

A Position on an Aspect Oriented Approach to the Observer Pattern

Paul Freeman (pfreeman@ccs.neu.edu)

Abstract

An Aspect Oriented language that provides a means to capture points within a program's execution, such as AspectJ, can be used to implement a clean and well-separated Observer pattern. The Observer pattern, as described in Design Patterns [1], requires that ConcreteObserver classes implement a predefined Observer interface, and register themselves with a Subject object that will throw the events to be monitored. Previous work by Hannemann and Kiczales [2] illustrates how AspectJ can be used to produce a clean and well-separated strict translation of the Observer pattern. However, the nature of the Observer pattern itself creates an unnecessary coupling, due to the advent of languages such as AspectJ, between observable events.

By defining an Observer interface the programmer is in effect pulling the definitions of the events to be thrown out of the Subject class and placing them into a rigid interface that must be implemented in its entirety by all ConcreteObserver classes. Subsequent changes to the Observer interface must be propagated to all implementing classes, despite the fact that a particular ConcreteObserver class may not be affected by a change to the Observer interface.

The following proposed Aspect Oriented approach not only removes the need for Observer registration and management in the Subject object, it also pushes the interface definition back into the Subject class, and removes the requirement that all ConcreteObserver classes implement a rigid interface. A style is proposed where the Subject class defines each event explicitly, in effect registering the event as "available for listening". The behavior of the ConcreteObserver class that wishes to monitor a specific event is then abstracted to a ConcreteObserver aspect. The ConcreteObserver aspect need only use the Subject's event definition to apply advice to be executed around said event, effectively decoupling each event from the entire interface.

What follows is an example implementation of the Object Oriented (OO) Observer pattern followed by an implementation of the proposed Aspect Oriented (AO) approach, a brief discussion of another AO approach, as well as a brief comparison of the benefits and liabilities between the OO and AO Observer pattern approaches.

Observer Pattern

The intent of the Observer pattern is to "define a one-to-many dependency between objects so that when one object changes state, all it's dependents are notified and updated automatically" [1]. To accomplish this, a coupling is created between classes of Observer objects that implement the appropriate Observer interface and the class of Subject objects that defines the events to be observed. This coupling is desired since the goal here is to allow the Observer to monitor predefined events in the Subject. In the Object Oriented implementation however, because an interface is used to define the events that are to be monitored in the Subject object, an undesirable coupling is created *between all of the events* being observed. This coupling is undesirable as it is quite common in the practical use of the Observer pattern, to create ConcreteObserver classes that need only listen to some of the events defined in the Observer interface.

As an alternative, I propose an Aspect Oriented approach that decouples the events defined in the Observer interface. In fact, the Observer interface is removed all together and the events are instead defined directly in the Subject class through the use of AspectJ pointcuts. All Subject objects, i.e. instances of the Subject class, can then have their events observed through the use of the predefined pointcuts. Using this approach, a ConcreteObserver class is no longer required to implement the entire Observer interface. Instead, the behavior of the ConcreteObserver class is abstracted to a ConcreteObserver Aspect that observes only the events it is directly interested in.

A preferred side effect of this AO approach is discontinuation of the need to register ConcreteObserver objects with a Subject object, a requirement of the OO Observer Pattern. Instead, the Subject class effectively registers each event as “available for listening” through definition of an explicit pointcut for the event. Observer aspects may then declare advice that executes before, after, or around the pointcut. Observer registration now occurs at compile time.

One drawback to this AO approach is that observing aspects can’t be unregistered. Even though Aspect unregistration cannot be accomplished in the conventional manner, it is quite easy to place a boolean switch at the beginning of advice to control code execution (see Fig. 5) and effectively unregister an aspect.

Another drawback to the proposed AO approach is that ConcreteObserver aspects may need to filter reception of a single observable event that could have been raised by different Subject objects. Filtering events from different subject objects can be accomplished by passing a reference to the Subject, which threw the event, back to the observers. Filtering can also be achieved through strictly defining pointcuts so as to exclude Subject subclasses. However as filtering is generally undesirable, use of this pattern should be limited to instances where filtration can be kept to a minimum. For example, this pattern would be well suited to capturing “window open” events of a GUI system, but would not be recommended for use in a graphical drawing/editing tool that represents all graphical objects as observers of their corresponding models.

