# COMP 401

*Prasun Dewan[[1]](#footnote-1)*

Multi-Component Observers

The focus in the previous chapter was on the observer and MVC design patterns rather than the application itself. The illustrative example used, a counter, is perhaps the simplest possible interesting object. Part of its simplicity arises from the fact that it has only one component – an integer value. This feature prevented us from presenting an important concept in the Observer (and hence also MVC) design pattern: how should a multi-component observer notify its observables?

# Observable Beans

As an example of an object with multiple components, consider the BMISpreadsheet:

public interface BMISpreadsheet {

public double getHeight();

public void setHeight(int newVal);

public double getWeight();

public void setWeight(int newWeight) ;

public double getBMI();

}

Let us consider what it means to make it an observable object. This means we must design an observer interface for it. As we saw in the case of a counter, we can design an application-independent or dependent observer interface. Let us consider the more challenging task of an application-independent interface.

We have already seen an “application-independent” observer interface:

public interface Observer {

public void update(Observable o, Object arg);

}

The quotes indicate that, in general, the arg argument would require an application-dependent interpretation. In our example, it requires a way to distinguish between a change to the Weight and Height properties. This example suggests a general way to encode changes to Beans - objects with properties – that specifies the name of the property that has changed and its new value.

# Fine-Grained Property Notification

This idea is supported by the java.beans package, which defines the following interface for objects observing changes to Bean objects:

public interface PropertyChangeListener

extends EventListener {

public void propertyChange(PropertyChangeEvent evt);

}

The terms observer and listener are synonymous – the Beans and other libraries use the latter. Unlike the Observer update() method, the propertyChange() method takes a single argument, of class PropertyChangeEvent:

public class PropertyChangeEvent

extends EventObject {

public PropertyChangeEvent (

Object source, String propertyName,

Object oldValue, Object newValue) {…}

public Object getNewValue() {…}

public Object getOldValue() {…}

public String getPropertyName() {…}

…

}

This predefined class providers the notified observer with getters to access the source of the notification, that is, the observable that called the method, the name of the changed property, and its old and new value. It provides the notifying observable with a constructor to initialize these values.

Why provide both the old and new value of the property? There are two reasons. First, the Java beans package also provides observable beans with a class called PropertyChangeSupport for notifying observers. This class does not send a notification if its old and new value are not the same. Second, certain observers may be interested in how much a property has changed. For example, a counter observer might wish to display the increment added and a BMISpreadsheet observer might wish to display the changes in the three properties. Write methods now become more complicated as they must keep both the old and new property values. If an observable knows the observer does not make use of the old value, it can simply send null for it.

Providing a single notification method for all properties is not safe as we can assign illegal value to the property name, old value, and new value parameters. For instance, the property name parameter may be accidentally assigned “Wght” because that used to be the name of property now called “Weight”. As the old and new value are of type Object, we can assign to them values that are inconsistent with the type of the property. Thus, this method must be used with care. These issues show the drawbacks of an application-independent interface.

# Introspection-based Bean Observer Registration and Deregistration

Recall that one problem with the Java Observer/Observable types is that each observable was required to be a subclass of the class Observable. This drawback does not occur in the Java Beans framework. As we see above, the type of the source observable Bean is Object. So, an observable Bean does not have to a subclass of some preexisting class or even implement an interface! So how does an observer know what methods it should call to register or unregister itself?

The Java Beans framework defines standard headers for the registration and deregistration methods:

addPropertyChangeListener(PropertyChangeListener l) {…}

removePropertyChangeListener(PropertyChangeListener l) {…}

An observable Bean is expected to provide the correct implementation of these methods. Thus, each Bean must implement observer registration/deregistration methods, which was not the case when we subclassed Obsever. The advantage, of course, is that the Bean can inherit from other class.

Like the headers of getters and setters, these are not specified in any interface. A potential observer of a Bean is expected to analyze or “introspect” all the headers of a Bean. If it finds these two methods, it assumes that the Bean is observable, and calls them. (Java provides a “reflection” API to list and call methods of an object.)

This introspect approach is error prone. What if we accidentally misspelled the name of the registration method to addPropertyChangeListner? The error will go uncaught as these methods in a Bean are options. This is exactly the kind of situation interfaces can guard against. Therefore, the ObjectEditor library defines in the util.models package an interface with the header:

public interface PropertyListenerRegisterer {

void addPropertyChangeListener(

PropertyChangeListener aListener);

}

Observable Beans should implement this interface. The interface does not include the deregistration method to reduce the burden of the implementer of the interface. A Bean is of course free to implement this method.

