COMP 110/401

Prasun Dewan[1]

15. Model-View-Controller (MVC) and Observer

The principle we have used so far to implement interactive applications has been to keep computation and user-interface code in separate classes. All user-interface code, however, was put in one class. We will see here the rationale for the principle and also how we can refine it to support both reusability and multiple, concurrent, consistent user interfaces manipulating the same object. The technique we will use can be abstracted into the general Model-View-Controller (MVC) pattern. This pattern is built on top of the Observer pattern, which has applications beyond user interfaces.

Programming Interface vs. User Interface

MVC has to do with implementing user-interfaces. User-interfaces are different from the interfaces we have studied so far in class in that they are the interfaces through which the user rather than the programmer manipulates objects. Interfaces are often termed as programming interfaces to distinguish them from user-interfaces. The figure illustrates the difference between the two kinds of interfaces for the BMI example:

```
public class ABMISpreadsheet
implements BMISpreadsheet {
   double height;
   public double getHeight() {
      return height;
   }
   public void setHeight(double
            newHeight) {
      height = newHeight;
   }
   double weight;
   public double getWeight() {
      return weight;
   }
   public void setWeight(double
            newWeight) {
      weight = newWeight;
   }
   public double getBMI() {
      return weight/(height*height);
   }
}
```

Service

BMISpreadsheet.java

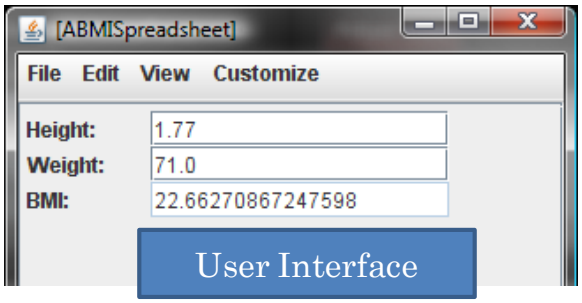
```
public interface BMISpreadsheet {

    public double getWeight();
    public void setWeight(double newVal);

    public double getHeight();
    public void setHeight(double newVal);

    public double getBMI();
}
```

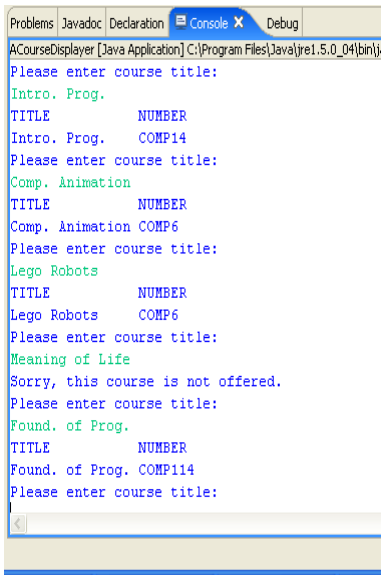
Programming Interface



User Interface

Automated by ObjectEditor

Almost objects we have created so far have had user-interfaces for invoking methods of the objects. (One exception was the scanner objects, which were internal objects that were never manipulated directly by the user.) However, we have not worried about most of these user-interfaces, since ObjectEditor took care of implementing most of them. We did implement some user-interface manually – in particular the console user-interfaces such as the one shown below:




```
public class ACourseDisplayer    {
   public static void main(String[] args)  {
         fillCourses();
         while (true) {
            System.out.println("Please enter course title:");
            String inputLine = System.console().readLine();
            if (inputLine.equals("."))
                  break;
            Course matchedCourse = courses.matchTitle(inputLine);
            if (matchedCourse == null)
                  System.out.println("Sorry, this course is not offered.");
            else {
                  printHeader();
                  print (matchedCourse);
            }
      }
}
```

Our implementation of these interfaces was rather messy in that the complete user-interface was implemented in the main-class, as shown in the figure above. It was messy in that it made changing of the user-interface and sharing of user-interface code difficult.

This would not be a problem if implementation of user-interfaces was trivial. In fact, it is a very difficult problem. To reduce this problem, interactive systems provide libraries to create Console-based user interfaces, and user-interface toolkits to create graphical user-interfaces. However, surveys have shown that, on average, about fifty percent of programs that user the libraries/toolkits is devoted to handling the user-interface tasks for creating a display, mapping input commands to application functions, and displaying results of these commands. This problem is aggravated by the fact that user-interfaces of applications keep changing as illustrated by comparing the user-interfaces of Windows XP and Vista, and Office 2003 and 2007.

Therefore, it is important to modularize the user-interface code so that it can be easily changed and shared.

Multiple User-Interfaces

To illustrate this more concretely and in detail let us take a simpler example – that of a counter. The counter provides a (programming) interface that allows addition of an arbitrary positive/negative value to an interface. Figure 1 shows three different user interfaces to manipulate the counter. Each user interface allows input of a value to add to the counter and displays the updated value to the user. Figure 1 (left) uses the console window for input and output. First, it displays the initial value of the counter, and then it allows the user to enter the (positive or negative) values to be added to the counter, and after the input value, it shows the current value of the counter. The user interface of Figure 1 (middle) retains the mechanism to change the counter but uses an alternative way to display the counter value. Instead of printing it in the console window, it creates a new “message” window on the screen that shows the value. Figure 1 (right) shows that it is possible to combine elements of the other two user interfaces. It also retains console-based input but displays the counter in both the console window and the message window.

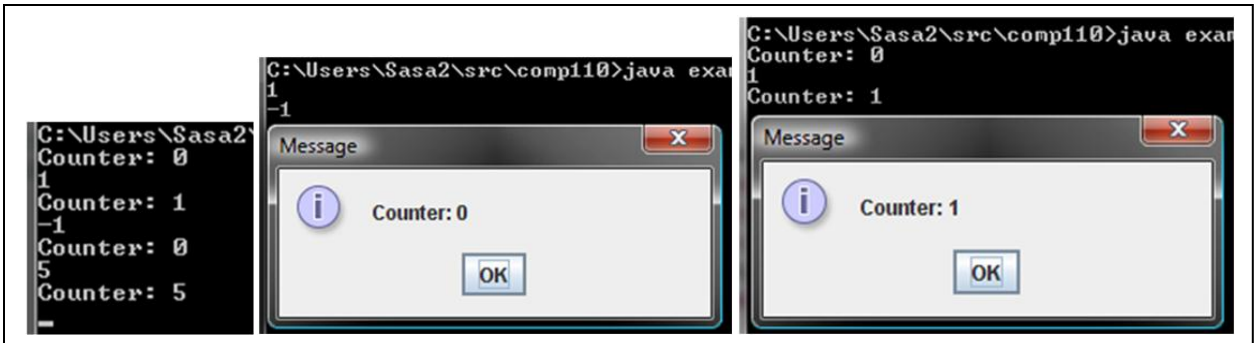


Figure 1. Three different user interfaces for manipulating a counter

If we were told to implement only the interface of Figure 1 (left), we would probably write the following code:

```
public class ConsoleUI {
	static int counter = 0;
	public static void main(String[] args) {
		System.out.println("Counter: " + counter);
		while (true) {
			int nextInput = Console.readInt();
			if (nextInput == 0) break;
			counter += nextInput;
			System.out.println("Counter: " + counter);
		}
	}
}
```

If only Figure 1 (middle) was to be implemented, we would similarly write:

```
import javax.swing.JOptionPane;
public class ConsoleUI {
	static int counter = 0;
	public static void main(String[] args) {
		while (true) {
			int nextInput = Console.readInt();
			if (nextInput == 0) break;
			counter += nextInput;
			JOptionPane.showMessageDialog(null,
				"Counter: " + counter.getValue());
		}
	}
}
```

Instead of appending to the console, it uses the `JOptionPane` class of the Swing toolkit to display the value in a new message window.