Object Oriented Approach

What follows is an illustration of how the OO implementation of the Observer pattern, using an Observer interface, creates an undesirable coupling between all of the events in the Subject object that are being monitored. Lets assume a class structure as described by the UML class diagram in Fig 1. The StartObserver class in the diagram only performs an action when the `subjectStarted(Subject)` method is called, yet it must also provide an implementation of the `subjectStopped(Subject)` method. A similar situation exists for the StopObserver class. Assuming the abstractions are correct, i.e. assuming it is not correct to combine the StartObserver code with the StopObserver code creating one Observer, the undesirable coupling is evidenced by the fact that both observers must implement and listen to a method they would just as soon ignore.

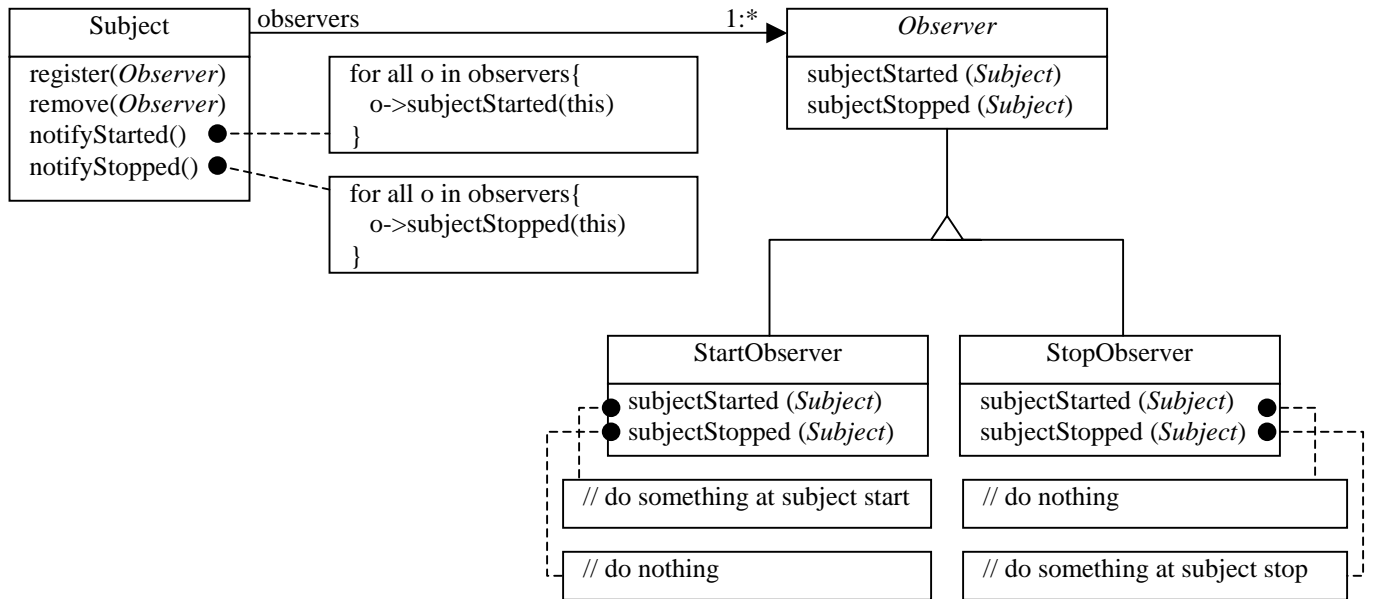


Fig. 1

The undesirable nature of the coupling becomes more evident when a simple change is made to the signature of one of the methods in the Observer interface. For example, adding a boolean parameter to the subjectStopped method would require the propagation of that change to both the StartObserver and StopObserver classes (see Fig. 2).

