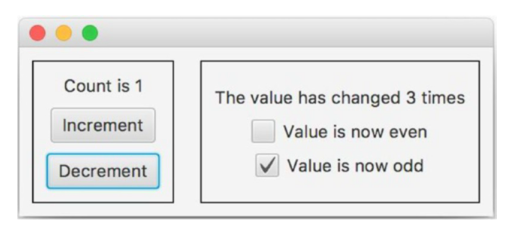
**The MVC Pattern**

Many GUI applications rely on the MVC pattern to synchronize their views. To illustrate see following example:



The two views each have a pane within the window. The second view is a “watcher” view. It keeps track of how many times the count was changed, and displays whether the count is even or odd. But how can the watcher view know when the model has changed? The answer is to use the observer pattern! The model needs to broadcast changes made by the counting view so that the watcher view can observe them. The model therefore needs to be modified to be an observable. Define an observer interface, called CountObserver. This interface will have one observer method, update, which pushes the new count to its observers. Its code appears in Listing 1

Listing 1. The CountObserver Interface

public interface CountObserver {

public void update(int count);

}

The class CountModel needs to manage an observer list. Its updateBy method will broadcast the new count to the observers on the list.

public class CountModel implements Observable {

private int count = 0;

private Collection<CountObserver> observers = new ArrayList<>();

public void addObserver(CountObserver obs) {

observers.add(obs);

}

private void notifyObservers(int count) {

for (CountObserver obs : observers)

obs.update(count);

}

public void updateBy(int n) {

count += n;

notifyObservers(count);

}

public int getCount() {

return count;

}

}

The watcher’s controller en the counter’s controller will be the model observers. When the controller receives a notification from the model, it will determine the changes that need to be made to its view and pass those changes to the view. For being able to do this the controller needs a reference to the view, and the view needs to have a method that the controller can call. By giving the controller a reference to its view, the view and its controller will have references to each other. The view gets its reference via constructor injection and the controler defines a method setView. When the view is created, it can call the controller’s setView method, passing the controller a reference to itself. The view defines the methods for the controller to call and to update the view.

Listing 3 gives the code for the controller. Note that the controller is responsible for keeping track of the number of times the model changes, because I decided that this value is not relevant to the model. If you feel otherwise, you should change the model so that it keeps that information.

Listing 3. The WatcherController Class

public class WatcherController implements CountObserver {

private WatcherView view;

private int howmany = 0;

public WatcherController(CountModel model) {

model.addObserver(this);

}

// called by the view

public void setView(WatcherView view) {

this.view = view;

}

// called by the model

public void update(int count) {

howmany++;

boolean isEven = (count%2 == 0);

boolean isOdd = !isEven;

String msg = "The count has changed " + howmany + " times";

view.updateDisplay(msg, isEven, isOdd);

}

}

Listing 4 gives the code for the watcher view. Its constructor calls the controller’s setView method, thereby establishing the two-way connection between the view and controller. The updateDisplay method ùàsets the value of the view’s three controls. Note that the view has no idea what these values mean.

Listing 4. The WatcherView Class

class WatcherView {

private Label lbl = new Label("The count has not yet changed");

private CheckBox iseven = new CheckBox("Value is now even");

private CheckBox isodd = new CheckBox("Value is now odd");

private Pane root;

public WatcherView(WatcherController controller) {

root = createNodeHierarchy();

controller.setView(this);

}

public Pane root() {

return root;

}

public void updateDisplay(String s, boolean even, boolean odd) {

lbl.setText(s);

iseven.setSelected(even);

isodd.setSelected(odd);

}

private Pane createNodeHierarchy() {

iseven.setSelected(true);

isodd.setSelected(false);

VBox p = new VBox(8);

p.setAlignment(Pos.CENTER);

p.setPadding(new Insets(10));

p.getChildren().addAll(lbl, iseven, isodd);

return p;

}

}

The main program, Count3, configures the two views into a single window. In order to deal with the multiple views, the code places the node hierarchies of the two views into a single HBox pane. Listing 5 gives the code.

