Keeping Your Java Objects Informed with the Observer Design Pattern

by Barry A. Burd and Michael P. Redlich

This article, the second in a series on design patterns, introduces the Observer pattern, one of the 23 design patterns defined in the legendary 1995 book *Design Patterns – Elements of Reusable Object-Oriented Software*. The authors of this book, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, are known affectionately as the "Gang of Four." (So popular is this book that the Gang of Four has its own acronym – GoF.)

Design Patterns

The GoF book defines 23 design patterns. The patterns fall into three categories:

- A creational pattern abstracts the instantiation process.
- A structural pattern groups objects into larger structures.
- A behavioral pattern defines better communication among objects.

The Observer design pattern fits into the behavioral category. Like the Decorator design pattern, the Observer pattern is one of the most widely used patterns. As you read this article, you may recognize the Observer pattern as an important element of the Java Standard API.

The Observer Pattern

The description of a pattern has four parts:

- The pattern's intent,
- The pattern's motivation,
- The pattern's implementation, and
- The consequences of the pattern's use.

Intent

We again reference the GoF book to describe the intent of the Observer design pattern:

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

Some authors call observers "dependents" because of the one-to-many dependency relationship. Other authors refer to observers as "subscribers" because one object publishes notifications and one or more observer objects subscribe to the notifications. All these points of view can be helpful, so in this article, we use the terms "observer," "subscriber," and "dependent" interchangeably.

Motivation

The motivation for the Observer design pattern is to avoid tight coupling. For example, consider a stock quote application. The application sends quotes to its subscribers. A first attempt to write the stock quote application might look like this:

```
public class StockData
      {
      private String symbol; // the stock symbol
      private float close; // the closing stock price
      private float high; // the high stock price for the day
      private float low; // the low stock price for the day
      private int volume; // the amount of stocks traded

      // getter and setter methods defined here...

      public void sendStockData(String symbol,float close,float high,float low,long volume)
            {
            symbol = getSymbol();
            close = getClose();
            high = getHigh();
            low = getLow();
            volume = getLow();
            tradingFool.update(symbol,close,high,low,volume);
            bigBuyer.update(symbol,close,high,low,volume);
            }
      }
```

Here, the code assumes that **tradingFool** and **bigBuyer** are just two of potentially many class instances interested in obtaining the latest stock data. To analyze this code, it uses a single method, **update()**, with all interested parties as shown below:

```
tradingFool.update(symbol,close,high,low,volume);
bigBuyer.update(symbol,close,high,low,volume);
```

The trouble is, you'll eventually want to add additional parties to the **sendStockData()** method:

```
stockBoy.update(symbol,close,high,low,volume);
```

To add **stockBoy**, you have to recompile the **StockData** class. That's not good. The **StockData** class shouldn't be so tightly coupled to things like **stockBoy** and **bigBuyer**. Instead, **StockData** should be ready to provide information to anyone (well, to a large, ever-changing group of "anyones").

Implementation

Figure 1 shows the official UML diagram for the Observer pattern.

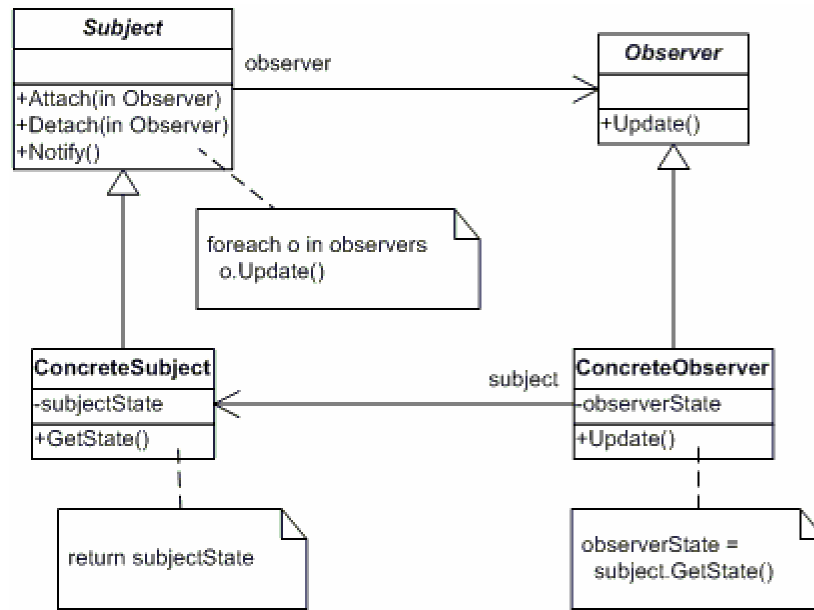

Figure 1: The official UML diagram for the Observer design pattern

Here's a walk-through of the parts of the UML diagram.

