverPattern`). They are notified everytime the state of the subject changes.

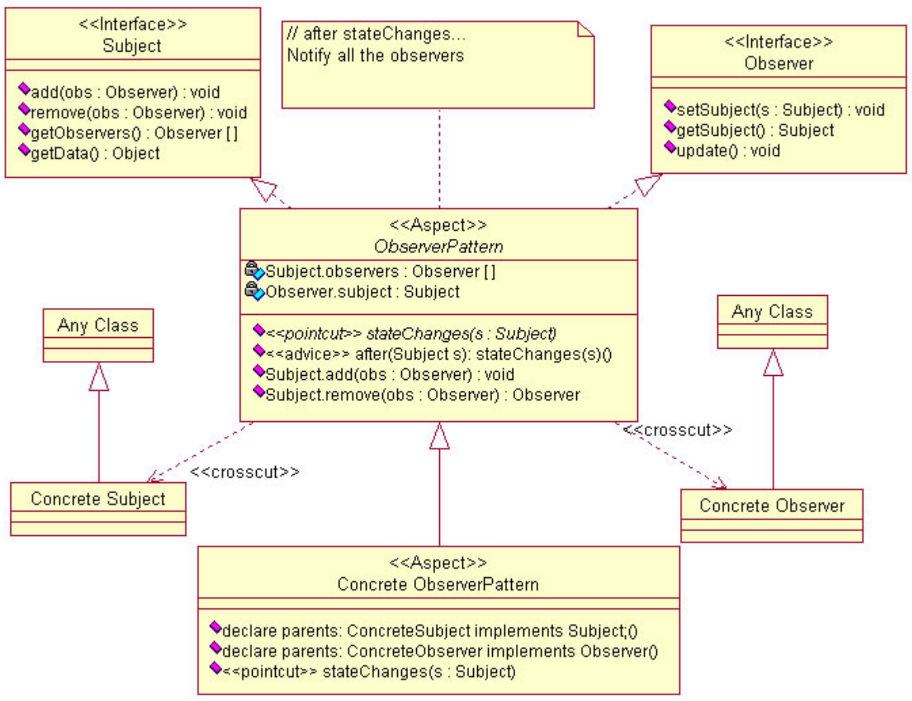
<<Interface>>
Subject
add(obs : Observer) : void
remove(obs : Observer) : void
getObservers() : Observer []
getData() : Object
// after stateChanges...
Notify all the observers
<<Interface>>
Observer
setSubject(s : Subject) : void
getSubject() : Subject
update() : void
<<Aspect>>
ObserverPattern
Subject.observers : Observer []
Observer.subject : Subject
<<pointcut>> stateChanges(s : Subject)
<<advice>> after(Subject s): stateChanges(s)()
Subject.add(obs : Observer) : void
Subject.remove(obs : Observer) : Observer
Any Class
Any Class
<<crosscut>>
<<crosscut>>
Concrete Subject
Concrete Observer
<<Aspect>>
Concrete ObserverPattern
declare parents: ConcreteSubject implements Subject;()
declare parents: ConcreteObserver implements Observer()
<<pointcut>> stateChanges(s : Subject)

Figure 1: Observer's class diagram

ConcreteSubject Store state information to be used by ConcreteObservers. It does not, however, send notifications to its Observers. This responsibility is part of the ObserverPattern role.[Gamma et al., 1995]

ConcreteObservers Servers as basis to field and method's introduction performed by the ObserverPattern.

ObserverPattern The ObserverPattern is an abstract aspect that encapsulte the behaviour of the Observer pattern. The `ObserverPattern` contains the fields and methods to be included in the classes that are affected by the `ConcreteObserverPattern`.

Concrete ObserverPattern This participant specifies in what situations the `ConcreteObservers` are going to be notified as well as what is going to be executed when the `ConcreteObservers` are notified.

2.6 Dynamics

When the classes corresponding to concrete subjects and concrete observers are weaved together with the concrete ObserverPattern, the following methods and fields are attached to the concrete subject and concrete observer:

fields `Observer[] observers` (Concrete Subject), `Subject subject` (Concrete Observers)

methods `void add(Observer obs)`, `void remove(Observer obs)` (Concrete Subject), `void setSubject(Subject s)`, `Subject getSubject()` (Concrete Observers)

These attachments are specified in the abstract aspect (the ObserverPattern). The concrete subjects implements the subject interface. The concrete observers implement the observer interface. The other modifications are done to the dynamic nature of the observer and subject classes, telling that every time that the state of the subject changes the update method of the observers is invocated.