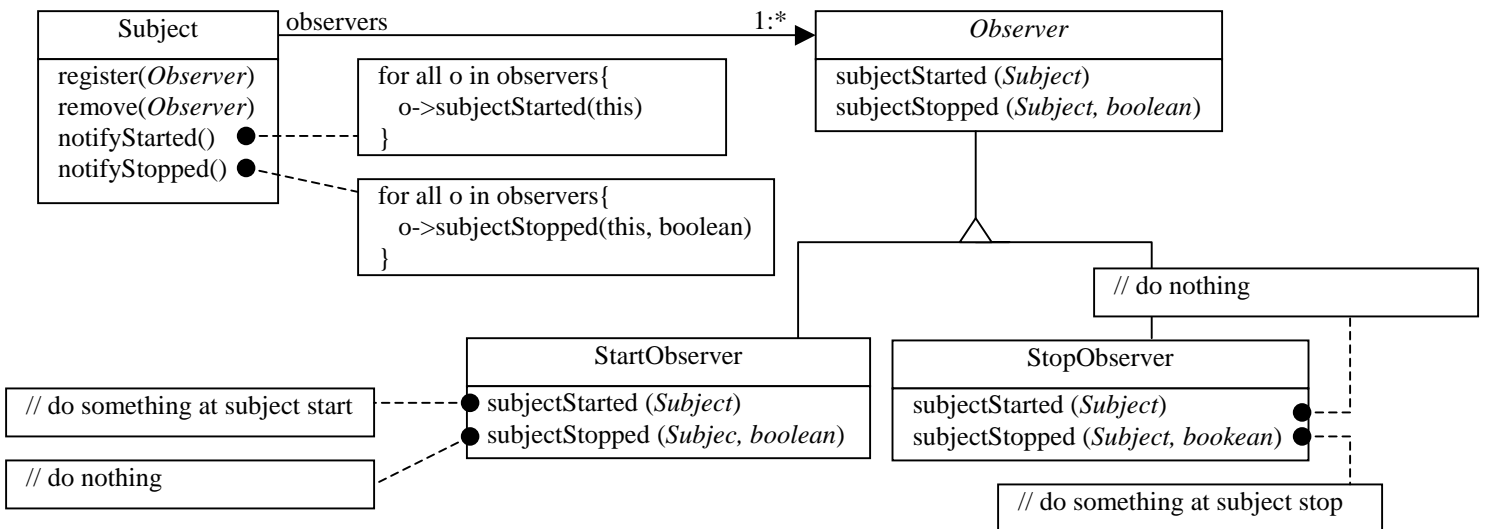


Fig. 2

Aspect Oriented Approach

An AO implementation of the observer pattern decouples the individual events in the Observer interface by removing the interface altogether. The interface, a requirement of an OO approach, was a description of the events in the Subject object that could be monitored. The purpose of the interface was to provide a means of notifying the dependent Observer objects that the state of the Subject object has changed. Using an AO language like AspectJ, the Subject class itself can define the individual events that are available for observation, removing the requirement of using an interface. The behavior of an observing object's class is abstracted to an Aspect, which can

then use the event definition, i.e. pointcut, to monitor the individual event. Figure 3 is a UML class diagram detailing the class structure for the proposed AO implementation of the previous Observer pattern problem. Note that the StartObserver and StopObserver Classes are now Aspects.

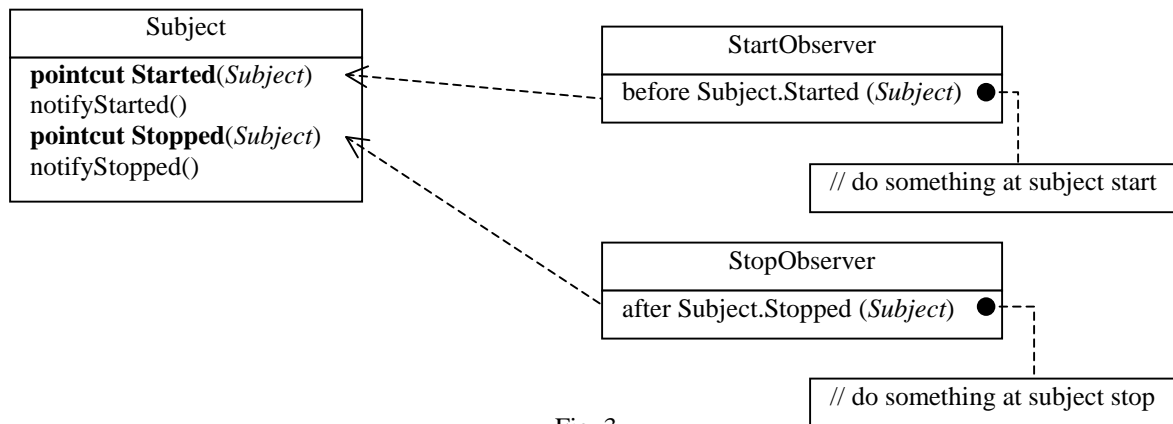


Fig. 3

As can be seen in Figure 3, the StartObserver and StopObserver aspects are no longer dependent on the same interface. Instead the advice within each Observer aspect is dependent only upon the pointcut definitions in the Subject class. Now, adding an addition Boolean parameter to the Stopped pointcut will have no affect on the StartObserver class (see Fig. 4).

