COMP 401

Prasun Dewan[1]

MVC and User-Interface Toolkits

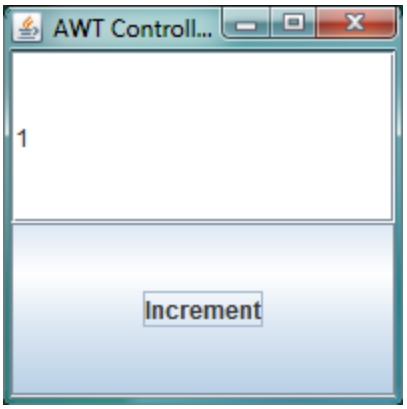
In the chapter on MVC introduction, we saw perhaps the simplest possible views and controllers. They were simple for two reasons. First, the model was a single-component counter. Second, and more important, the implemented user-interface was not a GUI (Graphical User Interface). The input in all three user-interfaces was console-based – no buttons or menus were pressed. The output in two of the user-interfaces did not involve the console – but the JOptionPane was used much like a println() - each new output was displayed in a new area on the screen.

A GUI is another term for a WIMP (Windows, Icon, Menu, Pointer) user-interface, in which these four elements are involved in providing input and viewing output. It is "wimpy" compared to user-interface involving the display of graphics – geometric shapes. How to create such interfaces is the subject of the next chapter. In fact, as we will see, WIMP user-interfaces are built on objects provided for graphics objects. We have seen that ObjectEditor is able to automatically create such interfaces. In this and the next chapter, we will see how we can create some of the user-interfaces created by ObjectEditor. As we will implement them in the context of MVC, we will be able to reuse the model objects we have created so far, and attach them transparently to out handcrafted user-interfaces – thereby seeing a concrete benefit of MVS in particular and separation of concerns in general.

In an earlier chapter, we saw how we can display a GUI by creating a logical structure involving windows and widgets. This chapter will focus on how we can do input and output with this structure, that is, how we can convert user actions into calls to model write methods, and how changes in model state results in updates to the window structure. In this process, you will learn how menus, buttons, sliders, and progress bars are implemented using a toolkit, though tutorials on the Web will provide you much better information on this subject.

Counter GUI

The distinction between GUI and non GUI interface can be concretely illustrated by transforming the non GUIs for the counter model into the following GUI:



It has two widgets – windows with some predefined behavior – a text field on the top and a button at the bottom. The text field shows the current value of the counter. The button adds 1 to the counter. Each time the button is pressed, the text-field updates. Thus, new output replaces previous output rather than being displayed in a new console line or a JOption message.

Structure and Layout as Separate Concerns

In a previous chapter on trees, DAG and graphical structures, we saw how to display widgets on the screen. Recall that the top window is a JFrame or Frame object that roots a tree of widgets, and it has leaves for performing input and/or output. In this example JFrame is the top object in this tree, and its leaves are instances of JTextField and JButton.

Before we focus on widget I/O, let consider a related issue. How do we make sure that the two widgets are displayed in a single column rather than some different layout such as a single row? One answer might be to provide different kinds of panels. For instance, we can provide a (a) horizontal panel, which arranges its children in a row, (b) vertical panel, which arranged its children into a column, and (c) a flow panel, which puts its children at specified positions in the panel. We can control the layout by putting the two leaves in one of these panels, and then adding the panel to the top-level window.

This approach meet the separation of concerns principles. A panel's job is to collect a bunch of components into one widget – to define an internal node in the window structure. It is not the only kind of object that provides such containment – for Java provides JPanel, Panel, Canvas, JFrame, Frame and several other Container instances that perform this task differ in other ways. Layout of components is a separate task. When we combine these tasks, for each containment widget, we need to provide a version that displays children horizontally, vertically, or at specified positions. Thus, we have to provide the cross product of containment and layout alternatives. If we separate these two tasks into different classes, we need to provide the union of the two kinds of alternatives.

Early toolkits bundled containment and layout together. One of the many nice innovations of Java is to separate them into different kinds of classes. Any container class can be attached to any layout task to create the necessary containment and layout. This separation is illustrated by the following code to create the display of Figure ???.

```
JFrame frame = new JFrame("AWT Controller and View");
JButton button = new JButton("Increment");
JTextField textField = new JTextField();
frame.setLayout(new GridLayout(2,1));
frame.add(textField);
frame.add(button);
frame.setSize(200, 200);
frame.setVisible(true);
```

Here, we have attached a JFrame, a special kind of container, to a GridLayout, a special kind of layout. The constructor of the layout indicates the number of row and columns in the grid. Here, specify that two rows and one column.