Listing 5. The Count3 Class

public class Count3 extends Application {

public void start(Stage stage) {

CountModel model = new CountModel();

// the first view

CountController ccontroller = new CountController(model);

CountView cview = new CountView(ccontroller);

// the second view

WatcherController wcontroller = new WatcherController(model);

WatcherView wview = new WatcherView(wcontroller);

// Display the views in a single two-pane window.

HBox p = new HBox();

BorderStroke bs = new BorderStroke(Color.BLACK,

BorderStrokeStyle.SOLID,

null, null, new Insets(10));

Border b = new Border(bs);

Pane root1 = cview.root(); Pane root2 = wview.root();

root1.setBorder(b); root2.setBorder(b);

p.getChildren().addAll(root1, root2);

stage.setScene(new Scene(p));

stage.show();

}

public static void main(String[] args) {

Application.launch(args);

}

}

Listing 6. The CountController Class

class CountController implements CountObserver {

private CountModel model;

private CountView view;

public CountController(CountModel model) {

this.model = model;

model.addObserver(this);

}

// Methods called by the view

public void setView(CountView view) {

this.view = view;

}

public void incrementButtonPressed() {

model.updateBy(1);

}

public void decrementButtonPressed() {

model.updateBy(-1);

}

// Method called by the model

public void update(int count) {

view.setLabel("Count is " + count);

}

}

Listing 7. The CountView Class

class CountView {

private Label lbl = new Label("Count is 0");

private Pane root;

public CountView(CountController controller) {

root = createNodeHierarchy(controller);

controller.setView(this);

}

public Pane root() {

return root;

}

public void setLabel(String s) {

lbl.setText(s);

}

private Pane createNodeHierarchy(CountController cont) {

Button inc = new Button("Increment");

Button dec = new Button("Decrement");

... // create the node hierarchy, having root p

inc.setOnAction(e -> cont.incrementButtonPressed());

dec.setOnAction(e -> cont.decrementButtonPressed());

return p;

}

}

Consider what now happens to the count view and count controller when the Increment button is clicked.

• The view calls the controller’s incrementButtonPressed method.

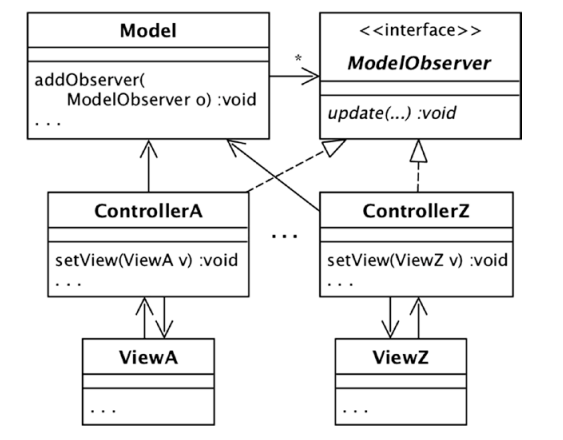
• That method calls the model’s updateBy method.

• That method updates the count and calls notifyObservers, which calls the controller’s update method.

• That method formats the string for the view to display, and calls the view’s setLabel method.

• That method modifies the text of its label to be the current count.

The insight that controllers should be observers is the basis for the MVC Design Pattern. This pattern asserts that an application should be structured similarly to Count3. In particular: **the model should be an observable and all controllers should be model observers; the controllers talk directly to the model; and each view/controller pair can talk directly to each other**. This pattern is expressed by the following class diagram.



**Communication using the MVC pattern works as follows:**

* **An action on a view (such as a button click) gets communicated to its controller.**
* **The controller translates that action to a method call on the model.**
* **If that method call is a request for data, then the model returns the requested data directly to the controller, which forwards it to its view.**
* **If that method call causes the model to change, the model notifies its observers.**
* **Each controller, being a model observer, decides if the update is relevant to its view. If so, it calls the appropriate view methods.**