2.7 Consequences

- The use of the Observer pattern allows to reuse subjects and observers in an independent way, since you can add new observers without change the subject or the others existing observers.
- Using an aspect-oriented implementation, the ObserverPattern could be reused without further modifications, as well as the classes affected by the pattern (that could be reused without considerations about the pattern).
- The use of aspect-oriented programming helped to separate the code related to the design pattern from the code of the base program itself.
- The use of an abstract aspect leads to a better reuse of the design pattern since it can be applied to several cases without changes. The developer should develop only the concrete aspect (as in this paper) to apply the pattern to the base code.
- Other consequences of the Observer pattern, as stated in [Gamma et al., 1995] are: abstract coupling among subjects and observers and support to a broadcast communication. One disadvantage on using this pattern is that the subject does not know how much costs an updating in all its observers.

- Another advantage is that the classes do not need to extend the subject and observer classes in an explicit way. The user attaches the fields and methods needed to implement the pattern.

2.8 Implementation

Some considerations could be made while implementing the Observer pattern:

- In order to send notification to its observers, the subject usually declares an explicit vector containing all its Observers. You could implement it as other data structure (such as: hash tables, linked lists etc) to solve performance or memory problems.
- `ObserverPattern` and `ObserverPatternImpl` could be mixed into one class. The reuse of the pattern in this case is limited.
- Other implementations issues can be found in [Gamma et al., 1995]

2.9 Example

In this section we are going to show an example on using `AspectJ` [Kiczales et al., 2001] to implement the Observer design pattern using a temperature domain. Suppose a set of thermometers that gather information from a temperature source. Each time that the source temperature changes the thermometers display should be updated.

An `Observer` interface is defined in order to describe the Observer role. All Observers must be in accordance to this interface. The idea here is to allow each Observer to have a corresponding Subject. This example does not treat multiple Subjects to one Observer.

```
interface Observer {
  void setSubject(Subject s);
  Subject getSubject();
  void update();
}
```

Figure 2: Observer Interface

In a similar way, we have a `Subject` interface (Figure 3), that all subjects must be in accordance to.

```
import java.util.Vector;
interface Subject {
  void add(Observer obs);
  void remove(Observer obs);
  Vector getObservers();
  Object getData();
}
```

Figure 3: Subject Interface

In Figure 4 we have a class called `Celsius` which is a source of temperature data. In this class is stored information about current temperature in Celsius. This information can be modified or retrieved using the `setDegrees` and `getDegrees` methods, respectively. This class is going to perform the `Subject` role. Note that there are not references to the `Subject` class or interface. Each time that the `setDegrees` method is invocated, the subjects are notified about changes in the temperature source state (this is going to be implemented as aspects).

```
public class Celsius{
  private double degrees;
  public double getDegrees(){
    return degrees;
  }
  public void setDegrees(double aDegrees){
    degrees = aDegrees;
  }
  Celsius(double aDegrees){
    setDegrees(aDegrees);
  }
}
```

Figure 4: Celsius class - The Subject

Another important class is the class that represents a thermometer (Figura 5). This class has a field called `tempSource` that points to a `Celsius` instance. This class is the base classe for the thermometers in the examples and is going to perform the observer role.

Extending the thermometer class we have a Celsius (Figure 6) and a Fahrenheit thermometer (Figure 7). These classes override the

```
public class Thermometer{
  private Celsius tempSource;
  public void setTempSource(Celsius atempSource){
    tempSource = atempSource;
  }
  public Celsius getTempSource(){
    return tempSource;
  }
  public void drawTemperature(){}
}
```

Figure 5: Thermometer class - The Observers superclass

`drawTemperature()` method providing different scales of temperature.

```
public class CelsiusThermometer extends Thermometer{
  public void drawTemperature(){
    System.out.println("Temperature in Celsius:"+
      getTempSource().getDegrees());
  }
}
```

Figure 6: The CelsiusThermometer class

In this example the `Celsius` class has the subject role and the thermometers classes are the observers. The reader could have already noted that `Celsius` class neither thermometers classes have explicit connection with the observer and subject classes.