Can we use a similar solution to address mistakes in names of Bean getters and setters?

# Notification Creation and Broadcast as Separate Concerns

As the class Observable shows, observer registration, deregistration and notification is application independent. Thus, we need a way to reuse this code in different observable Beans. Using an IS-A link from our class reduces our flexibility to inherit from another class. As we saw in the previous chapter, the HAS-A link provides another approach, called delegation, to reuse code. Thus, we can define reusable types for observer registration, to which Observable Beans have references. The following is an implementation of this class:

public class APropertyListenerSupport implements PropertyListenerSupport {

public final int MAX\_SIZE = 50;

PropertyChangeListener[] contents = new PropertyChangeListener[MAX\_SIZE];

int size = 0;

public int size() {

return size;

}

public PropertyChangeListener elementAt (int index) {

return contents[index];

}

boolean isFull() {

return size == MAX\_SIZE;

}

public void addPropertyChangeListener(PropertyChangeListener l) {

if (isFull())

System.*out.println("Adding item to a full history");*

else {

contents[size] = l;

size++;

}

}

public void notifyAllListeners(PropertyChangeEvent event) {

for (int index = 0; index < size(); index++) {

elementAt(index).propertyChange(event);

}

}

}

The code here is very similar to what we saw in ACounterObserver. The class keeps a list of observers and provides observer registration and deregistration methods to change this list. It provides a notification method, which broadcasts its argument to all of the observers. The main difference between this class and ACounterObserver is that it does not contain any write methods that actually call the notifyAllListeners() method. It is based on the principle that creation and broadcast of notifications are separate concerns. A Bean model is responsible for creation of the notification and this class manages its broadcast.

Thus class is a simplified version of the PropertyChangeSupport class provided by the java.bean package. It has been presented here to show how such support can be implemented. In your project, you are free to copy this code create your own broadcast support, or use the Java implementation of it. Though the former approach requires a bit more work, it will allow you to better debug your program if you think a notification is not reaching an observer.

# Implementation of Observable Bean

Let us illustrate the concepts above by showing how they can be used to create an observable BMISpreadsheet:

public class AnObservableBMISpreadsheet

implements BMISpreadsheet, PropertyListenerRegisterer {

PropertyListenerSupport propertyListenerSupport =

new APropertyListenerSupport();

public void addPropertyChangeListener(

PropertyChangeListener listener) {

…;

}

…

public void setWeight(double newWeight) {

…

}

}

The class implements the PropertyListenerRegistererinterface to ensure the header of the observer registration method is correct. It creates an instance ofAPropertyListenerSupportto manage broadcast of notifications. The addPropertyChangeListener() method in PropertyListenerRegisterer forwards the call to the instance of APropertyListenerSupport and informs the listener about its current property values:

public void addPropertyChangeListener(

PropertyChangeListener listener) {

propertyListenerSupport.addPropertyChangeListener (listener);

listener.propertyChange(**new** PropertyChangeEvent(**this**, "Weight", 0, getWeight()));

listener.propertyChange(**new** PropertyChangeEvent(**this**, "Height", 0, getHeight()));

listener.propertyChange(new PropertyChangeEvent(this, "BMI", 0, getBMI()));

}

The setter for each property now does not simply change the property. Before changing a property, it stores the old value of the property. After making a property change, it uses the old and new value of the property to create a notification event, which it sends to the support for broadcast.

public void setWeight(double newWeight) {

double oldWeight = weight;

double oldBMI = getBMI();

weight = newWeight;

propertyListenerSupport.notifyAllListeners(

new PropertyChangeEvent(this, "weight", oldWeight, newWeight));

propertyListenerSupport.notifyAllListeners(

new PropertyChangeEvent(this, "bmi", oldBMI, getBMI()));

}

# ObjectEditor Support of Property Notifications

Observable Beans allow us to finally overcome the auto refresh problem in ObjectEditor. Recall that our solution so far involves the use of the refresh() method:

public class ABMISpreadsheetRefreshedByMain {

public static void main (String[] args) {

BMISpreadsheet bmiSpreadsheet = new ABMISpreadsheet();

OEFrame oeFrame = ObjectEditor.*edit*(bmiSpreadsheet);

bmiSpreadsheet.setHeight(1.77);

bmiSpreadsheet.setWeight(75);

oeFrame.refresh();

}

}

This “solution” has two problems. First, it involves a brute force refresh of the complete logical structure of the displayed object. Second, each piece of code that changes any component of the logical structure must be aware of the OEFrame instance returned by the edit() call, and call refresh. With application-independent Bean notifications, each Bean component of the logical structure notifies its observers about fine-grained changes to it. ObjectEditor registers appropriate listeners for *each* such component, to solve the two problems above.

