NEWS

Speaking on the Observer pattern

How can you use the Observer pattern in your Java design?

By Tony Sintes

JavaWorld | May 25, 2001 2:00 AM PT

Q: I want to use the Java Observer pattern in my project. With that in mind, can you give me some sample code to demonstrate how it works

A: Just as object-oriented programming encourages code reuse, design patterns encourage design reuse. Indeed, design patterns allow you to reuse clean, time-tested designs. However, design patterns have come under an increasing amount of criticism lately. Critics point out that inexperienced developers can easily fall into the *pattern trap.*

The pattern trap blinds inexperienced developers. As such, instead of solving a solution in the best way possible, the end goal for novice developers centers on implementation of as many design patterns as possible. To some, using a design pattern seems to guarantee a good design. By that logic, if you use a lot of design patterns, you'll have a great design! Often times this belief leads to designs that just don't make sense -- even though the design might incorporate multiple patterns. Unfortunately, a design pattern does not guarantee good design.

In order to properly use a design pattern in your design, you must uphold three criteria:

1. Understand your problem
2. Understand the pattern
3. Understand how the pattern solves your problem

You need to know criterion number 1 first and foremost. How can you apply a pattern if you do not fully know what problem you are trying to solve?

You also need to know the second criterion: You must fully understand the patterns you are interested in applying. How can you apply a pattern if you do not understand it? More importantly, how can you even consider a pattern if you do not understand what it does?

Finally, if you cannot articulate how the pattern will solve your problem (why it is appropriate), forget it. Don't fall into the pattern trap and simply use a pattern so you can say you used it.

I'm not saying that the reader is necessarily falling into this trap. However, by the wording of the question, many developers may get the wrong impression about patterns. From the question, I understand that the reader probably knows the problem and understands the Observer pattern. He or she simply needs to see the pattern implemented in Java.

Before giving a Java example, it may help other readers to first give a brief description of the Observer pattern.

Simply, the Observer pattern allows one object (the observer) to watch another (the subject). The Observer pattern allows the subject and observer to form a publish-subscribe relationship. Through the Observer pattern, observers can register to receive events from the subject. When the subject needs to inform its observers of an event, it simply sends the event to each observer.

For example, you might have a spreadsheet that has an underlying data model. Whenever the data model changes, the spreadsheet will need to update the spreadsheet screen and an embedded graph. In this example, the subject is the data model and the observers are the screen and graph. When the observers receive notification that the model has changes, they can update themselves.

The benefit: it decouples the observer from the subject. The subject doesn't need to know anything special about its observers. Instead, the subject simply allows observers to subscribe. When the subject generates an event, it simply passes it to each of its observers.

(For more information on the Observer pattern please see Resources.)

Consider the following Java example:

```
public interface Subject {
      public void addObserver( Observer o );
      public void removeObserver( Observer o );
}
```

In the code above, the `Subject` interface defines the methods that a `Subject` must implement in order for `Observers` to add and remove themselves from the `Subject`.

```
public interface Observer {
      public void update( Subject o );
}
```

The `Observer` interface (above) lists the methods that an `Observer` must implement so that a `Subject` can send an update notification to the `Observer`.

Let's consider a simple implementation of `Subject` -- an `IntegerDataBag`:

```
import java.util.ArrayList;
import java.util.Iterator;
public class IntegerDataBag implements Subject {
	private ArrayList list = new ArrayList();
	private ArrayList observers = new ArrayList();
	public void add( Integer i ) {
		list.add( i );
		notifyObservers();
	}
	public Iterator iterator() {
		return list.iterator();
	}
	public Integer remove( int index ) {
		if( index < list.size() ) {
			Integer i = (Integer) list.remove( index );
			notifyObservers();
			return i;
		}
		return null;
	}
	public void addObserver( Observer o ) {
		observers.add( o );
	}
	public void removeObserver( Observer o ) {
		observers.remove( o );
	}
	private void notifyObservers() {
		// loop through and notify each observer
		Iterator i = observers.iterator();
		while( i.hasNext() ) {
			Observer o = ( Observer ) i.next();
			o.update( this );
		}
	}
}
```

`IntegerDataBag` holds onto `Integer` instances. The `IntegerDataBag` also allows `Observers` to add and remove themselves.

Consider these two implementations of `Observer` -- `IntegerAdder` and

IntegerPrinter:

