CHAPTER 17

The Observer Pattern

Overview

This chapter continues the e-tailing case study discussed in Chapters 14-16.

In this chapter

In this chapter,

- I introduce the categorization scheme of patterns.
- I introduce the Observer pattern by discussing additional requirements for the case study.
- I apply the Observer pattern to the case study.
- I describe the Observer pattern.
- I describe the key features of the Observer pattern.
- I describe some of my experiences using the Observer pattern in practice.

Categories of Patterns

There are many patterns to keep track of. To help sort this out, the Gang of Four has grouped patterns into three general categories, as shown in Table 17-1.[1]

The GoF has three categories

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, Mass.: Addison-Wesley, 1995, p. 10.

Table 17-1 Categories of Patterns

Category	Purpose	Examples in This Book	Use For
Structural	Bring together existing objects	Facade (Chapter 6) Adapter (Chapter 7)	Handling interfaces
		Bridge (Chapter 9) Decorator (Chapter 15)	Relating implementations to abstractions
Behavioral	Give a way to manifest flexible (varying) behavior	Strategy (Chapter 14)	Containing variation
Creational	Create or instantiate objects	Abstract Factory (Chapter 10) Singleton (Chapter 16) Double-Checked Locking (Chapter 16) Factory Method (Chapter 19)	Instantiating objects

A note on the classification of the Bridge and Decorator patterns

When I first started studying design patterns, I was surprised to see the Bridge and Decorator patterns were structural patterns rather than behavioral patterns. After all, they seemed to be used to implement different behaviors. At the time, I simply did not understand the GoF's classification system. Structural patterns are for tying together existing function. In the Bridge pattern, we typically start with abstractions and implementations and then bind them together with the bridge. In the Decorator pattern, we have an original functional class, and want to decorate it with additional functions.

My "fourth" category: decoupling

I have found it valuable to think of a fourth category of patterns, one whose primary purpose is to decouple objects from each other. One motivation for these is to allow for scalability or increased flexibility. I call this category of patterns *decoupling patterns.* Since most of the patterns in the decoupling category belong to the Gang of Four's behavioral category, I could almost call them a subset of the behavioral category. I chose to make a fourth category simply because my intent in this book is to reflect how I look at patterns, focusing on their motivations—in this case, decoupling.

I would not get too hung up on the whys and wherefores of the classifications. They are meant to give insights into what the patterns are doing.

This chapter discusses the Observer pattern, which is the best example of a decoupling pattern pattern there is. The Gang of Four classifies Observer as a Behavioral pattern.

Observer is a decoupling (behavioral) pattern

More Requirements for the Case Study

In the process of writing the application, suppose I get a new requirement to take the following actions whenever a new customer is entered into the system:

New requirement: take actions for new customers

- Send a welcome e-mail to the customer.
- Verify the customer's address with the post office.

Are these all of the requirements? Will things change in the future?

One approach

If I am reasonably certain that I know every requirement, then I could solve the problem by hard-coding the notification behavior into the Customer class, such as shown in Figure 17-1.

F

or example, using the same method that adds a new customer into the database, I will also make calls to the objects that generate welcome letters and verify post office addresses.

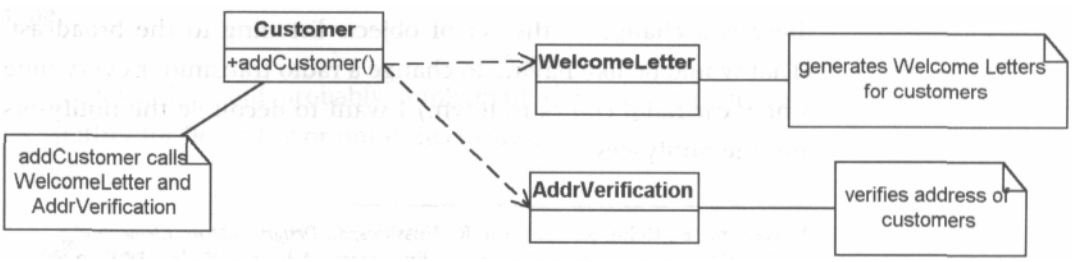

Figure 17-1 Hard-coding the behaviors

These classes have the following responsibilities:

Class	Responsibility
Customer	When a customer is added, this object will make calls to the other objects to have the corresponding actions take place.
WelcomeLetter	Creates welcome letters for customers that let them know they were added to the system.
AddrVerification	This object will verify the address of any customer that asks it to.