- The **`ConcreteSubject`** class implements the **`Subject`** interface.
- The **`Subject`** interface declares the methods for observers to register and remove themselves with a **`ConcreteSubject`**. The interface also contains a method to notify all registered observers. A particular subject can have many observers.
- The **`Observer`** interface declares a method for obtaining updates from the **`Subject`**. The subject publishes an update when its state changes.
- The **`ConcreteObserver`** class implements the **`Observer`** interface.
- Each **`ConcreteObserver`** object has its own instance of the **`ConcreteSubject`** class. The **`subject`** variable (in the **`ConcreteObserver`** class) refers to this instance.

In the ongoing stock quote example, each concrete observer (**`TradingFool`**, **`BigBuyer`**, and **`StockBoy`**) implements the **`update()`** method declared in the **`Observer`** interface. The **`ConcreteSubject`** calls each observer's **`update()`** method, sending updated information to each concrete observer. In turn, each concrete observer uses the updated information in whatever way it sees fit.

Figure 2 shows a refactored UML diagram for the stock quotes application using the Observer design pattern.

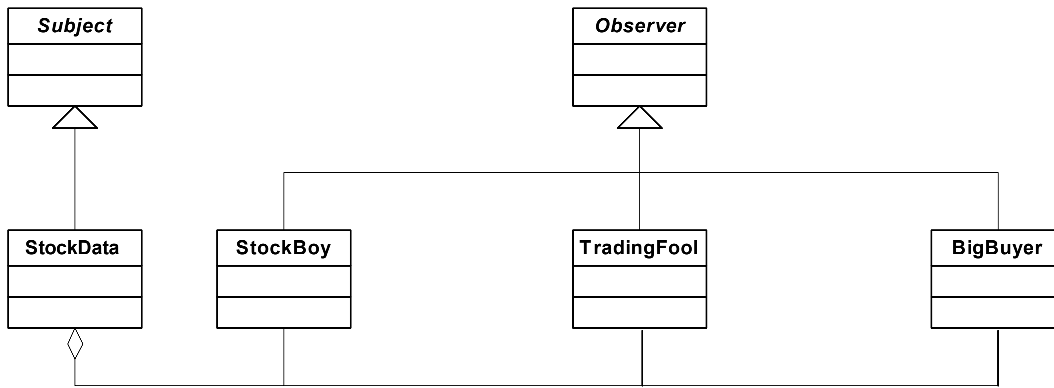

Figure 2: The refactored UML diagram for the stock quote application.

The source code for this refactored diagram is in Listings 1 to 5, and the client code is in Listing 6.

```
public interface Subject
      {
      public void attach(Observer observer);
      public void detach(Observer observer);
      public void notifyObservers();
      }
```

Listing 1: The Subject interface.

```
public class StockData implements Subject
      {
      private String symbol;
      private float close;
      private float high;
      private float low;
      private long volume;
      private List<Observer> observers;

      public StockData()
            {
            observers = new ArrayList<Observer>();
            }

      public String getSymbol()
            {
            return symbol;
            }

      public float getClose()
            {
            return close;
            }

      public float getHigh()
            {
            return high;
            }
```

```
	public float getLow()
		{
		return low;
		}

	public long getVolume()
		{
		return volume;
		}

	public void attach(Observer observer)
		{
		observers.add(observer);
		}

	public void detach(Observer observer)
		{
		int i = observers.indexOf(observer);
		if(i >= 0)
			observers.remove(i);
		}

	public void notifyObservers()
		{
		int n = observers.size();
		for(int i = 0;i < n;++i)
			{
			Observer observer = (Observer)observers.get(i);
			observer.update(symbol,close,high,low,volume);
			}
		}

	public void sendStockData()
		{
		notifyObservers();
		}

	public void setStockData(String symbol,float close,float high,float low,long volume)
		{
		this.symbol = symbol;
		this.close = close;
		this.high = high;
		this.low = low;
		this.volume = volume;
		sendStockData();
		}
	}
```

Listing 2. The StockData (ConcreteSubject) class.

```
public interface Observer
	{
	public void update(String symbol,float close,float high,float low,long volume);
	}
```