Separation of concerns is, of course, what MVC is designed to follow. Here we see another design pattern related to user-interfaces in which this principle is followed, which says that widget containment and layout should be in different objects.

Let us now consider the main task at hand: how to do widget I/O using MVC.

Model Write Methods → View Notifications → Widget Write Methods

Recall that a view reacts to changes in the model by calling application-independent output provided by an I/O library. In the previous chapter, the I/O library was the console output and JOption class. Now the library is the full set of toolkit widgets. Our view now calls that part of the toolkit functionality that does output. We do not know this functionality, and a full knowledge of it is beyond the scope of this course.

For the JTextField widget used for output in our example, it is easy to derive part of this functionality. The widget displays updatable text. So it makes sense for it to provide a write method that takes a String argument and replaces the current text with the argument. Not surprisingly, the method is called setText(). Thus, our widget-based counter view calls this method rather than println().

One slight complication is that println() was called on a well- known application-independent object stored in System.out. setText() can be called in one of many instances of JTextField. This, the view must know the instance that displays the counter value. In our example, the task of composing the logical structure and layout of the user-interface will be done by a class that is separate from the view and controller – the main class. Thus, the main class must pass the JTextField instance to the view.

The following is an implementation of our widget-based counter view:

```
public class ACounterJTextFieldView implements CounterObserver {
  JTextField textField;
  public ACounterJTextFieldView(JTextField theTextField){
    textField = theTextField;
  }
  public void update(ObservableCounter counter) {
    textField.setText("" + counter.getValue());
  }
}
```

Its constructor receives the JTextField instance. The update() method called in response to the counter change simply calls the setText() method in the instance to update the text. The expression, “” + counter.getValue() is a convenient way to convert an int to a String. The + operation used here takes String and int arguments, and return a concatenation of the String with a string representation of the int. As we have passed it an null String, the result is a String representation of the int.

We can summarize the steps related to a view as follows.

Before any read or write methods are called in the model:

1. The view is registered with its models.
2. The view creates or receives references to output widgets it updates.

When a model write method is called:

1. A notification method is called in the view.
2. The method might call read methods in the model in case the arguments to the notification method do not sufficiently describe the change.
3. The method updates relevant output widgets.

Of course a notification method does have to directly handle steps 2 and 3 - it can call helper methods to do so.

Based on this example and step summary, widget-based output is not more difficult than console or message –based output, with the only complication being the possibility of there being many output widgets and the need to know which widget should be updated in response to a particular notification. As in this example, it is rare for there to be multiple output widgets in a single view, though there may be many views with different output widgets.

Widget-based Input is much more complicated!

Widget Events → Controller Notification Methods → Model Write Methods

Input is handled in a fundamental different way in toolkits. In console input, the caller stops all computation and “blocks” until the user has input the expected value. Thus, when the controller calls:

```
int nextInput = scanner.nextInt();
```

the caller blocks until the input is received – it can take no steps while the user is providing input.

We could imagine a similar input scheme for widgets. For example, to receive input from a button we could call:

```
button.waitForClick()
```

If there was only one input widget in an application, and output occurs only in response to input from this widget, this approach would work. While the program is blocked no output can occur and no input can be received from another widget. In most programs there are many input widgets and in distributed display can change in response to external events. In these applications we cannot use blocking input.

Thus, rather than using a blocking input approach in which a controller pulls input from a widget, we will use a non-blocking on event-based input approach in which a widget pushes input to the controller. A pushed input is called an event, hence the term event-based input.

The question now is how does a widget push information to a controller? If the widget knows about a controller, it could call write methods in a controller, much as a controller calls write methods in a model. This approach works in the controller-model case because we write custom controllers for models. Thus, these controllers have references to and know about the write methods of the models for which they have been written. The widgets provided by the toolkit are application-independent – hence the term toolkit. Thus, they have no knowledge of the write methods in the controller.

A common approach to address to receive data from an external class is to subclass it and override appropriate methods in it. We will see this approach when we study paint methods.