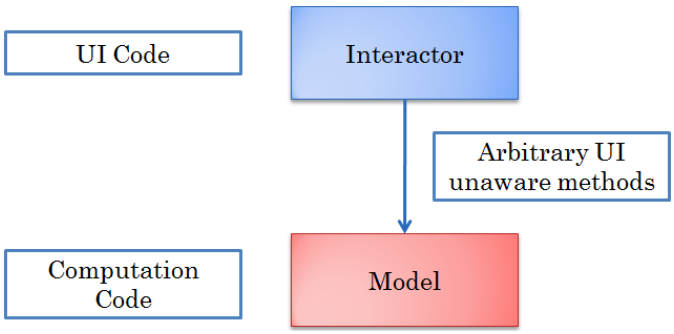


Figure 2. Interactor Pattern

Interactor Pattern

However, this is far from the ideal solution if we want to reuse code among the three interfaces. For one, it does not allow the code manipulating the counter to be shared. We can create a special object that allows a counter to be incremented or decremented by an arbitrary value and provides a method to access the current counter value. The following class describes this object:

```
public class ACounter implements Counter {
      int counter = 0;
      public void add (int amount) {
            counter += amount;
      }
      public int getValue() {
            return counter;
      }
}
```

Thus, we are now following the principle of keeping user interface and computation code separate. This principle is captured by the model/interactor pattern shown in Figure 2.

Here we refer to the class implementing the user interface code as the interactor and the object it manipulates as the model. This pattern allows us to create new user interfaces for an object without changing existing model and user interface code. Figure 3 shows how the pattern can be applied to create the three user interfaces from Figure 1. Separate classes, `AConsoleUI`, `AMixedUI`, and `AMultipleUI`, implement the user interface of Figure 1 left, middle, and right, respectively. Each class is independent of the other main classes and dependent only on the interface of the model.

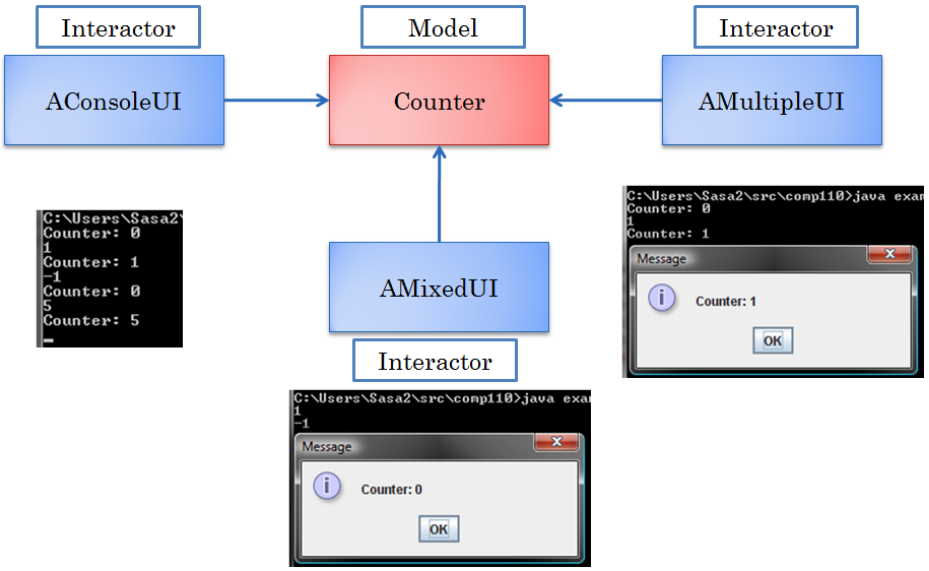


Figure 3. Three interactor classes using the same model class

The code for `AConsoleUI` is given below:

```
public class AConsoleUI implements ConsoleUI {
      public void edit (Counter counter) {
            System.out.println("Counter: " + counter);
            while (true) {
                  int nextInput = Console.readInt();
                  if (nextInput == 0) break;
                  counter.add(nextInput);
                  System.out.println("Counter: " + counter.getValue());
            }
      }
}
```

The implementation of the console I/O accepts the model as a parameter, displays its initial value to the user, and then executes a loop that calls the add method of the model with each input value and then displays the new counter.

The main class now simply composes the interactor with the model:

```
public static main (String args[])
      (new AConsoleUI()).edit(new ACounter());
}
```

Similarly, the interactor for `AMixedUI` is:

```
public class AMixedUI implements ConsoleUI {
	public void edit (Counter counter) {
		while (true) {
			int nextInput = readInt();
			if (nextInput == 0) break;
			counter.add(nextInput);
			JOptionPane.showMessageDialog(null,
				"Counter: " + counter.getValue());
		}
	}
}
```

Again, instead of appending to the console, it uses the `JOptionPane` class of the Swing toolkit to display the value in a new message window.

The interactor for the combined user interface, `AMultipleUI`, combines the output of the other two interactors:

```
public class AMultipleUI implements ConsoleUI {
	public void edit (Counter counter) {
		System.out.println("Counter: " + counter.getValue());
		while (true) {
			int nextInput = Console.readInt();
			if (nextInput == 0) break;
			counter.add(nextInput);
			System.out.println("Counter: " + counter.getValue());
			JOptionPane.showMessageDialog(null,
				"Counter: " + counter.getValue());
		}
	}
}
```

Need to Share UI Code

The model-interactor pattern allows sharing of model code in the various user-interfaces. However, it does not provide a way for sharing code among the various user interfaces. As mentioned before, all user interfaces provide the same way to input the counter increment. It should be possible to share its implementation among the three classes. Moreover, the last user interface combines the mechanisms of displaying output in the other two approaches. There should be a way to share the implementation of these mechanisms.

The need to share code is illustrated in more depth by looking at the implementation of the three interactor classes above. The input code is duplicated in all three classes and the output code between the first and second and between the second and third classes. Figure 4 shows this graphically.