Listing 3. The Observer interface.

```
public class TradingFool implements Observer
	{
	private String symbol;
	private float close;
```

```
	public TradingFool(StockData stockData)
		{
		stockData.attach(this);
		}

	public void update(String symbol,float close,float high,float low,long volume)
		{
		this.symbol = symbol;
		this.close = close;
		display();
		}

	public void display()
		{
		DecimalFormatSymbols dfs = new DecimalFormatSymbols();
		DecimalFormat priceFormat = new DecimalFormat("###.00",dfs);
		System.out.println("Trading Fool says... ");
		System.out.println("\t" + symbol + " is currently trading at $" +
			priceFormat.format(close) + " per share.");
		System.out.println();
		}
	}
```

Listing 4. The TradingFool (ConcreteObserver) class.

```
public class BigBuyer implements Observer
	{
	private String symbol;
	private float close;
	private float high;
	private float low;
	private long volume;

	public BigBuyer(StockData stockData)
		{
		stockData.attach(this);
		}

	public void update(String symbol,float close,float high,float low,long volume)
		{
		this.symbol = symbol;
		this.close = close;
		this.high = high;
		this.low = low;
		this.volume = volume;
		display();
		}

	public void display()
		{
		DecimalFormatSymbols dfs = new DecimalFormatSymbols();
		DecimalFormat volumeFormat = new DecimalFormat("###,###,###,###",dfs);
		DecimalFormat priceFormat = new DecimalFormat("###.00",dfs);
		System.out.println("Big Buyer reports... ");
		System.out.println("\tThe lastest stock quote for " + symbol + " is:");
		System.out.println("\t$" + priceFormat.format(close) + " per share (close).");
		System.out.println("\t$" + priceFormat.format(high) + " per share (high).");
		System.out.println("\t$" + priceFormat.format(low) + " per share (low).");
		System.out.println("\t" + volumeFormat.format(volume) + " shares traded.");
		System.out.println();
		}
```

```
	}
```

Listing 5: The BigBuyer (ConcreteObserver) class.

```
public class StockQuotes
	{
	public static void main(String[] args)
		{
		System.out.println();
		System.out.println("-- Stock Quote Application --");
		System.out.println();

		StockData stockData = new StockData();

		// register observers...
		new TradingFool(stockData);
		new BigBuyer(stockData);

		// generate changes to stock data...
		stockData.setStockData("JUPM",16.10f,16.15f,15.34f,(long)481172);
		stockData.setStockData("SUNW",4.84f,4.90f,4.79f,(long)68870233);
		stockData.setStockData("MSFT",23.17f,23.37f,23.05f,(long)75091400);
		}
	}
```

Listing 6: The StockQuotes class – our client application

You may benefit from walking through some of the code in Listings 1 through 6. Listing 6 creates an instance of the **`StockData`** class. The **`StockData`** object maintains an **`observers`** list (an **`ArrayList`** that holds all registered observers). The client code creates instances of **`TradingFool`** and **`BigBuyer`**. As these instances construct themselves, they add themselves to the **`observers`** list. That is, **`tradingFool`** and **`bigBuyer`** register to receive **`stockData`** subject notifications.

```
new TradingFool(stockData);
new BigBuyer(stockData);
```

The constructor of each concrete observer contains the following lines to accomplish the registration:

```
stockData.attach(this);
```

The **`attach()`** method in the **`StockData`** class (Listing 2) adds the interested observer to the **`ArrayList`**.

At this point, a call to the **`setStockData()`** method is a trigger for the subject to notify all registered observers that something has changed. In this application, the **`setStockData()`** method passes in the stock symbol, the closing price, the high price for the day, the low price for the day, and the total volume of shares traded. Once these variables have their values, it calls **`sendStockData()`** to publish the news.

The first attempt to write a **`StockData`** class had a ten-line **`sendStockData()`** method. In comparison, the **`sendStockData()`** method in Listing 2 is a one-liner. This new version of **`sendStockData()`** simply calls the **`notifyObservers()`** method. You may be wondering why the new class keeps this silly-looking **`sendStockData()`** method. There's a good reason for this. If the **`sendStockData()`** method has already been established in other client code, then bypassing the **`sendStockData()`** method could break that client code. Thus, the one-line **`sendStockData()`** method changes the implementation while maintaining the established interface.