Let us illustrate this approach using JButton. The button has to take some action on being pressed – it need to simulate the action button press. A controller can override the method that takes this action, call the super class version of it, and then call the appropriate model method. In general, in any widget, there is some method that reacts to input by giving the end user feedback. The feedback method can be overridden to call model methods. If a controller has multiple input widgets, it can create a subcontroller for each widget that overrides the appropriate method of the widget.

This approach was used in the Smalltalk toolkit, which in turn, influenced several later toolkits including the Java 1.0 toolkits. It has two related disadvantages. First, it leads to proliferation of classes as a separate subclass must be created for each input widget in a user-interface. Second, this approach leads to an abuse of the IS-A principle. Arguably, a (sub) controller is not a widget, it is an agent for converting widget events into model methods. It should not have to know what about the feedback producing methods of widgets. The correct relationship between a controller and a widget is HAS-A.

In Java 1.1, the observer pattern was applied to widget-controller interaction. Like a model, a widget is an observable object that produces information. Widget-specific controllers can become observers of it. Thus, a widget now allows controller observers to be registered and calls application-independent notification methods in them. These notification methods in the controllers then call application-dependent methods in the model.

The MVC pattern and its sub-pattern, the observer pattern, were invented by the designers of the Smalltalk-80 environment. Yet, in that environment, the observer pattern was not used to receive data from input widgets. The fact that the designers of a pattern did not apply it in all situations they encountered underscores the difficulty of object-oriented programming and identification and use of design patterns.

ActionEvent Observables and Observers

Several observer interfaces are provided by Java AWT/Swing toolkits – their details are not relevant here. One illustrative interface, define by the java.awt package, is:

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

Several observable widget classes allow instances of this interface to be added as observer using the following method header:

public void addActionListener (ActionListener actionListener);

Unfortunately, this method header is not defined in any interface. Thus, we have an interface for ActionEvent observers but not ActionEvent observables, much as we saw in the design of the Java Observer/Observable types earlier. This limitation reduces reusability, even in our counter application, as we see below.

ActionEvent observables call the actionPerformed() notification method in each registered observer when the appropriate user action occurs. The action in button widgets is button press and in text field widgets is hitting the Enter key after inputting text. The argument gives details of the user action, which we will ignore for now.

The following implementation of the widget-based controller illustrates an ActionEvent observable:

```
public class ACounterJButtonController implements CounterController,
ActionListener {
  JButton button;
  ObservableCounter counter;
  public ACounterJButtonController(JButton theButton) {
    button = theButton;
    button.addActionListener(this);
  }
  public void setModel (ObservableCounter theCounter) {
    counter = theCounter;
  }
  // do not need to implement anything in this controller
  public void processInput() {
  }
  //ActionListener method
  public void actionPerformed(ActionEvent arg0) {
    counter.add(1);
  }
}
```

Like the view, our controller is passed in its constructor the relevant widget. It becomes an observer of it by calling the method addActionListener(). When the ActionEvent notification method is called, it ignores the argument, like many obsevers do, and calls the model write method to increment the counter.

Here, we have violated our rule of implementing a single interface, which should have been an extension of the two interfaces shown here, simply to reduce the space that would have been required to show the interface. One of these interfaces is, of course, the ActionListener interface. The other is CounterController, which was also implemented by the console-based controller we defined earlier. It defined a method to pull input from the I/O library. As input is being pushed to this event-based controller, this method does nothing in this controller. A cleaner design would create another interface without this method that is subtyped by the console controller interface.

UI and MVC Composition

In the console application, our main program composed the model view(s) and controller, connecting them to each other. In particular, it created these objects, called the setModel() method in the controller, and the addCounterObsever() registration method in the model. The main program performs the same steps here, except that it creates a different controller and view. In addition, it composes the the GUI, creating a tree rooted by an instance of JFrame, adding appropriate layout managers, and making the tree visible. The following is the complete main:

```
public class SingleFrameAWTComposer {
  public static void main (String args[]) {
  // compose AWT components
    JFrame frame = new JFrame("AWT Controller and View");
    JButton button = new JButton("Increment");
    JTextField textField = new JTextField();
    frame.setLayout(new GridLayout(2,1));
    frame.add(textField);
    frame.add(button);
    frame.setSize(200, 200);
    frame.setVisible(true);
    //compose model view and controller
    ObservableCounter model = new AnObservableCounter();
    CounterObserver view =
                    new ACounterJTextFieldView(textField);
    model.addObserver(view);
    CounterController controller =
                    new ACounterJButtonController(button);
    controller.setModel(model);
  }
}
```

The code shows the relationship between these two tasks – the button and textfield widgets created by the UI composition code need to be passed to MVC composition code, implying that the GUI should be created first.