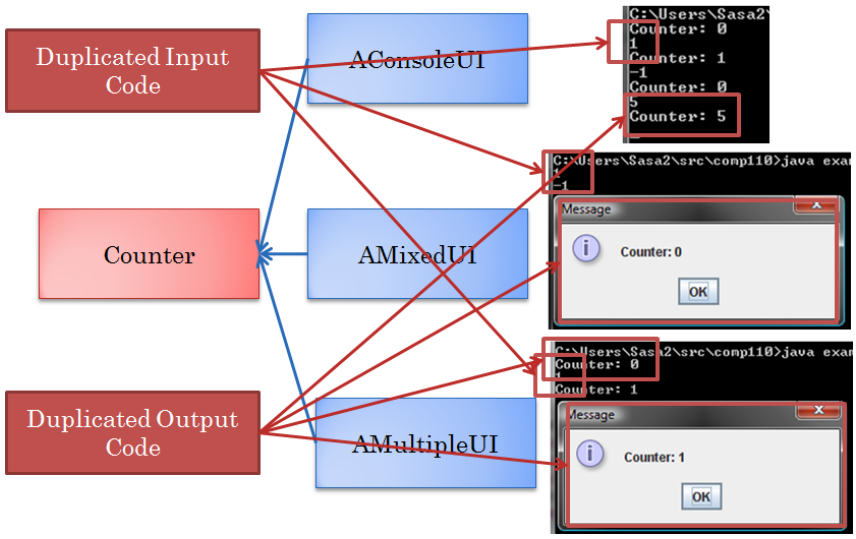


Figure 4. Input/Output mechanisms duplicated with the interactor/model pattern

Views and Controllers

The above discussion suggests that the input and output component of each user interface should be managed by separate classes, called *controllers* and *views*, respectively (Figure 5 and Figure 6).

When the user inputs a new command the controller calls some write method in the model such as `add` in our example. Conversely, to display a result, the view calls a read method such as `getValue` in the counter example.

The figure above does not indicate if the model, views, and controllers are different classes or instances. In general, a program may simultaneously create multiple models with the same behavior but different state. For example, a program may create multiple counters. This implies that the model must be an instance. Different models may simultaneously have independent controllers and views with the same behavior. More interesting, a particular model may simultaneously have multiple independent controllers and views with the same behavior. Consider a game program that allows multiple users on different computers to independently interact with an object using the same kind of user interface. Or a distributed presentation that can be viewed simultaneously by distributed users. This implies that, in general, controllers and views must also be instances.

Observer Pattern

One last question we need to answer to complete the description of this user-interface organization has to do with giving feedback to a user command. When a controller processes an input command, how do all the views know they should update their displays? For instance, if the console controller is used to increment a counter, how do all the views know that they should display the new counter value?

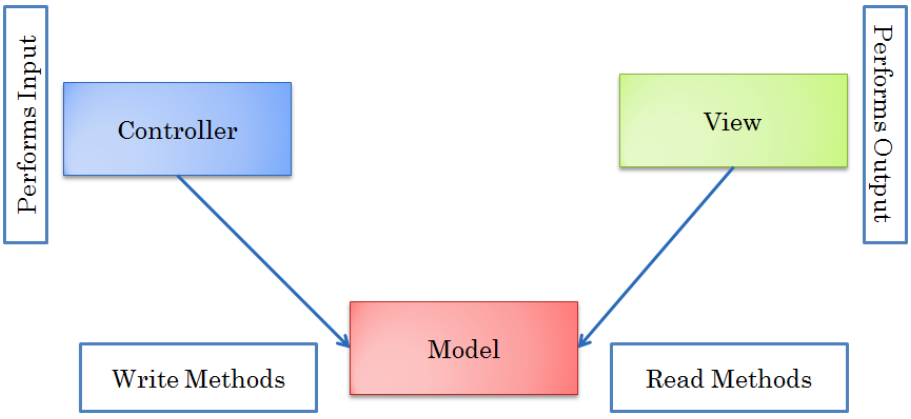


Figure 5. Splitting an interactor into a view and a controller

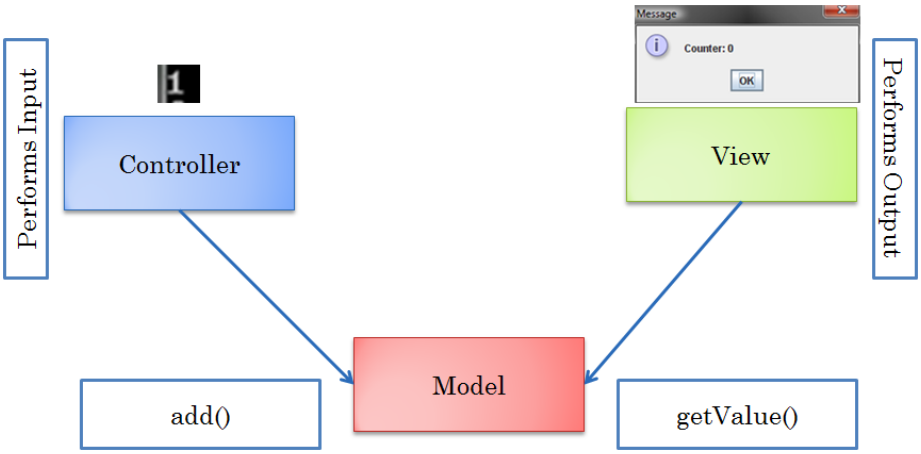


Figure 6. Example of splitting an interactor into a view and a controller

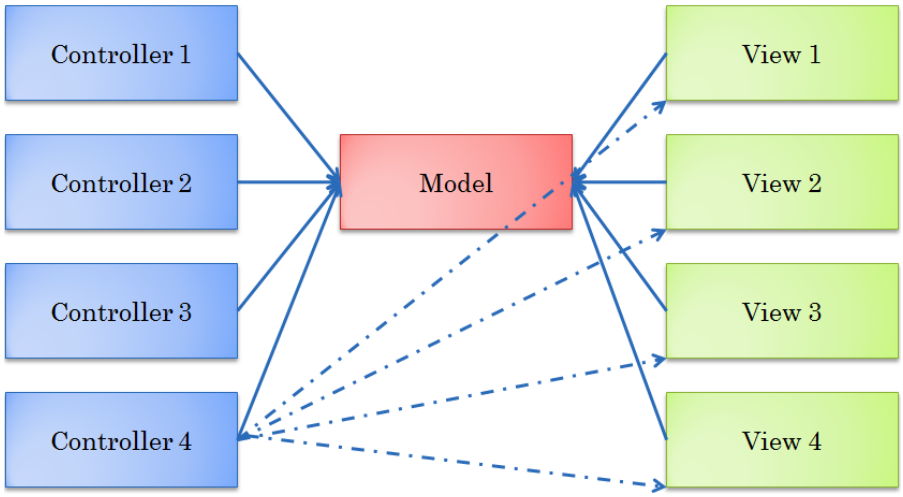


Figure 7. Controller notifying all of the observers after invoking a write method on the model

We can, of course, make the controller notify all the other views after it has invoked a write method in a model (Figure 7). This means that each controller must be aware of every view, which means the pieces of code that create views and controllers must coordinate with each other, making them complex. Moreover, this approach does not allow views to be updated autonomously, that is, without user input. Imagine a counter being updated every second or a temperature sensor object being updated when a new thermometer reading is sensed.

Fortunately, a simpler, centralized approach can be taken that addresses both problems. Whenever the model history is changed, autonomously or by a controller, it informs all of its views that it has changed. As a view is dynamically created/destroyed, it registers/unregisters itself with the model so that the model knows who it should notify.