The problem? Requirements always change

The hard-coding approach works fine—the first time. But requirements *always* change. I know that another requirement will come that will require another change to Customer's behavior. For example, I might have to support different companies' welcome letters, which would require a different Customer object for each company. Surely, I can do better.

The Observer Pattern

The intent, according to the Gang of Four

According to the Gang of Four, 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."[2]

What this means: Handling notification automatically

Often, I have a set of objects that need to be notified whenever an event occurs. I want this notification to occur automatically. However, I do not want to change the broadcasting object everytime there is a change to the set of objects listening to the broadcast. (That would be like having to change a radio transmitter every time a new car radio comes to town.) I want to decouple the notify-ors and the notify-ees.

2. Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, Mass.: Addison-Wesley, 1995, p. 293.

This pattern is a very common one. It also goes by the names *Dependents* and *Publish-Subscribe?* and is analogous to the notify process in COM. It is implemented in Java with the Observer interface and the Observable class (more on these later). In rule-based, expert systems, they are often implemented with daemon rules.

A common pattern

Applying the Observer to the Case Study

My approach is to look in the problem for clues as to what is varying. Then, I attempt to encapsulate the variation. In the current case, I find:

Two things are varying

- **Different kinds of objects**—There is a list of objects that need to be notified of a change in state. These objects tend to belong to different classes.
- **Different interfaces**—Since they belong to different classes, they tend to have different interfaces.

First, I must identify all of the objects that want to be notified. I will call these the *observers* since they are waiting for an event to occur.

Step 1: Make the observers behave in the same way

I want all of the observers to have the same interface. If they do not have the same interface, then I would have to modify the *subject*—that is, the object that is triggering the event (for example, Customer), to handle each type of observer.

By having all of the observers be of the same type, the subject can easily notify all of them. To get all of the observers to *be* of the same type,

- In Java, I would probably implement this with an interface (either for flexibility or out of necessity).

3. ibid, p. 293.

- In C++, I would use single inheritance or multiple inheritance, as required.

Step 2: Have the observers register themselves

In most situations, I want the observers to be responsible for knowing what they are to watch for and I want the subject to be free from knowing which observers depend on it. To do this, I need to have a way for the observers to register themselves with the subject. Since all of the observers are of the same type, I must add two methods to the subject:

- *attach* (Observer)—adds the given Observer to its list of observers
- *detach* (Observer)—removes the given Observer from its list of observers

Step 3: Notify the observers when the event occurs

Now that the Subject has its Observers registered, it is a simple matter for the Subject to notify the Observers when the event occurs. To do this, each Observer implements a method called update. The Subject implements a *notify* method that goes through its list of Observers and calls this update method for each of them. The update method should contains the code to handle the event.

Step 4: Get the information from the subject

But notifying each observer is not enough. An observer may need more information about the event beyond the simple fact that it has occurred. Therefore, I must also add method(s) to the subject that allow the observers to get whatever information they need. Figure 17-2 shows this solution.

How this works

In Figure 17-2, the classes relate to each other as follows:

1. The Observers attach themselves to the Customer class when they are instantiated. If the Observers need more information from the subject (Customer), the update method must be passed a reference to the calling object.

2. When a new Customer is added, the *notify* method calls these Observers.

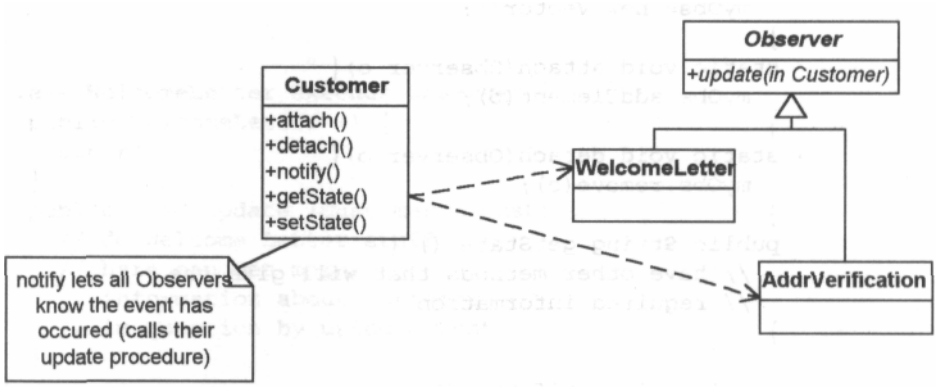

Figure 17-2 implementing Customer with Obs erver.