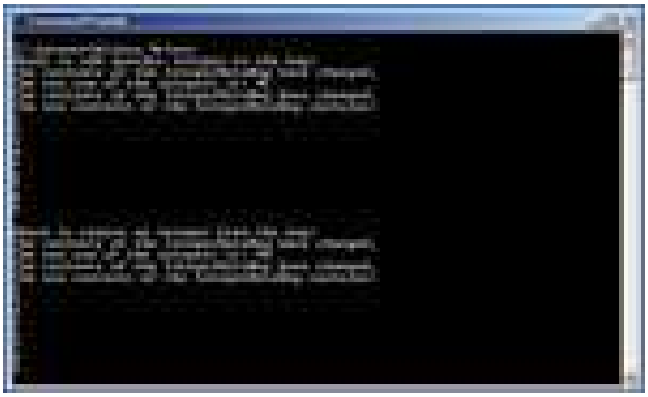
```
import java.util.Iterator;
public class IntegerAdder implements Observer {
	private IntegerDataBag bag;
	public IntegerAdder( IntegerDataBag bag ) {
		this.bag = bag;
		bag.addObserver( this );
	}
	public void update( Subject o ) {
		if( o == bag ) {
			System.out.println( "The contents of the IntegerDataBag
have changed." );
			int counter = 0;
			Iterator i = bag.iterator();
			while( i.hasNext() ) {
				Integer integer = ( Integer ) i.next();
				counter+=integer.intValue();
			}
			System.out.println( "The new sum of the integers is: "
+ counter );
		}
	}
}
import java.util.Iterator;
public class IntegerPrinter implements Observer {
	private IntegerDataBag bag;
	public IntegerPrinter( IntegerDataBag bag ) {
		this.bag = bag;
		bag.addObserver( this );
	}
	public void update( Subject o ) {
		if( o == bag ) {
			System.out.println( "The contents of the IntegerDataBag
have changed." );
			System.out.println( "The new contents of the
IntegerDataBag contains:" );
			Iterator i = bag.iterator();
			while( i.hasNext() ) {
				System.out.println( i.next() );
			}
		}
	}
}
```

`IntegerAdder` and `IntegerPrinter` add themselves to the integer bag as observers. When an `IntegerAdder` receives an update, it sums up the `Integer` values held in the bag and displays them. Likewise, when `IntegerPrinter` receives an update, it prints out the `Integers` held in the bag.

Here is a simple `main()` that exercises these classes:

```
public class Driver {
      public static void main( String [] args ) {
            Integer i1 = new Integer( 1 ); Integer i2 = new Integer( 2 );
            Integer i3 = new Integer( 3 ); Integer i4 = new Integer( 4 );
            Integer i5 = new Integer( 5 ); Integer i6 = new Integer( 6 );
            Integer i7 = new Integer( 7 ); Integer i8 = new Integer( 8 );
            Integer i9 = new Integer( 9 );
            IntegerDataBag bag = new IntegerDataBag();
            bag.add( i1 ); bag.add( i2 ); bag.add( i3 ); bag.add( i4 );
            bag.add( i5 ); bag.add( i6 ); bag.add( i7 ); bag.add( i8 );
            IntegerAdder adder = new IntegerAdder( bag );
            IntegerPrinter printer = new IntegerPrinter( bag );
            // adder and printer add themselves to the bag
            System.out.println( "About to add another integer to the
bag:" );
            bag.add( i9 );
            System.out.println("");
            System.out.println("About to remove an integer from the
bag:");
            bag.remove( 0 );
      }
}
```

Upon running the main, you will see:



The results from running the main() method

The `IntegerDataBag/IntegerAdder/IntegerPrinter` is a simple example of the Observer pattern. Within Java itself there are a number of examples of the Observer pattern: the AWT/Swing event model, as well as the `java.util.Observer` and `java.util.Observable` interfaces serve as examples.

Learn more about this topic

- "The 'Event Generator' Idiom," Bill Venners (*JavaWorld*, September 1998) provides an interesting discussion of the Observer pattern as well as an excellent resource list http://www.javaworld.com/jw-09-1998/jw-09-techniques.html (http://www.javaworld.com/jw-09-1998/jw-09-techniques.html)
- "Java Tip 38The Trick to 'Iterator Observers'," Philip Bishop (*JavaWorld*) shows a combination of the Iterator and Observer patterns http://www.javaworld.com/javatips/jw-javatip38.html (http://www.javaworld.com/javatips/jw-javatip38.html)
- "Programming Java Threads in the Real World, Part 6," Allen Holub (*JavaWorld,* March 1999) demonstrates how to implement the Observer pattern in a multithreaded environment http://www.javaworld.com/jw-03-1999/jw-03-toolbox.html (http://www.javaworld.com/jw-03-1999/jw-03-toolbox.html)
- The famous Gang of Four book*Design Patterns* Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley Publishing Co., 1995. ISBN0201633612) http://www.amazon.com/exec/obidos/ASIN/0201633612/javaworld (http://www.amazon.com/exec/obidos/ASIN/0201633612/javaworld)
- Want more? See the **Java Q&A** Index for the full Q&A catalog http://www.javaworld.com/javaworld/javaqa/javaqa-index.html (http://www.javaworld.com/javaworld/javaqa/javaqa-index.html)
- For over 100 insightful Java tips from some of the best minds in the business, visit *JavaWorld*'s **Java Tips** index http://www.javaworld.com/javatips/jw-javatips.index.html (http://www.javaworld.com/javatips/jw-javatips.index.html)
- Sign up for the *JavaWorld This Week* free weekly email newsletter (look for it under the IS/IT Management Series) http://reg.itworld.com/cgi-bin/subcontent12.cgi (http://reg.itworld.com/cgi-bin/subcontent12.cgi)