Thus, now the model is aware of its views, as indicated by the dashed lines. One of the advantages of the previous architecture was that model was completely independent of its user-interface components – both views and controllers. As a result, these components could change without requiring any changes to the model. Now that the model is aware of its views, would we ever have to change it if we decide to create some new kind of view for it?

Fortunately, the methods invoked by an object on its views can be independent of the behavior of the view. All the object has to do is notify its views that it has changed. This information is independent of the kind of display created by the view. In fact, it can be sent not only to views of an object but also other objects interested in monitoring changes to it. For instance, a counter could notify an object keeping a log of all changes to its counter. A temperature sensor object could notify an object that alarms users if the temperature goes beyond some threshold value. The Java toolkit allows an object representing a button or text field to also send notifications to objects that wish to react to button presses and text field edits, respectively. The following figure shows an example of a non-view observer of our counter object. A “rocket launcher” observer “launches” a rocket by printing a message when the counter value goes to zero (Figure 8). The fact that its message is printed after the console views prints the zero value of the counter indicates that it was notified after the console view.

The notifying object is called an observable and the notified object is called its observer. The dashed lines from the models to its views indicate that the awareness in it of its observers is *notification awareness*: it knows which observers needs to be notified about changes to it but does not know any aspect of their behavior. [2] Even though the observable/observer notion appeared first with the context of the larger MVC pattern, it is now recognized as an independent pattern. Figure 9 shows the pattern independent of MVC.

[2] This is in contrast to the awareness in the views and controllers of the point history, which know about its getter and setter methods. If the model changes, the views and controllers typically change in response because of this awareness.

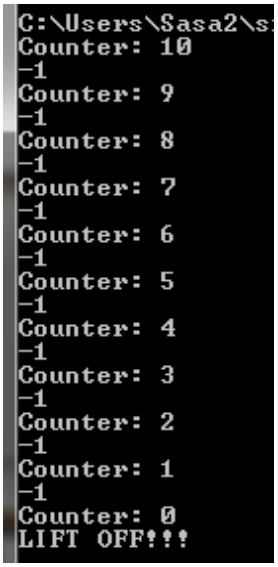


Figure 8. Non-view Observer

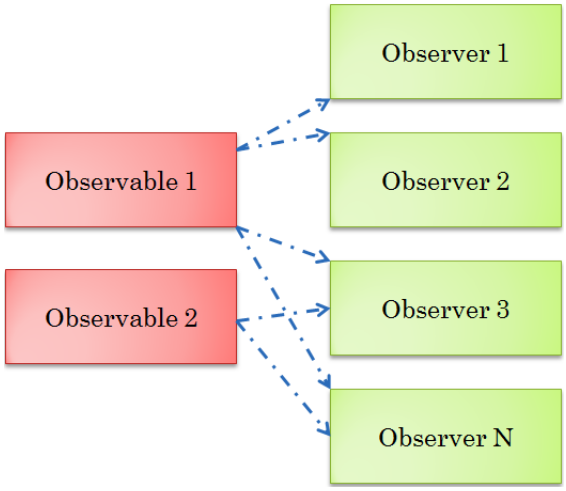


Figure 9. Observer Pattern

Not just objects, but also humans regularly use this idea of notification. For instance, students in a class are often notified when the web page for the class is changed. Consider the various steps that take place to make such interaction possible:

- The professor for a course provides a way to add and remove students from a mailing list.
- Students may directly send the professors add and remove requests or some adminstrator may do so on their behalf as they add/drop the course.
- When a professor modifies the information linked from the web page, he/she sends mail to the students in the mailing list. On the other hand, when he simply reads the information, he does not send any notification.
- The notification tells the student which professor sent it so that they know which web page to look at.

- The students access the page to read the modified information.

The observable/observer pattern has analogous steps:

- An observable provides methods to add and remove observers from an observer list.
- Observers may directly invoke these methods or some other object, such as another model, may do so on their behalf.
- When an observable modifies its state, that is, invokes a write method, it calls a method in each observer to notify it about the change. On the other hand, when it executes a read method, it does not call this notification method.
- The call to the notification method identifies the observable doing the notification. The reason is that an object may observe more than one observable. For example, a battle simulation user interface may observe the positions of multiple tanks and planes.
- Each notified observer calls read methods in the observable to access the modified state.

The observer pattern allows models and views to be bound to each other. A separate mechanism is needed to bind a controller to one or more models. A controller can provide a setter method for doing so, and it or some other object (such as façade, which we will learn about later) can call this method. A view with a single model can also provide such a method so that it does not have to rely on the notification method providing the model reference.

MVC Implementation

We are ready to see how the observer-based MVC pattern can be implemented in our sample application. Let us first define the interfaces of an observable counter:

Observable/Observer Interfaces

```
public interface ObservableCounter {
	public void add(int amount);
	public int getValue();
	public void addObserver(CounterObserver observer);
	public void removeObserver(CounterObserver observer);
}
```

The interface defines the same methods as the Counter interface, for reading and writing the counter, and methods to add and remove counter observers, which are defined by the following interface:

```
public interface CounterObserver {
	public void update(ObservableCounter counter);
}
```

The update method is the notification method called each time an observable changes. Its argument is the observable that changed.

Observable Implementation

The implementation of the observable interface supports two operations defined by it. The write method, `add`, notifies the observers, as shown below:

```
public class AnObservableCounter implements ObservableCounter {
      int counter = 0;
      ObserverList observers = new AnObserverList();
      public void addObserver(CounterObserver observer) {
            observers.addElement(observer);
            observer.update(this);
      }
      public void removeObserver(CounterObserver observer) {
            observers.removeElement(observer);
      }
      public void add(int amount) {
            counter += amount;
            notifyObservers();
      }
      public int getValue() {
            return counter;
      }
      void notifyObservers() {
         for (int observerNum = 0; observerNum < observers.size();
                  observerNum++)
            observers.elementAt(observerNum).update(this);
      }
}
```

The list of registered observers is kept in the observer history, `observers`. The methods `addObserver` and `removeObserver` simply call existing methods to add and remove elements from the history.

The method `notifyObservers` retrieves each element of `observers`, invoking the `update` method defined by the observer interface.

As we have seen earlier, the keyword `this` in a method refers to the object on which the method is called. Thus, the program fragment

```
update(this)
```

passes `update` a reference to the observable so that it can invoke getter methods on it to display the counter.

Observer Implementations

The console and message views implement the generic interface for observing a counter. The console view implements the update method by simply appending a display of the counter to the console.

```
public class ACounterConsoleView implements CounterObserver {
	public void update(ObservableCounter counter) {
		appendToConsole(counter.getValue());
	}
	void appendToConsole(int counterValue) {
		System.out.println("Counter: " + counterValue);
	}
}
```

The message view provides a different implementation of the method, creating a message window displaying the counter value:

```
public class ACounterJOptionView implements CounterObserver {
	public void update(ObservableCounter counter) {
		displayMessage(counter.getValue());
	}
	void displayMessage(int counterValue) {
		JOptionPane.showMessageDialog(
			null, "Counter: " + counterValue);
	}
}
```

Since it used the Swing class, `JOptionPane`, to create the message view, it is called `ACounterJOptionView`.

