Observer Pattern using Aspect-Oriented Programming

Eduardo Kessler Piveta* and Luiz Carlos Zancanella
Departamento de Informática e Estatística
Universidade Federal de Santa Catarina
88040-900 Florianópolis, SC
{kessler,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.

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. Its 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. To implement one of the patterns described above in an envolving system you may have to modify several classes, affecting their relationships and their clients.

Aspect Oriented Programming [12] can help on separating some of the system's design patterns, specifying and implementing them as single units of abstraction.

*Professor at Centro Universitário Luterano de Palmas - TO, Brazil

The main goal is to show how the Observer [8] pattern could be implemented using aspect-oriented programming and how it could be specified using an aspect-oriented design model.

The intended audience of this paper is composed by object-oriented developers that do not have experience on aspect-oriented programming and by aspect-oriented developers aiming for implementations of design patterns based on crosscutting mechanisms.

In the following section, the core concepts of aspect-oriented programming (AOP) are discussed. Readers that are familiarized with AOP could go directly to the section that describes the pattern.

2 Aspect Oriented Programming

Software engineering and programming languages exist in a mutual relationship support. The most used design processes break a system down into a set of small units. To implement these units, programming languages provide mechanisms to define abstractions and composition mechanisms in order to implement the desired behavior [2].

A programming language coordinates well with a software design when the provided abstraction and composition mechanisms enable the developer to clearly express the design units. The most used abstraction mechanisms of languages (such as procedures, functions, objects, classes) are derived from the system functional decomposition and could be grouped into a generalized procedure model [12].

However, there are many properties that do not fit well into generalized procedures, such as: exception handling, real-time constraints, distribution and concurrency control. They are usually spread over into several system modules, affecting performance and/or semantics systematically.

When these properties are implemented using an object-oriented or a procedural language, their code is often tangled with the basic system functionality. It is hard to separate one concern from another, see or analyze them as single units of abstraction.

This code tangling is responsible by part of the complexity found in computer systems today. It increases the dependencies among the functional modules, deviating them from their original purposes, making them less reusable and error-prone.

This separation of concerns is a fundamental issue in software engineering and it is used in analysis, design and implementation of computer systems. However, the most used programming techniques do not always present themselves in a satisfactory way regarding to this separation.

Aspect-oriented programming allows separation of these crosscutting concerns, in a natural and clean way, using abstraction and composition mechanisms to produce executable code.

The aspect-oriented programming main goal is to help the developer in the task of clearly separate crosscutting concerns, using mechanisms to abstract and compose them to produce the desired system. The aspect-oriented programming extends other programming techniques (object oriented, structured, functional etc) that do not offer suitable abstractions to deal with crosscutting [12].

An implementation based on the aspect oriented programming paradigm is usually composed of:

- a component language to program components (i.e. classes);

- one or more aspect languages to program aspects;
- an aspect weaver to compose the programs written in these languages;
- programs written in the components language;
- one ore more programs written in the aspect language.

2.1 Components

Components (in AOP) are abstractions provided by a language to implement systems basic functionality. Procedures, function, classes and objects are components in aspect-oriented programming. They are originated from functional decomposition. The language used to express componnents could be, for instance, an object-oriented, an imperative or a functional one [16].

2.2 Aspects

Properties affecting several classes could not be well expressed using current notations and languages. They are usually expressed through code fragments that spread over the system classes [5].

Some concerns that are frequently aspects: concurrent objects synchronization [6], distribution [13], exception handling [15], coordination of multiple objects [10], persistence, serialization, replication, security, visualization, logging, tracing, load balance, fault tolerance amongst others.

2.3 Component language

The component language should provide developers with mechanisms to write programs implementing the basic requirements and also do not predict what is implemented in the aspects, (this property is called obliviousness [7]). Aspect-oriented programming is not limited to object orientation, although, the most used component languages are object oriented ones, such as: *Java*, *SmallTalk* or *C#*.