Recall that the refresh() operation needed to be called only a property was changed by the program. If it was changed by ObjectEditor, ObjectEditor automatically called refresh() after the change, which was inefficient. With observable Beans, there is no need to do implicit refresh() after a property change, leading to a more efficient edit. The flip side is that if the observable does not send the appropriate notifications, the display will not be correctly updated. We discuss this more under partially observable objects.

# Summary of Observable Beans

As this chapter and the previous one together show, the idea of an observable bean requires much explanation. However, once we understand the idea, it is perhaps as easy to apply as a formula – which holds true for most design patterns. Here is a summary of the steps you need to take to create an observable bean:

1. Use the code of APropertyListenerSupport above to create your own version of the class, through which you can then step in a debugger or through print statements. You are free to give the class (and its interface) any name you wish, as it is not directly visible to the Bean observers.
2. Implement the PropertyListenerRegistrerer interface provided by the ObjectEditor library. Even though you have the code for it, do not define your own interface with the same name!
3. Implement the addPropertyListener() method by forwarding the call to the notifrication-support class and announcing initial property values to the observer. The announcement will not be important to ObjectEditor, which calls all getters when asked to edit an object. However, it will be important to other observables that do not call read methods.
4. In each write method, use the notification-support class to broadcast each property changed by the method, keeping track of the old value of the property to do so.

The steps above are required to implement an observable. If ObjectEditor is the only observer of your observable, then you do not need to do more. If you are writing your own observers, then you must:

1. Implement the predefined PropertyListener interface in each observer.
2. Register each observer with each observable it observes.

# Efficient Composite Observable Beans

Our example Bean object had only atomic properties. What happens when the Bean is a composite, that is, has one or more structured properties. To illustrate the new issues that arise with such objects, consider our example of a ACartesianPlane, which has four properties, XAxis, YAxis, XLabel, and YLabel. Consider the setAxisLength() method:

**public** **void** setAxesLength(**int** anAxesLength) {

axesLength = anAxesLength;

xAxis.setWidth(axesLength);

yAxis.setHeight(axesLength);

xAxis.setX(toXAxisX());

xAxis.setY(toXAxisY());

yAxis.setX(toYAxisX());

yAxis.setY(toYAxisY());

xLabel.setX(toXLabelX());

xLabel.setY(toXLabelY());

yLabel.setX(toYLabelX());

yLabel.setY(toYLabelY());

}

The method changes the AxesLength property and location of the two labels and the location and size of the two lines forming the axes. What notifications should be sent to allow the views of this object, such as ObjectEditor, to redraw? It needs to send notification only for the property it changes, AxesLength. It need not send notifications for the properties of the components these properties do not change. For example, the XAxis property does not change, through its properties do. This, we can create a subclass of our original ACartesianPlane in which the setAxesLength() is overridden as follows:

**public** **void** setAxesLength(**int** newVal) {

**int** oldVal = getAxesLength();

**super**.setAxesLength(newVal);

propertySupport.notifyAllListeners(**new** PropertyChangeEvent(**this**, "axesLength", oldVal, newVal));

}

The classes of the changed properties are, of course, responsible for sending changes. For instance, the setX() method of ALine and AStringShape A must notify changes to the X coordinate.

public void setX(int newVal) {

int oldVal = getX();

super.setX(newVal);

propertySupport.notifyAllListeners(

new PropertyChangeEvent(this, "X", oldVal, newVal));

}

If the two classes share the implementation of this method, then a single method must be changed to send X property notifications.

Thus, the call to setAxesLength() can result in multiple notifications, corresponding to the properties it changes, only one of which is announced directly by it. These notifications are sent by atomic shapes in the logical structures of the composite CartesianPlane.

Thus, a composite object needs to send notifications only for the properties it directly changes, not its descendants.

If, after resizing the axes length, we assigned a new shape to the XLabel property, then ACartesianPlane would have to send a notification about this change.