The implementation shown below, takes an action only when the notifying counter goes to zero:

```
public class ARocketLaunchingCounterObserver implements CounterObserver
{
	public void update(ObservableCounter counter) {
		if (counter.getValue() == 0)
			launch();
	}
	public void launch() {
		System.out.println("LIFT OFF!!!");
	}
}
```

Console Controller

Finally, consider the controller object. Its interface is given below:

```
public interface CounterController {
	public void setModel(Counter theCounter);
	public void processInput();
}
```

The first method can be called by an external object, such as the main class (later will also see an inteactor façade), to bind it to a model. The second method starts its input processing loop. Because of the loop, this method should be called after all other actions have been taken to create the model view controller pattern.

The class of the console controller is given below:

```
public class ACounterController implements CounterController {
	ObservableCounter counter;
	public void setModel(ObservableCounter theCounter) {
		counter = theCounter;
	}
	public void processInput() {
		while (true) {
			int nextInput = Console.readInt();
			if (nextInput == 0) break;
			counter.add(nextInput);
		}
	}
}
```

After each input, it calls the add method of the counter with the value. It does not have to worry about the output – that is the concern of the views.

Control Flow

To understand how the three kinds of objects, model, view, and controller, interact with each other, assume that the model is bound to a console view, a message view, and the rocket launcher. Let us see what happens when a user enters a new value:

- The controller gets the new value `int` and invokes `add` on the model.
- The model calls `notifyObservers`, which in turn, invokes `updated` on all the observers in the order in which they were registered.

The console view shows the new value in the console window, the message view creates a message window, and the rocket launcher prints its message if the counter has reached 0.

Variations in Observer/Observable Communication

The notion of observers is pervasive in object-oriented software. Different implementations of this idea differ in the syntax they use (e.g. the term Listener instead of Observer) and the amount of information an observer is passed about the changed object.

Consider the implementation we saw above:

```
public interface CounterObserver {
      public void update(ObservableCounter counter);
}
```

Here the notification method, `update`, takes a single parameter indicating the source of the notification. If an observer has only one observable during its life time, there is no need to pass the observable to the notification method. Instead, the observer can be informed about the observable in a constructor or in a separate method:

```
public interface CounterObserver {
      public void setObservable(ObservableCounter counter);
      public void update();
}
```

Imagine a distributed observable/observer scenario, where the two objects are located, say, in the U.S. and China, respectively. The above scheme is inefficient. The observable sends a message to China saying that it has changed. The observer must now send a message back to the US to get the changed value in which it is interested. If the observable knew the kind of change in which an observer was interested, it could have sent the change with the notification. Thus the message back to the U.S. would not have been necessary. The trick is to know in what the observer is interested.

In our example, the model can send the new value of the counter.

```
public interface CounterObserver {
      public void update(ObservableCounter counter, int newCounterVal);
}
```

Here the update method has an extra parameter representing the change.

We see this basic idea in the java.util.Observer types predefined by Java:

```
public interface java.util.Observer {
      public void update(Observable o, Object arg);
}
```

These types define a “standard” observable communication protocol for arbitrary objects. The notification method defined by the interface takes as an argument, `o`, indicating the observable that changed and another, `arg`, describing the change to the object.

In the above approach, the second argument to the notification method indicates the new value of some attribute of the observable in which the observer is interested. It could also indicate by how much the value of the attribute has changed:

```
public interface CounterObserver {
      public void update(
            ObservableCounter counter, int counterIncrement);
}
```

This approach has the advantage that an observer interested in how much the counter has changed does not have to keep the old value to determine this information. On the other hand, an observer interested in the new value, must keep the old value to determine the new value. We can have the best of both worlds by passing the old and new value of the changed attribute in the notification method:

```
public interface CounterObserver {
      public void update (
            ObservableCounter counter,
            int oldCounterValue, int newCounterValue);
}
```

We now have three parameters to the update method. We can combine them into one object:

```
public interface CounterObserver {
      public void update(CounterChangeEvent event);
}

public interface CounterChangeEvent {
      ObservableCounter getCounter();
      int getOldCounterValue();
      int getNewCounterValue();
}
```

This approach has the advantage that if multiple methods handle the notification event, a single object rather than multiple objects must be passed to these methods. Moreover, the notification information can be returned by a function, which can compute a single value. Also, the notification information can be very elaborate, containing not only a description of the change but also, for instance, the time when the change occurred and a unique ID for it. If separate parameters were used for all units of event information, then the notification method would have to declare each of these parameters, even those in which it was not interested. With a single object approach, the observer needs to know about and call getter methods for only the aspect of event information in which it is interested.

This approach is also implemented by predefined Java classes. Several AWT classes such as `java.awt.TextField`, `java.awt.Button`, and `java.awt.MenuItem` provide methods to add and remove objects implementing the `ActionListener` interface:

```
public interface java.awt.ActionListener {
      public void actionPerformed(ActionEvent e);
}
```

Actions such as pressing Enter in a text field, pushing a button, and selecting a menu item result in the notification method in the `ActionListener` interface to be called.

To motivate another implementation of the observer idea, consider the `BMISpreadsheet` interface:

```
public interface BMISpreadsheet {
      public double getHeight();
      public void setHeight(int newVal);
      public double getWeight();
      public void setWeight(int newWeight) ;
      public double getBMI();
}
```

An observer interested of such an object should be told, in addition to the information we have seen above such as the observable and old and new value, the property that changed. We can define a separate update method for each property:

```
public interface BMIObserver {
     public void updateHeight(
            BMISpreadsheet bmi, int oldHeight, int newHeight);
     public void updateWeight(
            BMISpreadsheet bmi, int oldWeight, int newWeight);
     public void updateBMI(
            BMISpreadsheet bmi, double oldBMI, double newBMI);
}
```

An alternative is to define a single update method that takes an additional parameter for the property name:

```
public interface BMIObserver {
     public void update(
            BMISpreadsheet bmi, String propertyName,
            Object oldValue, Object newValue);
}
```

As the properties are of different types, the old and new value parameters are also of type `Object`.

However, this single update method approach is not safe. The reason is that 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”. Thus, this approach must be used with care.