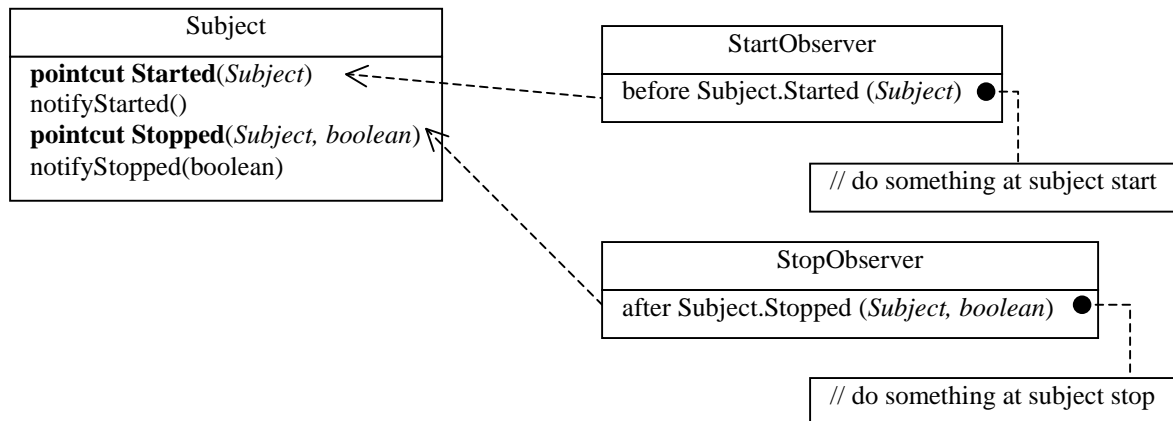


Fig. 4

It is apparent from the diagrams above that there is no longer a requirement to register each observing object with the Subject. Instead, registration of Observer aspects takes place at compile time as the advice is woven around the join points described by the pointcuts within the Subject class. In the OO approach, to stop an Observer object from listening to Subject events, the Observer would be unregistered through the “remove(observer)” method located in the Subject object (Fig. 1). In the AO approach, the Observer can never be “removed”, i.e. unregistered, however a boolean switch can easily be placed within the advice for a particular event to control code execution (see code in Fig 5).

What follows is a brief code implementation of the AO approach suggested by this paper. A class called Application implements the Subject class in the diagrams above. The StartObserver and StopObserver aspects have kept their respective names.

```

public class Application implements Runnable {

    public Application() {
    }

    @pointcut started(Application app): call(private void Application.notifyStarted()) && this(app);
    private void notifyStarted(){}

    @pointcut stopped(Application app): call(private void Application.notifyStopped()) && this(app);
    private void notifyStopped(){}

    public void run() {
        notifyStarted();
        System.out.println("Hello AO World");
        notifyStopped();
    }
}

```

```

privileged aspect StartObserver {

    private boolean listening = false;

    public void setListening(boolean listening){
        this.listening = listening;
    }

    before(Application app): Application.started(app){
        if(listening){
            System.out.println(this + ": " + app + " has started...");
        }
    }
}

```

```

privileged aspect StopObserver {

    private boolean listening = false;

    public void setListening(boolean listening){
        this.listening = listening;
    }

    before(Application app): Application.stopped(app){
        if(listening){
            System.out.println(this + ": " + app + " has stopped.");
        }
    }
}

```

Fig. 5

Alternative AO Observer Patterns

Hanneman and Kiczales [2] performed a strict translation of the OO Observer pattern to an AO Observer pattern. Their approach uses AspectJ to effectively modularize the inherent concerns of the Observer pattern, while also effectively making the pattern “pluggable”, i.e. compilation with or without their proposed aspects adds or removes the observer pattern from the program. “Pluggability” can also be achieved through the approach proposed in this paper simply by choosing to compile or not to compile with the various ConcreteObserver aspects.