Figure 3 shows a run of the **`StockQuotes`** program (the code in Listing 6).

```
-- Stock Quote Application --

Trading Fool says...
      JUPM is currently trading at $16.10 per share.

Big Buyer reports...
      The lastest stock quote for JUPM is:
      $16.10 per share (close).
      $16.15 per share (high).
      $15.34 per share (low).
      481,172 shares traded.

Trading Fool says...
      SUNW is currently trading at $4.84 per share.

Big Buyer reports...
      The lastest stock quote for SUNW is:
      $4.84 per share (close).
      $4.90 per share (high).
      $4.79 per share (low).
      68,870,233 shares traded.

Trading Fool says...
      MSFT is currently trading at $23.17 per share.

Big Buyer reports...
      The lastest stock quote for MSFT is:
      $23.17 per share (close).
      $23.37 per share (high).
      $23.05 per share (low).
      75,091,400 shares traded.
```

Figure 3: The output of the stock quote application

The code in Listings 1 through 6 hinges upon a very important principle (quoted from the GoF book):

"Strive for loosely coupled designs between objects that interact."

When you follow this principle, objects interact with each other, but they don't know each other's intimate details. With the Observer design pattern, the subject knows only that its registered subscribers implement the **`Observer`** interface. The subject merrily sends updated data to any of its observers. The observers can come and go as they please and nothing but the client code needs to know about the comings and goings. Using this loosely coupled design, you can build applications that can adapt easily to change. You've minimized (if not eliminated) the dependencies among objects.

Use of Observers in the Java API

The Observer design pattern uses a **push** model. In the push model, observers don't make requests for new information. Instead, the subject ***pushes*** new, unsolicited information to all registered observers. The subject pushes by calling each observer's **`update()`** method.

The opposite of the push model is a **pull** model. In the pull model, observers make requests for new information. (Observers ***pull*** information from the subject.) Each observer pulls information by calling the subject's getter methods.

The Java API has a built-in **`java.util.Observable`** class and a **`java.util.Observer`** interface to support both push and pull models.

The **`Observable`** class has pre-defined methods to manage observers:

```
void addObserver(Observer observer);
void deleteObserver(Observer observer);
void notifyObservers();
void notifyObservers(Object args);
```

The **`Observable`** class also has methods to keep track of state changes (e.g., changes in stock quote values):

```
void setChanged();
boolean hasChanged();
void clearChanged();
```

To demonstrate the pull model, the stock quote application has been modified to use the Java API versions of **`Observable`** and **`Observer`**. Listings 7 and 8 contain the modified code.

```
public class StockData extends Observable
      {
      private String symbol;
      private float close;
      private float high;
      private float low;
      private long volume;

      public StockData()
            {}

      public String getSymbol()
            {
            return symbol;
            }

      public float getClose()
            {
            return close;
            }

      public float getHigh()
            {
            return high;
            }

      public float getLow()
            {
            return low;
            }

      public long getVolume()
            {
            return volume;
```

```
		}

	public void sendStockData()
		{
		setChanged();
		notifyObservers();
		}

	public void setStockData(String symbol,float close,float high,float low,long volume)
		{
		this.symbol = symbol;
		this.close = close;
		this.high = high;
		this.low = low;
		this.volume = volume;
		sendStockData();
		}
	}
```

Listing 7. The StockData class using Observable.

```
public class BigBuyer implements Observer
	{
	private String symbol;
	private float close;
	private float high;
	private float low;
	private long volume;

	public BigBuyer(Observable observable)
		{
		observable.addObserver(this);
		}

	public void update(Observable observable,Object args)
		{
		if(observable instanceof StockData)
			{
			StockData stockData = (StockData)observable;
			this.symbol = stockData.getSymbol();
			this.close = stockData.getClose();
			this.high = stockData.getHigh();
			this.low = stockData.getLow();
			this.volume = stockData.getVolume();
			display();
			}
		}

	public void display()
		{
		DecimalFormatSymbols dfs = new DecimalFormatSymbols();
		DecimalFormat volumeFormat = new DecimalFormat("###,###,###,###",dfs);
		DecimalFormat priceFormat = new DecimalFormat("###.00",dfs);
		System.out.println("Big Buyer reports... ");
		System.out.println("\tThe lastest stock quote for " + symbol + " is:");
		System.out.println("\t$" + priceFormat.format(close) + " per share (close).");
		System.out.println("\t$" + priceFormat.format(high) + " per share (high).");
		System.out.println("\t$" + priceFormat.format(low) + " per share (low).");
		System.out.println("\t" + volumeFormat.format(volume) + " shares traded.");
		System.out.println();
		}
```

```
}
```

Listing 8: The BigBuyer class using Observable.

In Listing 7, the **`java.util.Observable`** class takes the place of the **`Subject`** interface (from Listing 1). The **`Observable`** class has pre-defined methods to manage observers, so the new **`StockData`** class in Listing 7 doesn't need to maintain its own **`ArrayList`**.