The advantage of the single method approach is that it can generalize to arbitrary objects. It is suitable for objects such as ObjectEditor that wish to become observers of arbitrary objects. For such observers, Java defines the following types:

```
public interface java.beans.PropertyChangeListener
            extends java.util.EventListener {
      public void propertyChange(PropertyChangeEvent evt);
}

public class java.beans.PropertyChangeEvent
            extends java.util.EventObject {
     public PropertyChangeEvent (
            Object source, String propertyName,
            Object oldValue, Object newValue) {…}
     public Object getNewValue() {…}
     public Object getOldValue() {…}
     public String getPropertyName() {…}
     …
}
```

An observable for such observers must define the following method to add an observer

```
addPropertyChangeListener(PropertyChangeListener l) {…}
```

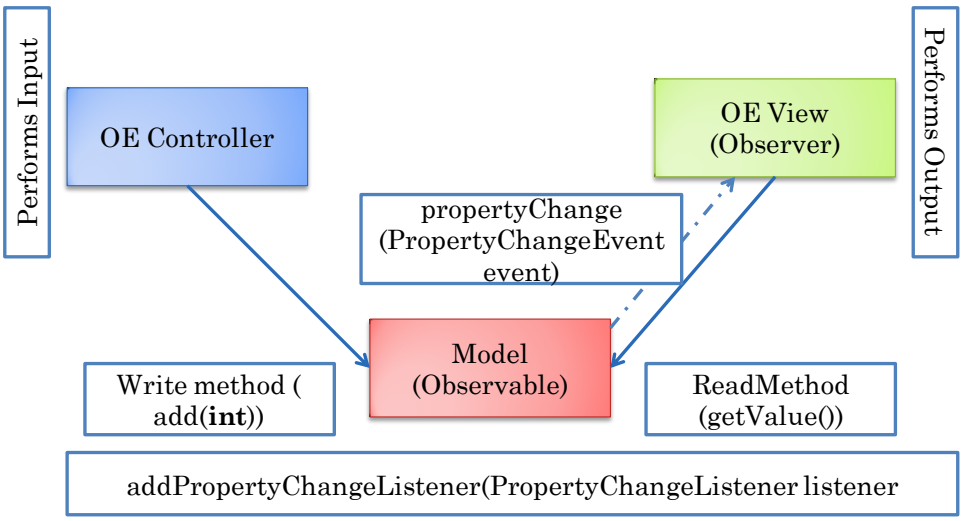
In addition, it can optionally define the following method to remove an observer:

```
removePropertyChangeListener(PropertyChangeListener l) {…}
```

You must use the predefined PropertyChangeListener as an argument to the addPropertyChangeListener and removePropertyChangeListener, and not for an interface you have defined that has the same name and arguments.

ObjectEditor Support of Property Notifications

The protocol above is used in ObjectEditor, as shown in the figure below:



ObjectEditor (OE) has both a view and a controller.

If an object defines the method:

```
addPropertyChangeListener(PropertyChangeListener l) {…}
```

the ObjectEditor view automatically calls it to register itself with the object. The method should store each PropertyChangeListener in list of listeners, as shown below:

```
public void addPropertyChangeListener(PropertyChangeListener listener) {
    listeners.addElement(listener);
}
```

The setter method for each property of the object should now call in its registered listeners the method:

```
public void propertyChange(PropertyChangeEvent evt);
```

This is illustrated in code below:

```
public void setHeight (int newVal) {
    int oldVal = height;
    height = newVal;
    notifyAllListeners(new PropertyChangeEvent(this, "height", oldVal, newVal));
}

public void setWeight (int newVal) {
    int oldVal = height;
    weight  = newVal;
    notifyAllListeners(new PropertyChangeEvent(this, "weight", oldVal, newVal));
}
```

```
public void notifyAllListeners(PropertyChangeEvent event) {
        for (int index = 0; index < observers.size(); index++) {
          listeners.elementAt(index).propertyChange(event);
}
```

Here, all setters share a single notifyAllListeners() method, which notifies each stored listener about the PropertyChangeEvent fired by a setter.

The ObjectEditor view reacts to the notification about a property change by updating the display of property, as shown below:

```
public class ObjectEditorView implements
        java.beans.PropertyChangeListener  {
    public void propertyChange (PropertyChangeEvent arg) {
        // update display of property arg.getPropertyName()
        // to show arg.getNewValue()
        …
    }
}
```

This is more efficient than the brute-force refresh approach of calling all getter methods when a setter is invoked. If your setter methods follow the above protocol, you can disable the brute-force refresh approach by invoking the command View→Auto Refresh command. We will see that this approach will be crucial to supporting animations.

Changing Object Properties

So far, we have seen changes only to primitive properties such as height and weight. To illustrate what happens with object properties, consider our example of a ACartesianPlane, which had four properties, XAxis, YAxis, XLabel, and YLabel. Consider the setAxisLength() method, reproduced below:

```
Line xAxis, yAxis;
Label xLabel, yLabel;
…
 public void setAxesLength(int newVal) {
      int lengthIncrease = newVal - axesLength;

      xAxis.setX(xAxis.getX() - lengthIncrease/2);
      yAxis.setY(yAxis.getY() - lengthIncrease/2);

      updateXOfLabel(xLabel, xLabel.getLocation().getX() + lengthIncrease/2);
      updateYOfLabel(yLabel, yLabel.getLocation().getY() - lengthIncrease/2);

      axesLength = newVal;
      xAxis.setWidth(axesLength);
      yAxis.setHeight(axesLength);
   }
   public void updateXOfLabel(Label label, int newX) {
      label.setLocation(new ACartesianPoint(
         newX, label.getLocation().getY());
   }

   …
}
```

As we see here, the method changes the 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 tically? As this method does not change the values of the four properties defined by ACartesianPlane, this class need not become an observable. As themethod changes the (logical) components of these properties, the classes of the values assigned to these properties (ALabel and ALine) must now become observables. For instance, the setLocation() method of ALabel must now notify changes to its location.

```
public void setLocation(Point newVal) {
   Point oldVal = location;
   location = newVal;
   notifyAllListeners(new PropertyChangeEvent(
   this, "location", oldVal, newVal));
}
```

Similarly, the class ALine must send notifications about changes to its size and location.

Typically composite objects such as ACartesianPlane do not have to become observables to ensure that their views are always upto date as long as the atomic objects displayed in the views are observables. A composite object needs to send notifications only if its properties are assigned new objects. For example, if, after resizing, we assigned a new label to the XLabel property, then ACartesianPlane would have to send a notification about this change.

It is possible but inefficient for a composite object to send coarser-grained updates than required to update its displays. For example, the setAxesLength() method in ACartesianPlane, could be rewritten to announce changes to all of its properties, as shown below.

```
Line xAxis, yAxis;
Label xLabel, yLabel;
…
 public void setAxesLength(int newVal) {
      int lengthIncrease = newVal - axesLength;

      xAxis.setX(xAxis.getX() - lengthIncrease/2);
      yAxis.setY(yAxis.getY() - lengthIncrease/2);

      updateXOfLabel(xLabel, xLabel.getLocation().getX() + lengthIncrease/2);
      updateYOfLabel(yLabel, yLabel.getLocation().getY() - lengthIncrease/2);

      axesLength = newVal;
      xAxis.setWidth(axesLength);
      yAxis.setHeight(axesLength);

       notifyAllListeners(new PropertyChangeEvent(
  this, "XAxis", xAxis, xAxis));
       notifyAllListeners(new PropertyChangeEvent(
  this, "YAxis", yAxis, yAxis));
       notifyAllListeners(new PropertyChangeEvent(
  this, "XLabel", xLabel, xLabel));
       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. However, some views such as ObjectEditor assume that the observable wishes each of these objects to be redrawn, and thus, refresh the display of a property even if its old and new value are the same. As a result, the atomic objects referenced by these properties need not become observables. However, this approach is inefficient as the entire object referenced by a property is redrawn instead of only the parts of the object that changed. For instance, to redraw the axis label, ObjectEditor would call all of its getters, even though only the location getter returns a new value. Therefore, it is more efficient to announce the smallest change that occurred.

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 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.