The approach proposed by Hanneman and Kiczales [2] is usable in a layered system, since it uses “register” and “remove” functions to add ConcreteObserver objects to the list of objects listening to a particular Subject. The approach described in this paper is limited to use in a single program compilation, i.e. one layer, at least until byte code compilation becomes available in AspectJ 1.1. Once byte code compilation is available, the pattern proposed by this paper should be usable in a layered system.

Hannemann and Kiczales [2] do not address the focus of this paper however. The code in their paper describes one general event, “subjectChange”, that can be monitored by the abstract aspect ObserverProtocol. Following the pattern of their code, to monitor an additional event one would be required to add another pointcut, abstract method, and piece of advice to the abstract aspect ObserverProtocol, effectively providing the same undesired coupling of individual events as was evidenced through the use of an Observer interface.

Benefits and Liabilities

What follows is a brief comparison of the benefits and liabilities of the Object Oriented (OO) approach, as described in Design Patterns [1], to the benefits and liabilities of the Aspect Oriented (AO) approach described above.

1. Both approaches allow you to vary subjects and observers independently. Observers can be added without modifying the Subject or other observers. You can also reuse subjects without reusing their observers. In the AO pattern proposed by this paper however, adding, or subclassing, Subjects may require modification of Observer advice in instances where Subject event filtering needs to occur.
2. Both approaches provide an abstract coupling between the Subject and Observer classes. However, the AO approach is more abstract than the OO approach. In the OO approach, the subject is aware of the observers that are registered to observe it and the observers are aware of the subject at the time of registration. Alternatively in the proposed AO approach, the subject has no knowledge what-so-ever of the classes observing it, but the observers are directly aware of the subject through reference of the subject’s pointcuts.

In the OO approach, the Subject and Observer can belong to different layers of abstraction in a system. If the AO approach is implemented using a language such as AspectJ, which currently does not provide for byte code manipulation, the Subject and Observer classes must be compiled together. This limitation should be eliminated with the addition of byte-code compilation in AspectJ 1.1.

3. Both approaches support broadcast communication. The notification sent by a subject does not need to specify the receiving, i.e. observing, object. The subject does not care how many or which observers are monitoring it, it is only concerned with notifying any observers that are monitoring it of specific events. This can be a drawback to the proposed AO approach, as the AO approach may require filtering events from various Subject objects. Filtering is unnecessary in the OO approach as a list of observing objects is maintained in the individual Subject objects.
4. Both approaches have the liability of producing unexpected behavior. Since observers have no knowledge of each other, actions taken at the occurrence of an event may have an unintended affect on the behavior of other observers.

Conclusion

AspectJ, or another similar language, can be used to improve upon the Observer Pattern [1] by removing the Observer interface, effectively decoupling the individual events described by the interface from each other. The decoupling of the Observer interface events minimizes required code modification due to changes during the development process, reduces overall code size, and improves code readability, as it is no longer necessary to implement empty Observer interface methods.

The AO Observer Pattern suggested by this paper improves the OO pattern even further by abstracting the behavior of Observer classes into individual ConcreteObserver aspects. Abstracting Observer object behavior to ConcreteObserver aspects makes Observer registration and maintenance by Subject objects no longer necessary, reducing a small amount of overhead, overall code size and code complexity.

As with most AO approaches to design, the improvements to the Observer Pattern suggested by this paper trend towards general code reduction and code localization. Implementation of a simpler Observer Pattern should improve development time by reducing overall code size and subsequently the number of errors inserted during the development process.

The drawbacks to this pattern may be significant in instances where filtering of Subject events is required. In cases such as this, the pattern proposed by Hennemann and Kiczales [1] should be used.

Acknowledgments

I would like to thank Dr. Karl Lieberherr for his guidance and encouragement. This paper was inspired by observations made during the development of an Aspect Oriented project in the graduate level Software Engineering course taught by Dr. Lieberherr at Northeastern University.

References

- [1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns," Addison-Wesley (1995), Chapter 5, pp 293 – 303
- [2] Jan Hennemann and Gregor Kiczales, "Design Pattern Implementation in Java and AspectJ", Proceedings of OOPSLA 2002