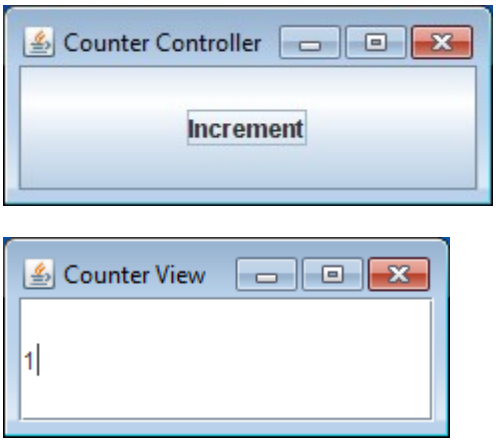
View Controller Coupling

In this example, the view and controller are passed different widgets. In most applications, the same widget might be used for input and output. For example, an application could display a text field showing the current counter value that can be edited to set the counter to a new value. Because of this coupling between the input and output widgets, many advocate the model-interactor model. As this

example shows, this coupling is irrelevant as long as the controller and views do not create the widgets with which they interact - the controllers and views could be passed overlapping widget sets.

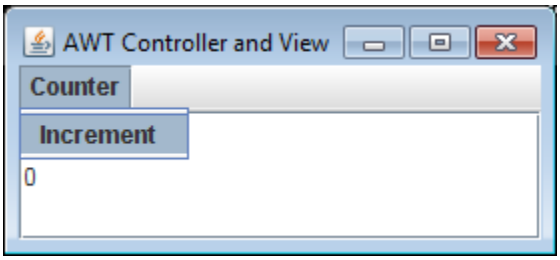
Power of Separation of Separate Composer and Interfaces

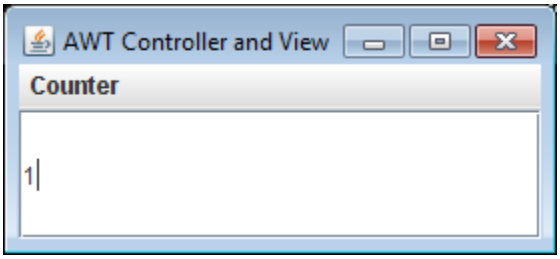
Let us use our simple example to show the power of decomposing code into separate classes. We have already seen that by separating the model from the UI code, we can reuse it in the three counter user-interfaces we saw in the MVC introduction chapter. By separating the controller from the view, we could share a single controller in all three UIs and each of the views in two UIs. The user-interface in Fgure ??? shows the benefit of separating widget creation from views and controllers.



As in Figure ?, a JTextField instance is used to display the counter and a JButton instance to increment it. The difference is that each of these widgets is displayed in a separate frame. To implement this UI, we need to change only our composer. The view and controller we created for Figure ??? (and of course, the model) remain the same, as the widgets with which they interact are the same. As the same models, views, and controllers can be used with different UI composition code, the separation of concerns principles tells us that the UI composer should be in a separate class. (Is it useful to separate MVC and UI composition in different classes?)

Here is an illustrative variation of the user-interface in which the button is replaced by a menu item.





The composer, of course, must now change to create a menu rather than a button. To show how menus are added, here is the relevant code:

```
public static void main (String args[]) {
  // compose AWT components
    JFrame frame = new JFrame("AWT Controller and View");
    JMenuItem menuItem = new JMenuItem("Increment");
    JMenu menu = new JMenu("Counter");
    JMenuBar menuBar = new JMenuBar();
    menu.add(menuItem);
    menuBar.add(menu);
    …
    //compose model view and controller
    CounterController controller = new ACounterJMenuItemController(menuItem);
    …
   }
 }
```

Interestingly, the controller also has to redone. The new controller is a copy of the previous controller except that it declares and initialized an instance of JMenuItem instead of JButton:

```
public class ACounterJMenuItemController
               implements CounterController, ActionListener {
  JMenuItem menuItem;
  ObservableCounter counter;
  public ACounterJMenuItemController(JMenuItem theMenuItem) {
    menuItem = theMenuItem;
    menuItem.addActionListener(this);
  }
  …
}
```