```
Line xAxis, yAxis;
Label xLabel, yLabel;
…
 public void setAxesLength(int newVal) {
      int lengthIncrease = newVal - axesLength;

      xAxis.setX(xAxis.getX() - lengthIncrease/2);
      yAxis.setY(yAxis.getY() - lengthIncrease/2);

      updateXOfLabel(xLabel, xLabel.getLocation().getX() + lengthIncrease/2);
      updateYOfLabel(yLabel, yLabel.getLocation().getY() - lengthIncrease/2);

      axesLength = newVal;
      xAxis.setWidth(axesLength);
      yAxis.setHeight(axesLength);

       notifyAllListeners(new PropertyChangeEvent(
  xLabel, "Location", oldXLabelVal, newXLabelVal));
       notifyAllListeners(new PropertyChangeEvent(
  xAxis, "Width", oldXAxisWidth, newXAxisWidth));
       ….

   }
}
```

Here, setAxesLength is announcing changes to properties of xLabel, xAxis, and other components.

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, ALabel 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 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. That is why ObjectEditor refreshes the display of an object after each method invocation if it received no notifications.

Debugging Observable-Observer Interactions

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 your registration method, check that the name of the registration method and its type are correct. If your own observer is not registered, there may be several reasons. The obvious one is that the resgitration code is not written or not called. The more subtle one is that it is being called on the wrong observable. Often, you will create several observables so ObjectEditor can automatically refresh them, but will write your own observers for only one of them. It is common to register with a component or parent of the observable in which you are interested.

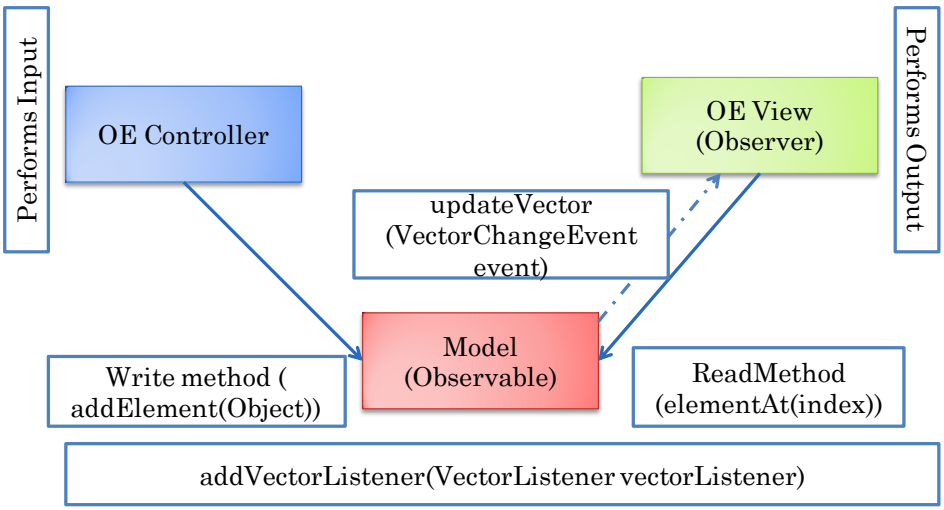
A common complaint that occurs when you use ObjectEditor is that "some value is refreshed correctly in the main window but not in the graphics window."

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

Property notifications, however, do not work for all kinds of objects. In particular, they do not work for changes to a variable sized collection, such as a AStringHistory. When we invoke a write method on such a collection such as add(), we do not change any property. There is no standard interface for communicating information about collection changes, so ObjectEditor defines one, which is summarized in the figure below.



ObjectEditor defines the following VectorListener and VectorChangeEvent interfaces, which corresponds to the PropertyChangeListener and PropertyChangeEvent interfaces:

```
package bus.uigen;
public interface VectorListener {
	public void updateVector(VectorChangeEvent evt);
}
```

```
package bus.uigen;
public class VectorChangeEvent {
	// constants to be used for event type
	public static final int AddComponentEvent = 1,
				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 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.

```
package collections;
import bus.uigen.VectorChangeEvent;
import bus.uigen.VectorListener;
public class AnObservableStringHistory extends AStringHistory {
VectorListenerHistory listeners = new AVectorListenerList();
    public void addVectorListener(VectorListener vectorListener) {
        listeners.addElement(vectorListener);
    }
    public void addElement(String element) {
        super.addElement(element);
        notifyListeners(new VectorChangeEvent(this,
    VectorChangeEvent.AddComponentEvent, size -1, null, element,
    size));
    }
    void notifyListeners (VectorChangeEvent event) {
        for (int i = 0; i < listeners.size(); i++) {
        listeners.elementAt(i).updateVector(event);
        }
    }
}
```

Multiple Observer –Observable Protocols

As we have seen above, there is no standard observer-observable protocol, and sometimes these protocols make different protocols. This means that an observable may need to register and notify multiple kinds of observers; and conversely, an observer may need to interact with multiple kinds of

observables. For example, the counter model may support the general `PropertyChangeListener` observers to allow ObjectEditor to become its observer. In addition, it might support the custom `CounterObserver` observers to support type-safe notification.

Wrapper Classes

Let us consider again the following two pieces of code we saw earlier:

```
public void setHeight (int newVal) {
    int oldVal = height;
    height = newVal;
    notifyAllListeners(new PropertyChangeEvent(this, "height",
    oldVal, newVal));
}
```

```
public class java.beans.PropertyChangeEvent extends
java.util.EventObject {
    public PropertyChangeEvent (Object source, String propertyName,
                                Object oldValue, Object newValue) {…}
    public Object getNewValue() {…}
    public Object getOldValue() {…}
    public String getPropertyName() {…}
    ….
}
```

As the highlighted pieces of code show, the int actual parameters, oldVal and newVal, are being assigned in setHeight() to the Object formal parameters, oldValue and newValue. This goes against what we have learned to fat about primitive values and objects being fundamentally different data values, with different storage schemes, operations, and assignment rules. For example we know that:

"Joe Doe".toString()

is legal, but

5.toString()

is not, as a String is an object while an int is not. The reason, recall, is that the class, Object defines the behavior of all Java objects because the type of each object is directly or indirectly a subtype of Object. It does not, however, define the behavior of primitive values, which are not objects. In fact, it is not possible to define in Java a type, Primitive, that describes all primitive values, or a type, Value, that describes all Java (object and non-object) values, because subtyping is not available for primitive types.

Treating primitive values and objects as fundamentally different has two related disadvantages. First, primitive values cannot be assigned passed as arguments to methods expecting objects. For instance, we cannot define a collection type to which we can add both primitive and Object values, because, as

mentioned above, there is no type describing both kinds of values. Second, primitive types are second-class types in that the benefits of inheritance are not applicable to them. For instance, we cannot create a new primitive type, say, Natural,that describes natural numbers (positive integers) and inherits the implementation of arithmetic operations from int.

Primitive types are also present in other object-oriented programming languages such as C++, but some more pure object-oriented languages such as Smalltalk only have object types. Primitive types can be more efficiently implemented than object types, which was probably the reason for including them in Java. However, this benefit comes at the expense of ease of programming and elegance.