Each observer calls *getstate* for information on the newly added Customer to see what it needs to do. *Note:* Typically, there would be several methods to get the needed information.

Note in this case, we use static methods for attach and detach because observers want to be notified for all new Customers. When notified, they are passed the reference to the Customer created.

Example 17-1 shows some of the code required to implement this.

Observer aids flexibility and keeps things decoupled

This approach allows me to add new Observers without affecting any existing classes. It also keeps everything loosely coupled. This organization works if I have kept all of the objects responsible for themselves.

New requirement: send coupons, too

How well does this work if I get a new requirement? For example, what if I need to send a letter with coupons to customers located within 20 miles of one of the company's "brick and mortar" stores.

Example 17-1 Java Code Frament: Observer Implemented

```
class Customer {
  static private Vector myObs;
  static {
    myObs= new Vector();
  }
  static void attach(Observer
    o){ myObs.addElement(o);
  }
  static void detach(Observer
    o){ myObs . remove (o) ;
  }
  public String getState () {
    // have other methods that will give the
    // required information }

  public void notifyObs () { for
    (Enumeration e =
    myObs.elements();
    e.hasMoreElements() ;) {
   ((Observer) e).update(this); } } }


abstract class Observer
  { public Observer () {
    Customer.attach( this);
  }
  abstract public void
    update(Customer myCust); }

class POVerification extends Observer
  { public AddrVerification () { super();
  }
  public void update
    ( Customer myCust) {
```

(continued)

Example 17-1 Java Code Frament: Observer Implemented *(continued)*

```
If do Address verification stuff here // can get
more information about customer // in question by
using myCust } }

class WelcomeLetter extends Observer
  { public WelcomeLetter (} { super();
  }
  public void update (Customer myCust) {
    // do Welcome Letter stuff
    // here can get more
    // information about customer
  // in question by using myCust } }
```

To accomplish this, I would simply add a new observer that sends the coupon. It only does this for new customers living within the specified distance. I could name this observer BrickAndMortar and make it an observer to the Customer class. Figure 17-3 shows this solution.

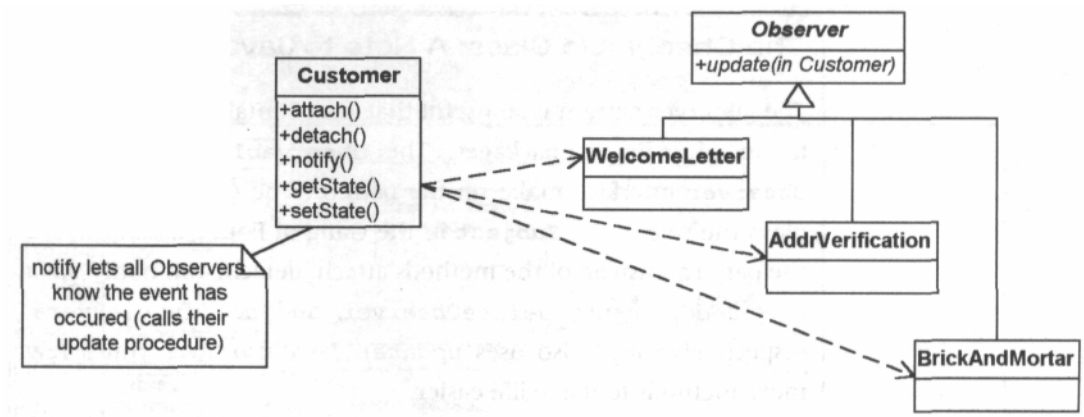

Figure 17-3 Adding the BrickAndMortar observer.

The **Observer** ***in*** *the real world*

Sometimes, a class that will become an Observer may already exist. In this case, I may not want to modify it. If so, I can easily adapt it with the Adapter pattern. Figure 17-4 shows an example of this.