The use of AspectJ allows the developer to separate the code related to the Observer pattern from the classes that use it.

In Figure 8 we have an abstract aspect called ObserverPattern that implements the basic functionality of the design pattern. This abstract aspect was retrieved from the AspectJ 1.1 examples and small modifications were made in the original structure.

This abstract aspect creates an abstract pointcut called `stateChanges` that will be extended in the concrete aspect to tell in what situations the observers are going to be notified. It implements an after advice that notifies all the observers when these points (situations) are reached.

It also create some fields (`Subject.observers` and `Observer.subject`) and methods (`Subject.add(..)`, `Subject.remove(..)`,

```
public class FahrenheitThermometer extends Thermometer{
  public void drawTemperature(){
    System.out.println("Temperature in Fahrenheit:"+
    (1.8 * getTempSource().getDegrees())+32);
  }
}
```

Figure 7: The FahrenheitThermometer class

`Subject.getObservers()`, `Observer.setSubject(..)`, `Observer.getSubject()`) in the classes that implements the `Subject` and `Observer` interfaces as implemented in Figure 8;

Here we extend the abstract aspect in order to tell to the compiler which classes are going to be treated as observers, as subjects and in which cases observers will be notified (the methods that are going to be executed when the notification happens are specified too). The sentences in `lines 3 and 5` bellow tells the compiler that the Celsius class implements the `Subject` interface and the `Thermometer` class implements the `Observer` interface. In `line 4` the method `getData()` is implemented in accordance with the `Subject` interface and in `line 6` the `update()` method is defined in order to accomplish the `Observer` interface.

The abstract pointcut defined in the `ObserverPattern` aspect is extended here defining that the observers are going to be notified every time the `setDegrees()` method is called.

2.10 Known Uses

Some common uses of this pattern (in the object-oriented form) could be found in [Gamma et al., 1995]. All the GoF patterns was already implemented in AspectJ and discussed in [Hannemann and Kiczales, 2002] with different levels of success.

3 Future Work

Future work will focus on finding common patterns in aspect-oriented programming that do not appear in object-oriented ones and on defining refactorings to aspect-oriented code.

```
import java.util.Vector;
abstract aspect ObserverPattern {
  abstract pointcut stateChanges(Subject s);
  after(Subject s): stateChanges(s) {
    for (int i = 0; i < s.getObservers().size(); i++)
      ((Observer)s.getObservers().elementAt(i)).update();
  }
  private Vector Subject.observers = new Vector();
  public void Subject.add(Observer obs) {
    observers.addElement(obs);
    obs.setSubject(this);
  }
  public void Subject.remove(Observer obs) {
    observers.removeElement(obs);
    obs.setSubject(null);
  }
  public Vector Subject.getObservers() { return observers; }
  private Subject Observer.subject = null;
  public void Observer.setSubject(Subject s) { subject = s; }
  public Subject Observer.getSubject() { return subject; }
}
```

Figure 8: Observer/Subject Protocol from AspectJ 1.1 examples

```
import java.util.Vector;
aspect ObserverPatternImpl extends ObserverPattern {
  declare parents: Celsius implements Subject;
  public Object Celsius.getData() { return this; }
  declare parents: Thermometer implements Observer;
  public void Thermometer.update() {
    drawTemperature();
  }
  pointcut stateChanges(Subject s): target(s) &&
    call(void Celsius.setDegrees(..));
}
```

References

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). Design patterns: Elements of reusable object-oriented software.

[Hannemann and Kiczales, 2002] Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in java and aspectj. In Ibrahim, M., editor, *Proc. 17th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA-2002).* ACM Press.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In Knudsen, J. L., editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin. Springer-Verlag.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag.

[Ossher and Tarr, 2001] Ossher, H. and Tarr, P. (2001). Hyper/J: Multi-dimensional separation of concerns for Java. In *Proc. 23rd Int'l Conf. on Software Engineering*, pages 729–730. IEEE Computer Society.