To overcome some these problems, for each primitive type, Java defines a corresponding class, called a *wrapper* for that type, and provides mechanisms for automatically converting among instances of a primitive type and the corresponding wrapper class. A wrapper class:

- provides a constructor to create a wrapper object from the corresponding primitive value,
- stores the primitive value in an instance variable,
- provides a getter method to read the value.

For example, Java provides the wrapper class, Integer, for the primitive type, int. The constructor:

public Integer(**int** value);

can be used to wrap a new object around the int value passed to the constructor, and the getter instance method:

public int intValue()

can be used to retrieve the value.

The wrapper classes for the other primitive types, double, char, boolean, float, long, and short are Double, Character, Boolean, Float, Long and Short. The constructors and getter methods for wrapping and unwrapping values of these types are:

public Double(double value);
public double doubleValue();

public Character(char value);
public char charValue();

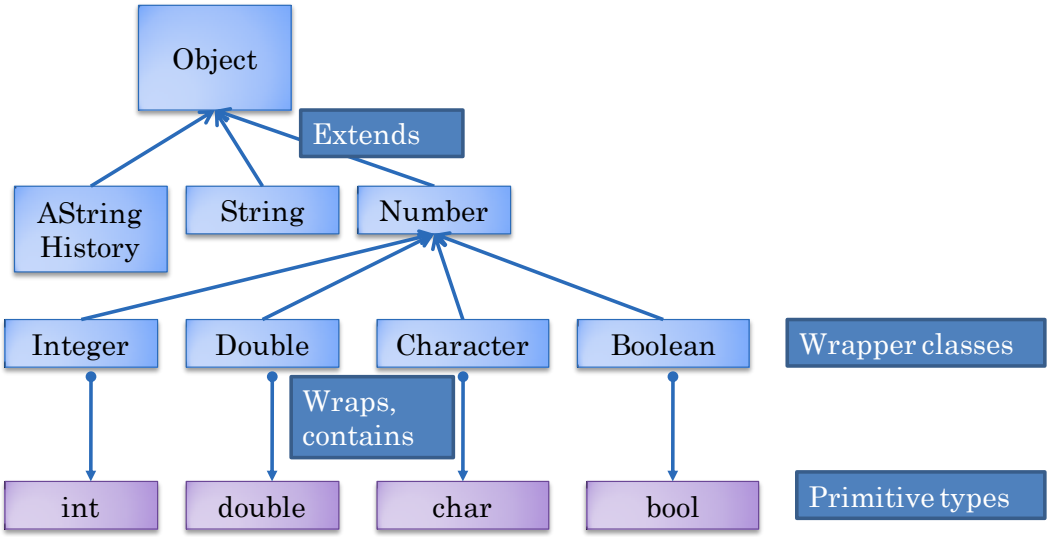
public Boolean(boolean value);
public boolean booleanValue();

public Float(float value);
public float floatValue();

public Long(long value);
public long longValue();

public Short(short value);
public short shortValue();

The following figure shows how wrapper types bridge the gap between objects and primitives.



The following code shows how a wrapper class can be used to convert between primitive and object types:

int i = 5;
Integer I = **new** Integer(5); // manually wrapping an int value into a wrapper object
int j = I.intValue() + 6; // manually unwrapping an int value from a wrapper object

Wrapping and unwrapping primitive values is fairly straightforward but tedious. Therefore, the latest Java version has automated this process by allowing primitive values to be directly assigned to variables typed using the corresponding wrapper class, and vice versa. Thus, the following code is legal:

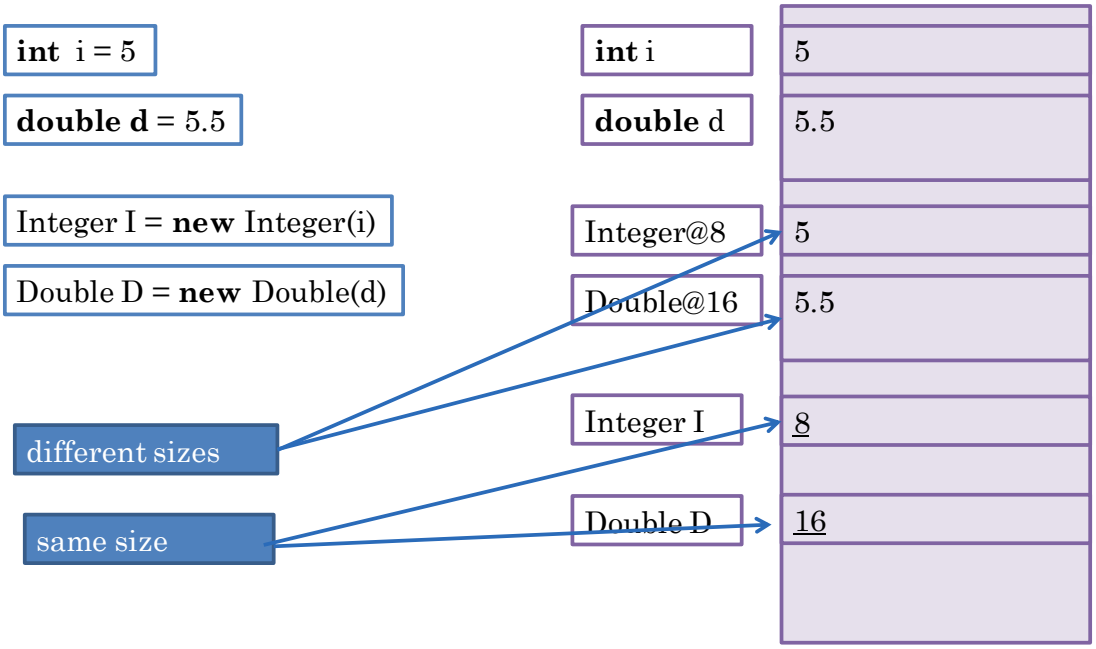
int i = 5;
Integer I = i; // automatically wrapping an int value into a wrapper object
int j = I + 6; // automatically unwrapping an int value from a wrapper object

As shown above, Java automatically wraps (unwraps) a primitive (wrapper) value when it is assigned to a wrapper (primitive) value.

It is important to note that differences in primitives and objects exist, it is just that some of them are now hidden. For instance, it is still illegal to say:

i.toString();

It is important to first assign an integer it to an object variable before we can invoke object methods on it. Similarly, the two kinds of entities are still stored differently, as the figure below shows.



A variable typed using a wrapper type contains a pointer to the wrapper object, while a variable typed using a primitive type directly stores the value. As a result, all wrapper variables are allocated the same sized storage while different primitive variables can be allocated different sizes, as the figure shows.

Besides wrapping and unwrapping primitive values, a wrapper class may also provide useful class methods for manipulating these values. For example, we have seen that we have used a class method to convert a string to the corresponding int value:

```
int i = Integer.parseInt("4");
```

and another Integer class method to convert an int value to the corresponding string:

```
String s = Integer.toString(4);
```

We must look at the documentation of each wrapper class to find what methods it provides.

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).