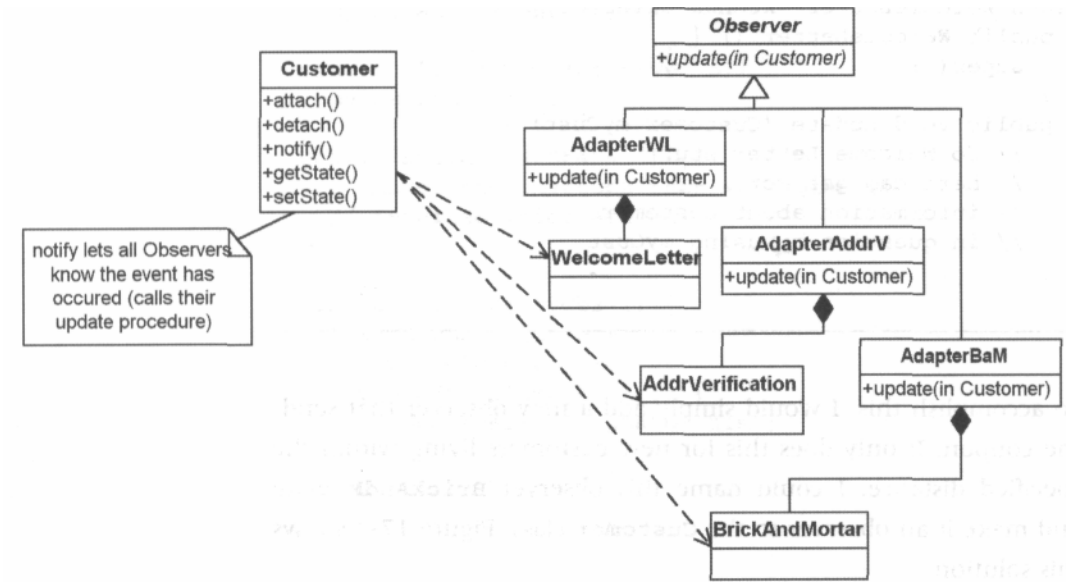

Figure 17-4 Implementing Observer **with** Adapters.

The Observable Class: A Note to Java developers.

The Observer pattern is so useful that Java contains an implementation of it in its packages. The Observable class and the Observer interface make up the pattern. The Observable class plays the role of the Subject in the Gang of Four's description of the pattern. Instead of the methods attach, detach, and notify, Java uses *addObserver*, *deleteObserver*, and *notifyObservers*, respectively (Java also uses update). Java also gives you a few more methods to make life easier.*

See *http://java.sun.com/j2se/l3tdocs/api/index.html* for information on the Java API for Observer and Observable.

The Observer Pattern: Key Features

Intent	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Problem	You need to notify a varying list of objects that an event has occurred.
Solution	Observers delegate the responsibility for monitoring for an event to a central object: the Subject.
Participants and Collaborators	The Subject knows its Observers because the Observers register with it. The Subject must notify the Observers when the event in question occurs. The Observers are responsible both for registering with the Subject and for getting the information from the Subject when notified.
Consequences	Subjects may tell Observers about events they do not need to know if some Observers are interested in only a subset of events (see "Field Notes: Using the Observer Pattern" on page 274). Extra communication may be required if Subjects notify Observers which then go back and request additional information.
Implementation	• Have objects (Observers) that want to know when an event happens attach themselves to another object (Sub j ect) that is watching for the event to occur or that triggers the event itself. • When the event occurs, the Subject tells the Observers that it has occurred. • The Adapter pattern is sometimes needed to be able to implement the Observer interface for all of the Observer-type objects.
GoF Reference	Pages 293-303.

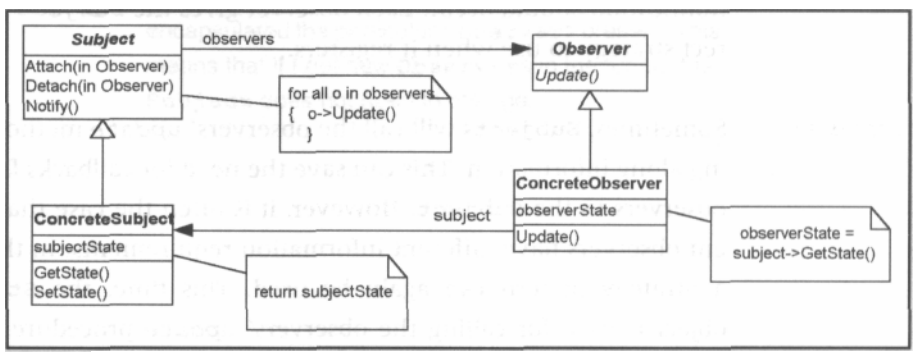

Figure 17-5 Standard, simplified view of the Observer pattern.