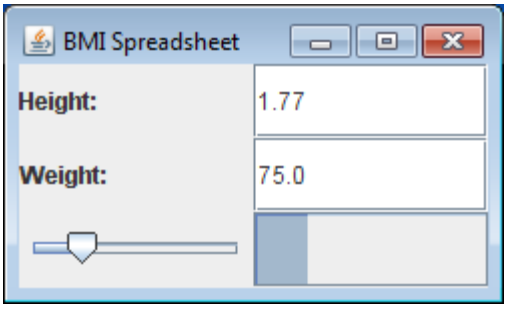
In both cases, it calls the addActionListener() method in the widget. If this method was in a common interface, say ActionLiistenerRegistrar, then we could have typed the widget using this interface instead of the two classes, and reused the same controller for both Figure ??? and ???:

```
public class ACounterActionBasedController
               implements CounterController, ActionListener {
  ActionListenerRegistrar inputWidget;
  ObservableCounter counter;
  public ACounterJMenuItemController(ActionListenerRegistrar anInputWidget) {
    inputWidget = anInputWidget;
    inputWidget.addActionListener(this);
  }
  …
}
```

The Java designers aparently did not imagine such reuse, and, in general it is difficult to imagine all possible ways in which our code may be reused later. This is the reason we have a rule that we should define an interface for each logical set of methods in our class.

Connecting Complex Model and Widget Structures

To illustrate what might happen when we go from a single-component model to a more multi-component one, let us consider the three-component BMISpreadsheet model. Figure ??? shows a user-interface to interact with it.



It is much like the one created by ObjectEditor. The height and weight are shown in labelled text fields. An important difference is that the BMI is value no longer shown in a single text widget. Instead, a "slider" and "progress bar" widget shown its vale with the position of the sider and the size of the grey bar, respectively. A single text-field is perhaps more useful; the goal of this example is to show that the same value can be output using multiple kinds of output widgets, which can be shown simultaneously.

Our goal now is to connect the BMI model structure to a widget structure that creates the user-interface for Figure ???. This structure uses, of course, the JTextField widget we saw earlier. In addition, it used two JLabel widgets to label the height and weight, and a single instance of the JSlider and JProgressBar widget classes.

This time, we will not make the input and output widgets direct descendants of the frame. We will group each text field and its corresponding label widget into an instance of a JPanel containing widget, and the two widgets displaying the BMI into another JPanel. The three JPanels will then become the components of the JFrame. This widget structure shows the logical relationship between the input and output widgets, which in turn, allows us to control the layout of each logical group independently. The following code creates the UI:

```
public class BMIFrameComposer {
  static JFrame frame = new JFrame("BMI Spreadsheet");
  static JTextField heightField = new JTextField();
  static JLabel heightLabel = new JLabel("Height:");
  static JPanel heightPanel = new JPanel();
  static JTextField weightField = new JTextField();
  static JLabel weightLabel = new JLabel("Weight:");
  static JPanel weightPanel = new JPanel();
  static JSlider bmiSlider = new JSlider();
  static JProgressBar bmiProgressBar = new JProgressBar();
```

```
  static JPanel bmiPanel = new JPanel();
  public static void main (String args[]) {
    composeLabelledField(heightPanel, heightLabel, heightField);
    composeLabelledField(weightPanel, weightLabel, weightField);
    composeBMI();
    composeFrame();
    composeMVC();
  }
public static void composeBMI() {
    bmiPanel.setLayout(new GridLayout(1, 2));
    bmiPanel.add(bmiSlider);
    bmiPanel.add(bmiProgressBar);
  }
  public static void composeFrame() {
    frame.setLayout(new GridLayout(3, 1));
    frame.add(heightPanel);
    frame.add(weightPanel);
    frame.add(bmiPanel);
    frame.setSize(250, 150);
    frame.setVisible(true);
  }
  public static void composeMVC() {
    BMISpreadsheet bmiSpreadsheet =
         new AnObservableBMISpreadsheet();
    new ABMISpreadsheetController(bmiSpreadsheet,
         heightField, weightField);
    PropertyChangeListener bmiSpreadsheetView =
    new ABMISpreadsheetView(heightField, weightField,
         bmiSlider, bmiProgressBar);
      bmiSpreadsheet.
      addPropertyChangeListener(bmiSpreadsheetView);
  }
}
```

This time we have broken the main into separate methods to group its steps into independent units. As before, the main also composes the MVC structure.