2.4 Aspect language

The aspect language defines mechanisms to implement crosscutting in a clear way, providing constructions describe the aspects semantics and behavior [12].

Some guidelines should be observed in the specification of an AO language: syntax should be related to the component language syntax (making easier the tools acceptance), the language should be designed to specify the aspect in a concise and compact way and the grammar should have elements to allow composition of classes and aspects [4].

2.5 Aspect Weaver

The aspect weaver main responsibility is to process aspect and component languages, in order to produce the desired operation. To do that, it is essential the *join-point* concept. A join-point is a well defined point in the execution or structure of a program. For instance,

in object-oriented programs join-points could be method calling, constructor calling, field read/write operations etc.

The representation of those points could be generated in runtime using a reflective environment. In this case, the aspect language is implemented through meta-objects, activated at method invocations, using join-points and aspects information to weave the arguments [14].

An aspect-oriented system design requires knowledge about what should be in classes and in aspects, as well as characteristics shared in both. Although aspect-oriented and object-oriented languages have different abstraction and composition mechanisms, they should use some common terms, allowing the weaver to compose the different programs.

The weaver parses aspect programs and collects information about the (join) points referenced by the program. Afterwards, it locates coordination points between the languages, weaving the code to implement what is specified in them [3].

An example of a weaver implementation is a pre-processor that traverse the classes parsing tree, looking for joint-points and inserting sentences declared in the aspects. This weaving process could be static (compile time) or dynamic (load and runtime).

3 Observer Pattern

3.1 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)

3.2 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 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 is hard to apply the pattern into an existing design as well as remove from it.

3.3 Context

The Observer pattern is used in the following situations, according to [8]:

- 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.

3.4 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.