Field Notes: Using the Observer Pattern

Not for all dependencies

The Observer pattern is not meant to be used every time there is a dependency between objects. For example, in a ticket processing system a tax object handles taxes, it is clear that when items are added to the ticket the tax object must be notified so the tax can be recalculated. This is not a good place for an Observer pattern since this notification is known up front and others are not likely to be added. When the dependencies are fixed (or virtually so), adding an Observer pattern probably just adds complexity.

... but for changing or dynamic dependencies

If the list of objects that need to be notified of an event changes, or is somehow conditional, then the Observer pattern has greater value. These changes can occur either because the requirements are changing or because the list of objects that need to be notified are changing. The Observer pattern can also be useful if the system is run under different conditions or by different customers, each having a different list of required observers.

Whether to process an event

An observer may only need to handle certain cases of an event. The Brick and Mortar case was an example. In such situations, the observer must filter out extra notifications.

Extraneous notifications can be eliminated by shifting the responsibility for filtering out these notifications to the Subject. The best way to do this is for the Subject to use a Strategy pattern to test if notification should occur. Each observer gives the Subject the correct strategy to use when it registers.

How to process an event

Sometimes, Subjects will call the observers' update method, passing along information. This can save the need for callbacks from the observers to the Subject. However, it is often the case that different observers have different information requirements. In this case, a Strategy pattern can again be used. This time, the Strategy object is used for calling the observers' update procedure. Again,

the observers must supply the Subject with the appropriate Strategy object to use.

Summary

In this chapter

In learning the Observer pattern, I looked at which object is best able to handle future variation. In the case of the Observer pattern, the object that is triggering the event—the Subject—cannot anticipate every object that might need to know about the event. To solve this, I create an Observer interface and require that all Observers be responsible for registering themselves with this Subject.

Summary of object-oriented principles used

While I focused on the Observer pattern during the chapter, it is worth pointing out several object-oriented principles that are used in the Observer pattern.

Concept	Discussion
Objects are responsible for themselves	There were different kinds of Observers but all gathered the information they needed from the Subj ect and took the action appropriate for them on their own.
Abstract class	The Observer class represents the concept of objects that needed to be notified. It gave a common interface for the subject to notify the Observers.
Polymorphic encapsulation	The subject did not know what kind of observer it was communicating with. Essentially, the Observer class encapsulated the particular Observers present. This means that if I get new Observers in the future, the Subject does not need to change.

Supplement: C++ Code Example

Example 17-2 C++ Code Fragment

```
class Customer {

 public:
  static void attach(Observer *o);
  static void detach(Observer *o);
  String getStateO;
 private:
  Vector myObs;
  void notifyObs();
}

Customer::attach(Observer *o){
  myObs.addElement(o); }
Customer::detach(Observer *o){
  myObs.remove(o);
}
Customer::getState () {
    // have other methods that will //
    give the required information
}

Customer::notifyObs () { for
    (Enumeration e =
    myObs.elements();
    e.hasMoreElements() ;)
    { ((Observer *) e)->
         update(this); }

  }
)

class Observer { public: Observer();
 void update(Customer *mycust)=0;
    // makes this abstract }
```

(continued)

Example 17-2 C++ Code Fragment *(continued)*

```
Observer::Observer () {
    Customer.attach( this);
}
class AddrVerification : public Observer
 { public:
  AddrVerification() ;
  void update( Customer *myCust); }

AddrVerification::AddrVerification () {
}
AddrVerification::update
   (Customer *myCust) {
     // do Address verification stuff here
     // can get more information about
     // customer in question by using myCust
}

class WelcomeLetter : public Observer { public:
  WelcomeLetter(};
  void update( Customer *myCust); }

WelcomeLetter::update( Customer *myCust) {
     // do Welcome Letter stuff here can get more //
     information about customer in question by //
     using myCust
}
```