# The Inefficiency Coarse of Grained Updates

Here is an alternative implementation of an observable version of ACartesianPlane in which we do not make ALine and AStringShape observables. Instead, we send property notifications in the overridden setAxesLength() method:

public void setAxesLength(int newVal) {

int oldVal = getAxesLength();

super.setX(newVal);

propertySupport.notifyAllListeners(new PropertyChangeEvent(this, "axesLength", oldVal,

newVal));

propertySupport. notifyAllListeners(new PropertyChangeEvent(this, "XAxis", xAxis, xAxis));

propertySupport. notifyAllListeners(new PropertyChangeEvent(this, "YAxis", yAxis, yAxis));

propertySupport.notifyAllListeners(new PropertyChangeEvent( this, "XLabel", xLabel, xLabel));

propertySupport.notifyAllListeners(new PropertyChangeEvent(

this, "YLabel", yLabel, yLabel));

}

}

These properties have not really changed in that new objects have not been assigned to the properties. This is the reason that the old and new values of these objects are the same. As our implementation of notification support does not compare the old and new values, these notifications will reach ObjectEditor. ObjectEditor also will not compare these values, and assume that the observable wishes each of these properties to be redrawn. (A proper comparison, in any case, requires the correct implementation of an equals() method discussed later, which an object cannot be guaranteed to provide.) It will delete the display of the old values of these properties and then add the displays of the new values. Thus, this approach is inefficient as the entire object referenced by a property is redrawn instead of only the parts of the object that changed. Therefore, it is more efficient to announce the smallest change that occurred – not a coarser grained change.

# Announcing Changes on Behalf of Other Objects

The smallest change can occur in an object that does not have a setter but has a getter whose value depends on the variables in a parent object. Assume a variation of the ACartesianPlane, we discussed earlier, in which the two labels and lines did not have any setters, and their getters returned values dependent on the variables such as axesLength in ACartesianPlane. In this case, the write methods in the parent object would announce changes to component objects, as shown in the modified setAxesLength() below:

**public** **void** setAxesLength(**int** newVal) {

**int** oldVal = getAxesLength();

**super**.setAxesLength(newVal);

**int** oldXAxisX = 0;

**int** newXAxisX = 0;

propertySupport.notifyAllListeners(**new** PropertyChangeEvent(**this**, "axesLength", oldVal,

newVal));

propertySupport. notifyAllListeners(**new** PropertyChangeEvent(xAxis, "X", oldXAxisX, newXAxisX));

// ...

}

The PropertyChangeEvent now does not take **this** as its first argument. Instead, it takes a component object as an argument, and announces a change to a property of the component.

We saw earlier that the getters of such a composite object are inefficient. We see here another problem, where the object must announce changes for all of its dependent descendants.

Nonetheless, this solution shows that it is possible to announce changes on behalf of other objects, which is useful when these objects do not have setters or their classes cannot be changed.

# Fully vs. Partially Observable

It is possible for an object to be partially observable if only some of the changes to its properties and their components are announced. Thus, there are two reasons for partial observability:

1. A class announces changes to only some of its logical components. For example, ALine announces changes to its location but not its size.
2. Only some of the objects assigned to its logical components announce changes. For example, the labels of ACartesianPlane announce changes to their properties but the axes of the class do not.

In general, to ensure that the display of an object is completely up to date in a view, each change to its logical structure must be announced, which can be tedious.

# Debugging Observable-Observer Interactions Involving ObjectEditor

Partial observability might be one reason why update to some component of an observer does not result in expected observer reaction. There might actually be four distinct reasons for this situation:

1. The class of the observable does not define a registration method.
2. The observer does not call this method in the observer.
3. The observer is not sent changes to the component.
4. The observer does not correctly react to the change.

Often you will define a single observable class for both ObjectEditor and your own observers. An important difference between the two cases is that ObjectEditor automatically calls the registration method while you must write and call your own code to register your observers with the observable. If ObjectEditror is not calling a registration method of a class, check that the class has implemented the predefined PropertyListenerRegisterer interface. If your own observer is not registered, there may be several reasons. The obvious one is that the registration code is not written or not called. The more subtle one is that it is being called on the wrong observable. In your project, you will create several observables but will write your own observers for only one of them, letting ObjectEditor display the rest. A common mistake is to register with a component or parent of the observable in which you are interested.

A common complaint that occurs when you use ObjectEditor is that “some value is refreshed correctly in the main window but not in the graphics window.”

An object/property is either displayed in the graphics window or in the main window - not both. For instance, a point's x and y coordinates are not displayed as text fields in the main window.

A more accurate description of the problem above is that you have two copies of the same value, one displayed in the main window and one in the graphics window, and you have announced changes to the former and not the latter.