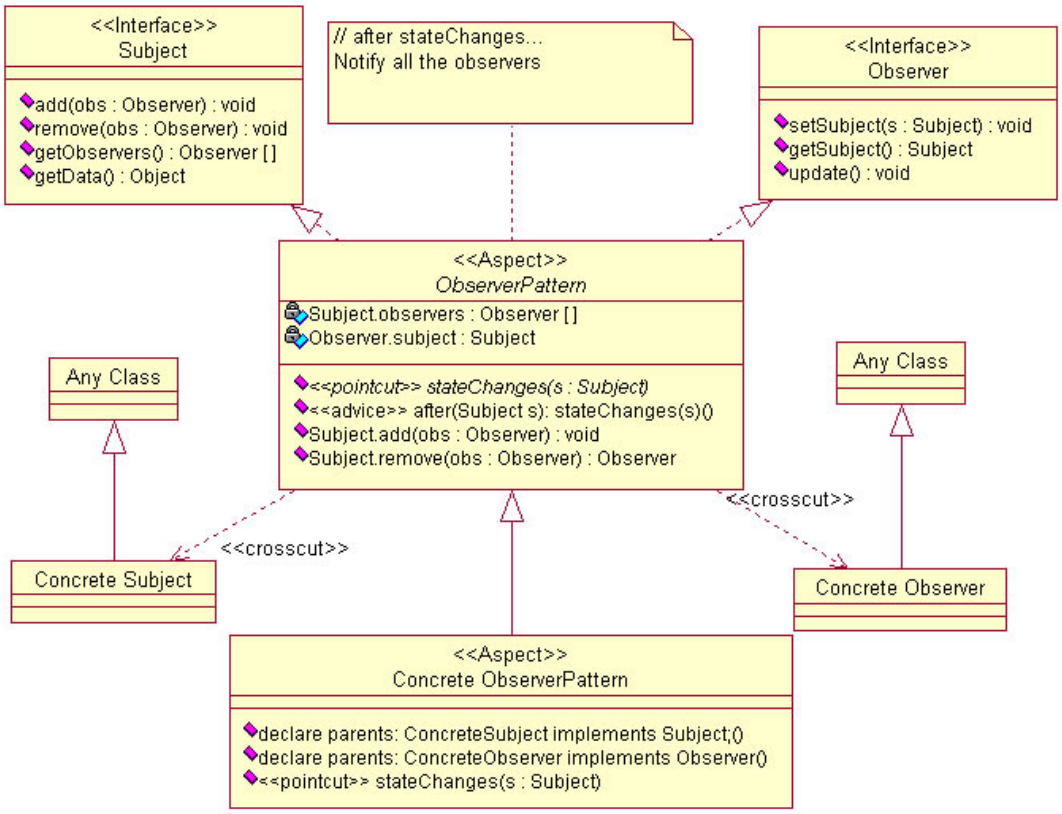

Figure 1: Observer's class diagram

3.5 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.

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. [8]

ConcreteObservers Serve 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.

3.6 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.

3.7 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 [8] 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.

3.8 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 [8]

3.9 Example with AspecJ

In this section we are going to show an example on using `AspectJ` [11] 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.

```
1  interface Observer {
2       void setSubject(Subject s);
3       Subject getSubject();
4       void update();
5  }
```

Figure 2: Observer Interface

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

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).

```
1  import java.util.Vector;
2  interface Subject {
3       void add(Observer obs);
4       void remove(Observer obs);
5       Vector getObservers();
6       Object getData();
7  }
```

Figure 3: Subject Interface

```
1  public class Celsius{
2       private double degrees;
3       public double getDegrees(){
4            return degrees;
5       }
6       public void setDegrees(double aDegrees){
7            degrees = aDegrees;
8       }
9       Celsius(double aDegrees){
10           setDegrees(aDegrees);
11      }
12 }
```

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.

```
1  public class Thermometer{
2       private Celsius tempSource;
3       public void setTempSource(Celsius atempSource){
4            tempSource = atempSource;
5       }
6       public Celsius getTempSource(){
7            return tempSource;
8       }
9       public void drawTemperature(){}
10 }
```

Figure 5: Thermometer class - The Observers superclass

Extending the thermometer class we have a Celsius (Figure 6) and a Fahrenheit thermometer (Figure 7). These classes override the `drawTemperature()` method providing different scales of temperature.

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.

```
1  public class CelsiusThermometer extends Thermometer{
2       public void drawTemperature(){
3         System.out.println("Temperature in Celsius:"+
4              getTempSource().getDegrees());
5       }
6  }
```

Figure 6: The CelsiusThermometer class

```
1  public class FahrenheitThermometer extends Thermometer{
2    public void drawTemperature(){
3         System.out.println("Temperature in Fahrenheit:"+
4         (1.8 * getTempSource().getDegrees())+32);
5    }
6  }
```

Figure 7: The FahrenheitThermometer class

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

```
1   import java.util.Vector;
2   abstract aspect ObserverPattern {
3        abstract pointcut stateChanges(Subject s);
4        after(Subject s): stateChanges(s) {
5             for (int i = 0; i < s.getObservers().size(); i++)
6               ((Observer)s.getObservers().elementAt(i)).update();
7        }
8        private Vector Subject.observers = new Vector();
9        public void Subject.add(Observer obs) {
10            observers.addElement(obs);
11            obs.setSubject(this);
12       }
13       public void Subject.remove(Observer obs) {
14            observers.removeElement(obs);
15            obs.setSubject(null);
16       }
17       public Vector Subject.getObservers() { return observers; }
18       private Subject Observer.subject = null;
19       public void Observer.setSubject(Subject s) { subject = s; }
20       public Subject Observer.getSubject() { return subject; }
21  }
```

Figure 8: Observer/Subject Protocol from AspectJ 1.1 examples

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 Cel-

sius 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.

```
1  import java.util.Vector;
2  aspect ObserverPatternImpl extends ObserverPattern {
3       declare parents: Celsius implements Subject;
4       public Object Celsius.getData() { return this; }
5       declare parents: Thermometer implements Observer;
6       public void Thermometer.update() {
7            drawTemperature();
8       }
9       pointcut stateChanges(Subject s): target(s) &&
10           call(void Celsius.setDegrees(..));
11  }
```