This structure still consists of a single view and controller, though now multiple output and input widgets, respectively, are passed to their constructors. The text-field widget are used both for input and output, so they are passed to both. The label widgets are purely cosmetic, they provide users with "syntactic sugar" to understand the display and do not provide input or output functions. Neither do the JPanel widgets, which serve only to group other widgets in the user-interface.

Recall that our ObservableBMISpreadsheet calls the propertyChange() notification method in view when the Height, Weight or BMI property is changed. The argument of the method is an instance of PropertyChangeEvent, which indicates the name of the property, and its old and new value. The method determines which property has changed, and updates the corresponding widget(s) with the new property value. Because ActionEvent is a generic event applying to JButton, JTextField, and JMenuItem and other widgets, it does not give widget-specific information about the input. Therefore, the

controller calls the getText() read method in JTextField() to determine the input String, and a standard method to convert the read String to a double.

```
public class ABMISpreadsheetView implements PropertyChangeListener{
  JTextField heightField, weightField;
  JSlider bmiSlider;
  JProgressBar bmiProgressBar;
  public ABMISpreadsheetView (JTextField theHeightField,
           JTextField theWeightField, JSlider theBMISlider,
           JProgressBar theBMIProgressBar) {
    heightField = theHeightField;
    weightField = theWeightField;
    bmiSlider = theBMISlider;
    bmiProgressBar = theBMIProgressBar;
  }
  public void propertyChange(PropertyChangeEvent event) {
    String propertyName = event.getPropertyName();
    Double newValue = (Double) event.getNewValue();
    if (propertyName.equalsIgnoreCase("height")) {
      heightField.setText(newValue.toString());
    } else if (propertyName.equalsIgnoreCase("weight")) {
       weightField.setText(event.getNewValue().toString());
    } else if (propertyName.equalsIgnoreCase("bmi")) {
       double newBMI = newValue;
       bmiSlider.setValue((int) newBMI);
       bmiProgressBar.setValue((int) newBMI);
    }}}}
```

As we see in the code, the JSlider and JProgress widgets provide the setValue() method, which is an int. The view converts the double value to an int before displaying in these widgets.

Like the view, the controller is also an observer, of two JTextField instances rather than a single model. It registers itself as an observer of both objects. The notification method, as in the previous conrtroller, is actionPerformed(). Like the arguments of other notification methods, the argument of actionPerformed indicates the observable that changed. The method uses the identity of the text-field that changed to call the appropriate setter in the model.

```
public class ABMISpreadsheetController implements ActionListener {
  JTextField height, weight;
  BMISpreadsheet bmiSpreadsheet;
  public ABMISpreadsheetController (
        BMISpreadsheet theBMISpreadsheet,
        JTextField theHeight, JTextField theWeight) {
    height = theHeight;
    weight = theWeight;
    bmiSpreadsheet = theBMISpreadsheet;
    height.addActionListener(this);
    weight.addActionListener(this);
   }
  public void actionPerformed(ActionEvent event) {
    JTextField source = (JTextField) event.getSource();
    String text = source.getText();
    double val = Double.parseDouble(text);
```

```
    if (source == height) {
      bmiSpreadsheet.setHeight(val);
    } else {
      bmiSpreadsheet.setWeight(val);
    }
  }
}
```

Summary of Widget-based MVC

An MVC pattern is tied to a set of I/O methods provided by some application-independent user-interface library. To create a WIMP user-interface, the I/O library to use is a user-interface toolkit, which provides predefined widgets to perform input and or output. These widgets determines the nature of the controller and view, but not the model, or the controller-model or model-view communication. The controller now also becomes an observer, not of the model but the input widgets in the user-interface. The following sequence of actions occurs when a user interacts with an input widget:

1. Controller notification: The input widget calls a notification method in the controller, passing it information about the event.
2. Widget determination by controller: A controller may be an observer of multiple input widgets, all of which call the same notification method. Therefore the controller may use the event information to determine which write method to call.
3. Widget reads by controller: The event may not give enough information about the actual input event, in which case the controller calls read methods in the widgets to get the appropriate information.
4. Model writes by controller: Like any controller, the widget-based controller uses the input information to call one or more write methods in the model.
5. Observer notification: The model, of course, calls notification methods in its observing views. In your assignments, it will call the propertyChange() method,
6. Model component determination by view: A single notification may be called for multiple components of the model. Therefore, the view might need to use the notified event to determine which component changed. This task is not peculiar to widget-based views – even a console-based view listening to our observable BMI spreadsheet would need to perform it. Model reads by view: In case the notification method does not provide enough information, the view calls read methods in the model. Fortunately, a PropertyChangeEvent gives us sufficient information, and does not require the read calls.
7. Widget writes by views: Finally, the view calls appropriate write methods in the widget(s) displaying the changed component such as setText() in JTextField and setValue() in JSlider and JProgressView.

To enable this event flow, both the UI and MVC structure need to be composed. These two composition tasks are related in that the input and output widgets in the UI structure need to be passed to the controller and views respectively. The same widget can be used for both input and output. This does not

mean that the controller and view should be integrated in an interactor. A separate class can perform most of the composition task, which allows the same views and controllers to be used for different user-interface structures. The part of the composition the composer does not do is making the controller an observer of the input widgets.

In the UI creation task:

1. A top-level widget object such as a JFrame or Frame instance roots the display structure.
2. A subset of the leaf widgets such as JTextField, JSlider and JMenuItem provide the input and output behavior. Other leaf widgets provide "syntactic sugar."
3. A hierarchy containers such as JPanel instances form the internal nodes in the display structure, providing logical grouping of syntactic sugar and I/O widgets.
4. Layout objects determine positioning and size of the container and leaf widgets.

In the MVC composition task:

1. As before, the controllers are connected to the model, and the views are registered with the model.
2. The controller is passed the input widgets it observers. The controller registers itself as an observer of these widgets.
3. The view is passed the output widgets it updates.

Modularization and Indirection → Ease of Understanding?

Toolkit-based MVC illustrates the nature and power modularizing a program into multiple classes. We have argued and demonstrated here that it leads to reusability. It is also argued that modularization leads to ease of understanding. A programmer must understand one class at a time rather than a monolithic class. Is this really true?

Consider the code of the original console-based counter we created:

```
public class MonolithicConsoleUI {
        static int counter = 0;
        static Scanner scanner = new Scanner(System.in);
        public static void main(String[] args) {
                while (true) {
                        System.out.println("Counter: " + counter);
                        int nextInput = scanner.nextInt();
                        if (nextInput == 0) return;
                                counter += nextInput;
            }
      }
```

It is trivial to understand this code. In this code, a class does all of the required tasks. By understanding the method call relationships in this one class, we can get a good picture of the algorithm. When we

modularize a class into multiple classes, we must, of course, not only understand each class, but also the relationship between these classes. In the monolithic case, actions occur directly in one class, possibly in one method. In the example above, the main method that reads input also updates the counter and produces output. In the modularization case, actions occur indirectly, involving multiple classes. In our MVC-based implementation of this user-interface, a controller talks to the model, which communicates with the view, which may communicate back with the model, before the output is produced. It is difficult to get an idea of the overall algorithm, though we may better understand the individual pieces.

This problem with modularization is reduced when:

1. We go to more complex programs, as understanding a large monolithic class is difficult, even it it is divided into methods,
2. Use well known design patterns, as the nature of the component communication is learnt once for a large set of applications.
3. We provide documentation about the control flow. Therefore, it is more important to document he relationship between the classes in a package, through package-info, than to comment the individual methods and classes.

Yet understanding heavily modularized programs, even those written by us, is challenging. Often we have to use sophisticated search and debugging capabilities to identify which class is responsible for some task.

Calls vs. Call backs

Usually, invocation of a method by an application component on a library component such as System.out or a JTextField is termed a call. The invocation of a method by a library component on an application component is termed a callback. It is so named because a library component does not know about application components (while the reverse is, of course, true). So the application component must register itself with the system component, requesting the return call, or callback. Thus, the actionPerformed() method called by a JButton on our controller is an example of a callback.

Here a component L is library for component C, if C knows about L, but L does not know about C. Both L and C can be written by the same team of programmers, and L may not be a well-distributed library. By this definition, a model is a library component for a view component, and any view notification method is also a callback.

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).