# ObjectEditor Support of Collection Notifications

We have seen how to use standard Bean libraries to notify changes to Bean components. Just as there are no standard conventions for extracting logical components of non-Bean structured types such as collections, there are also no standard libraries for notifying changes to non-Bean structured types. Again, just as ObjectEditor defines its own conventions for extracting the logical structure of indexed lists and tables, it defines code for notifying changes to components of these types.

For indexed collections, in utils.models, ObjectEditor defines the VectorListener, VectorChangeEvent, and VectorListenerRegisterer interfaces, which corresponds to the PropertyChangeListener, PropertyChangeEvent, and PropertyListenerRegister interfaces:

public interface VectorListener {

public void updateVector(VectorChangeEvent evt);

}

-------------------------------------------------------------------

public interface VectorListenerRegisterer {

public void addVectorListener(

VectorListener aListener);

}

-------------------------------------------------------------------

package util.models;

public class VectorChangeEvent {

Object source;

Object otherSource;

// constants for event types

public static final int *AddComponentEvent = 2*

*DeleteComponentEvent = 2,*

*ChangeComponentEvent = 3,*

*InsertComponentEvent = 4,*

*CompletedComponentsEvent = 5,*

*ClearEvent = 6,*

*UndefEvent = 1000;*

*// constructor, oldObject can be null when no value is replaced*

public VectorChangeEvent(Object theSource, int type, int posn,

Object oldObject, Object newObject, int newSize) {..}

The word, Vector, in these interfaces reflects the fact that they were originally invented for Vectors. In fact, they can be used for any variable sized collection. VectorChangeEvent recognizes supports four important different changes to such a collection: adding a new element to the end of the collection, deleting an existing element, inserting a new element at an arbitrary position, and clearing of the collection. It defined integer constants for each of these event types. When an instance of this class is created, the constructor must, like the PropertyChangeEvent, specify the source object of the event, the type of the event (using one of the constants mentioned above), the position of the element added/inserted/deleted by the write operation, the old object (if any) at that position, the new object (if any) put by the operation at that position, and the new size of the collection after the operation finishes execution.

If an object defines the following method:

**public void addVectorListener(VectorListener vectorListener) {…}**

the ObjectEditor view invokes it to register itself with the object. Each time a write method is invoked in the collection, it should notify each registered listener about the change. This is illustrated in the adaptation of the class AStringHistory we saw earlier.

public class AnObservableStringHistory

implements StringHistory, VectorListenerRegisterer{

VectorListenerSupport vectorListenerSupport =

new AVectorListenerSupport();

…

public void addElement(String element) {

if (isFull())

System.*out.println("Adding item to a full history");*

else {

contents[size] = element;

size++;

vectorListenerSupport.notifyAllListeners(

new VectorChangeEvent(this, VectorChangeEvent.*AddComponentEvent,*

size -1, null, element, size));

}

}

@ObserverRegisterer(ObserverTypes.*VECTOR\_LISTENER)*

public void addVectorListener(VectorListener aListener) {

vectorListenerSupport.addElement(aListener);

}

…

For tables, a simpler listener interface can be used, which does not require a special event to be created:

public interface HashtableListener {

public void keyPut(Object source, Object key,

Object value, int newSize);

public void keyRemoved(Object source, Object key, int newSize);

}

**public** **interface** HashtableListenerRegisterer {

**public** **void** addHashtableListener(HashtableListener aListener);

}

Again, the term Hashtable is a consequence of the fact that when this support was added, Hashtable was the only table provided by Java. These interfaces can be used for any table that follows the table signature patterns defined by ObjectEditor based on the methods of Hashtable.

# Summary

The MVC framework defines three kinds of objects: models, views, and controllers, which are connected to each other dynamically. When a user enters a command to manipulate a model, a controller processes it and invokes an appropriate method in the model, which typically changes the state of the model. If the method does changes the model state, it notifies the views and other observers of the model about the change. A view responds to the notification by redisplaying the model.

# Exercises

1. What advantages does the interactor pattern approach offer over the monolithic main approach?
2. What advantages does the MVC framework offer over the interactor approach? Do you see any disadvantages?
3. Distinguish between models and listenables, and views and listeners.
4. In the MVC pattern, describe the roles of the controller(s), model, and view(s). Classify all of the classes you created for Assignment 9 as a controller, a model, a view, or none of the above.
5. Compare and contrast the three different notifications schemes you learned about. The three schemes are embodied in Java’s java.util.Observer implemention, AWT components, and java.beans package (PropertyChangeListener and PropertyChangeEvent).

1. © Copyright Prasun Dewan, 2015. [↑](#footnote-ref-1)