The new **`BigBuyer`** class (Listing 8) stores any **`java.util.Observable`** instance. Before calling methods like **`getSymbol()`** and **`getVolume()`**, the **`BigBuyer`** class's **`update()`** method must do some casting.

The essential difference between the pull and push models is the flow of control. In the push model (Listings 1 to 6), the flow goes like this:

1. Someone calls the **`stockData`** object's **`setStockData()`** method.
2. Within **`stockData`**, the **`setStockData()`** method (directly or indirectly) calls the **`notifyObservers()`** method.
3. The **`notifyObservers()`** method reaches out of the **`stockData`** object to call the **`bigBuyer`** object's **`update()`** method.

With the push model, most of the action takes place in the **`StockData`** class. But in the pull model (Listings 7 and 8), the flow works differently:

1. Someone calls the **`stockData`** object's **`setStockData()`** method.
2. Within **`stockData`**, the **`setStockData()`** method (directly or indirectly) calls the **`setChanged()`** and **`notifyObservers()`** methods.
3. Behind the scenes, in the Java API, the call to **`notifyObservers()`** triggers a call to the **`bigBuyer`**'s **`update()`** method. Now the ball is in **`bigBuyer`**'s court!
4. The **`bigBuyer`** object takes the initiative to reach out and pull in the new stock quote data. That is, the **`bigBuyer`** object calls the **`stockData`**'s getter methods. Herein lies the difference between pushing and pulling. With the pull model, the **`bigBuyer`** observer pulls information from the **`stockData`** observable.

Typical Use of Observer Design Pattern

The Observer design pattern is typically used in GUI applications. In a multi-windowed application, one window (the subject) sends updates to other (dependent) windows. But, as shown in this article, you don't need a GUI application to make effective use of the Observer design pattern. The pattern works well in all kinds of applications.

Some Consequences

Use of the Observer design pattern can have both good and bad consequences. Here's a brief list:

Abstract Coupling Between Subject and Observer

A subject knows only that it must notify a list of observers. The subject doesn't know the concrete implementation of each observer, and observers can come and go as they please. When an observer comes or goes, neither the subject nor the other observers are modified. This creates a very loose coupling that allows for subjects and observers to live at different levels of abstraction. For example, a high-level observer can register itself with a low-level subject.

Support for Broadcast Communication

The subject broadcasts its notifications to all interested observers without regard for the number of observers that it needs to notify. The burden of using the updated data rests solely with the observers. This allows you to add (register) and delete (unregister) at any given time.

Unexpected Updates

A change to a subject may have an adverse effect on the registered observers. A spurious update may be difficult to track down if there is no protocol to explain exactly ***what*** changed in the subject.

A Versatile Tool

The Observer pattern is a very versatile tool for modeling real-life objects. That's what object-oriented programming is all about -- modeling real-life objects.

Resources

Design Patterns – Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
ISBN 0-201-63361-2
http://www.awprofessional.com/bookstore/product.asp?isbn=0201633612&rl=1

Head First Design Patterns
Eric & Elisabeth Freeman
ISBN 0-596-00712-4
http://www.oreilly.com/catalog/hfdesignpat/

Object-Oriented Software Construction
Bertrand Meyer
ISBN 0-13-629155-4
http://www.amazon.com/gp/product/0136291554/102-4538903-0207360?v=glance&n=283155

Data & Object Factory
http://www.dofactory.com/Patterns/Patterns.aspx

About the Authors

Barry Burd is a professor in the Department of Mathematics and Computer Science at Drew University in Madison, New Jersey. When he's not lecturing at Drew University, Dr. Burd leads training courses for professional programmers in business and industry. He has lectured at conferences in America, Europe, Australia and Asia. He is the author of several articles and books, including "Java 2 For Dummies" and "Eclipse For Dummies," both published by Wiley.

Michael P. Redlich is a Senior Research Technician (formerly a Systems Analyst) at ExxonMobil Research & Engineering, Co. in Clinton, New Jersey with extensive experience in developing custom web and scientific laboratory applications. Mike also has experience as a Technical Support Engineer for Ai-Logix, Inc. where he developed computer telephony applications. He holds a Bachelor of Science in Computer Science from Rutgers University. In his spare time, Mike facilitates the ACGNJ Java Users Group and serves as ACGNJ Secretary. Mike has co-written several articles for Java Boutique, and his computing experience includes computer security, relational database design and development, object-oriented design and analysis, C/C++, Java, Visual Basic, FORTRAN, Pascal, MATLAB, HTML, XML, ASP, VBScript, and JavaScript in both the PC and UNIX environments.