Figure 9: Concrete Observer protocol

3.10 Example using only Java

Consider an implementation of the Observer Pattern using Java, where we have an *Termometer* class playing the *Observer* role (Figure 10) and a *Clecius* class playing the *Subject* Role (Figure 11) .

```
1  public abstract class Termometer{
2       private Subject subject = null;
3       private Celcius tempSource;
4       // getter and setter methods
5       public abstract void drawTemperature();
6       public void update() {
7            drawTemperature();
8        }
9  }
```

Figure 10: Termometer Class

Note that there is an explicit reference to the subject.

In this implementation of the pattern, all the methods to add, remove and get observers are directly defined in the class playing the subject role. The concrete observers are not affected by this solution neither by the aspect-oriented one.

The diference between implementations is that in the OO code, the sentences related to the Observer pattern are tangled with the program basic functionalities. Although some of this problems could be solved using wrapper methods, reflection or changing the implementation language (smalltalk for instance), the use of aspect-oriented programming could still being another alternative to implement this pattern.

```
1  import java.util.Vector;
2  public class Celcius implements Subject{
3       private double degrees;
4       private Vector observers = new Vector();
5       public Object getData() { return this; }
6       public double getDegrees(){
7            return degrees;
8       }
9       public void setDegrees(double aDegrees){
10           degrees = aDegrees;
11            for (int i=0;i<getObservers().size();i++){
12                 ((Observer)getObservers().
13                 elementAt(i)).update();
14            }
15      }
16      public void add(Observer obs) {
17           observers.addElement(obs);
18           obs.setSubject(this);
19      }
20      public void remove(Observer obs) {
21           observers.removeElement(obs);
22           obs.setSubject(null);
23      }
24      public Vector getObservers()
25      { return observers; }
26      Celcius(double aDegrees){
27           setDegrees(aDegrees);
28      }
29 }
```

Figure 11: Celcius Class

3.11 Known Uses

Some common uses of this pattern (in the object-oriented way) could be found in [8]. All the GoF patterns was already implemented in AspectJ and discussed in [9] with different levels of success.

4 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.

5 Acknowledgements

We would like to thanks Deise Saccol and Thereza Padilha for their comments in early versions of this paper, to Federico Balaguer to the sheperding activities and to Robert Hammer, Joe Yoder, Paulo Borba and others in the group for their suggestions during the writing patterns sessions.

References

[1] *Workshop on Aspect Oriented Programming (ECOOP 1998)*, June 1998.

[2] Christian Becker and Kurt Geihs. Quality of service - aspects of distributed programs. In *Int'l Workshop on Aspect Oriented Programming (ICSE 1998)*, April 1998.

[3] K. Böllert. Aspect-oriented programming case study: System management application. In *Workshop on Aspect Oriented Programming (ECOOP 1998)* [1].

[4] Kai Böllert. On weaving aspects. In *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, June 1999.

[5] Krzystof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[6] John Dempsey and Vinny Cahill. Aspects of system support for distributed computing. In *Workshop on Aspect Oriented Programming (ECOOP 1997)*, June 1997.

[7] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October 2000.

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

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

[10] William Harrison and Harold Ossher. Subject-oriented programming—a critique of pure objects. In *Proc. 1993 Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, September 1993.

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

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[13] Cristina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[14] Anurag Mendhekar, Gregor Kiczales, and John Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL-97-009, Palo Alto Research Center, 1997.

[15] H. Ossher and P. Tarr. Operation-level composition: A case in (join) point. In *Workshop on Aspect Oriented Programming (ECOOP 1998)* [1].

[16] B. Tekinerdoğan and M. Akşit. Deriving design aspects from canonical models. In *Workshop on Aspect Oriented Programming (ECOOP 1998)* [1].