Java’s Model-View approach

In Lectures 9 and 10, we saw how VisualAge arranges for visual components to be updated when their underlying model changes. This is done using models and views.

Java has a similar mechanism, also using models and views. In Java,

- the model is an instance of `Observable`.
- the view is an implementation of `Observer`.

As in Smalltalk’s MVC, responsibility is divided as follows:

- Each model keeps track of the views that display the model’s instance-variable values.
- Whenever a model’s instance variable’s value changes, all the dependent views are notified of the change.
- Each dependent view, when it is notified of a change, interrogates the model’s instance-variable values to determine how it should change.

Each `Observable` has three or four key methods:

- `addObserver` attaches a view to this model;
- `deleteObserver` detaches a view;
- `setChanged` is called to indicate that a noteworthy change has occurred; and
- `notifyObservers` is used to announce to views that a noteworthy change has occurred, provided that `setChanged` has been called since the most recent call to `notifyObservers`.

Each `Observer` needs to implement one key method:

- `update`, which describes what is to be done when a view receives a change notification from the model.
**Example: Account manager**

Here is an example (from Deitel, *Advanced Java 2 Platform*) of an account manager.

- **Account** extends **Observable**, and implements the model. It simply contains the name of the owner and the balance.
- Two views are defined on **Account**:
  - **AccountTextView** and
  - **AccountBarGraphView**.
- **AccountController** accepts text input indicating the amount to deposit or withdraw, and then modifies the account.

Here is the definition of class **Account**. Since it is a model, it inherits from **Observable**.

```java
import java.util.Observable;

public class Account extends Observable {
    private double balance; // Acct. balance
    private String name;    // Name of owner

    public Account(String accountName,
                    double openingDeposit) {
        setBalance(openingDeposit);
    }

    private void setBalance(double newBalance) {
        balance = newBalance;
    }

    private void getBalance() {
    }
}
```
public void withdraw(double amount) throws IllegalArgumentException {
    if (amount < 0)
        throw new IllegalArgumentException(
            "Cannot withdraw negative amount");
    setBalance(getBalance() - amount);
}

public void deposit(double amount) throws IllegalArgumentException {
    if (amount < 0)
        throw new IllegalArgumentException(
            "Cannot deposit negative amount");
}

public String getName() {
    return name;
}

Note that all the setter methods need to call setChanged() and notifyObservers().

Since we have more than one view, let us define an AbstractAccountView class to encapsulate methods common to the views.
import java.util.*;
import java.awt.*;
import javax.swing.*;

public abstract class AbstractAccountView
    extends JPanel implements Observer {
    private Account account; // Acct. to observe

    public AbstractAccountView(Account newAccount)
        throws NullPointerException {
        if (newAccount == null)
            throw new NullPointerException();
        account = newAccount;
        // Make this AbstractAccountView an observer
        // of the account it is supposed to observe.
        account.addObserver(this);
        setBackground(Color.white);
    }

    public Account getAccount() {
    }

    protected abstract void updateDisplay();
    // Update display with current account balance.

    public void update(Observable obs, Object obj) {
        updateDisplay();
    }

    Note that our update method ignores both of its arguments.

    • If the observer were observing more than one object, the
      first parameter would tell it which object issued the update.
    
    • The second argument is a parameter that an Observable
      object could pass to the notifyObservers method. It might
      contain information about the change that was made.

    The first of our views is AccountTextView.
import java.util.*;
import java.text.NumberFormat;
import javax.swing.*;

public class AccountTextView extends AbstractAccountView {
    private JTextField balanceTextField = new JTextField(10);
    private NumberFormat moneyFormat = NumberFormat.getCurrencyInstance(Locale.US);
    // Number format for US dollars

    public AccountTextView(Account account) {
        super(account);

        // Specify that text field is read-only ...
        balanceTextField.setEditable(false);
        add(new JLabel("Balance: "));
        add(balanceTextField);
        updateDisplay();
    }
}

Why did we need to define a constructor that merely called the superclass constructor?

The second view is the AccountBarGraphView. It shows a positive balance in black and a negative balance in red ...

The coordinate system in Java has its origin (0, 0) at the upper left, with coordinates increasing downward and toward the right. Coordinates are measured in pixels relative to the origin.

In order to draw on the screen, a program must use a graphics context. A Graphics object manages a graphics context, controlling how information is drawn.

Graphics is an abstract class. It is implemented differently on different platforms. It provides a common interface so that programs can be written to run on every platform. All drawing in Java must go through a Graphics object.
To find a component’s Graphics object, use the component’s `getGraphics()` method.

A few of the properties represented in a Graphics object are—

- the current color,
- the current font,
- the Component object on which to draw.

The Component class in package `java.awt` is the superclass of many widgets that can appear on a screen, such as—

- windows,
- buttons,
- canvases,
- checkboxes,
- labels, and
- lists.

Component’s method `paint` takes a Graphics object as an argument.

The system passes this object to `paint` when the Component is to be displayed on the screen. The header for the paint method is

```java
public void paint(Graphics g)
```

Normally, `paint` is called automatically by the system when a Component is to be first displayed. Only if an event changes the display (e.g., by resizing the Component) will `paint` be called again.

Instead of calling `paint` directly, the programmer should call

```java
public void repaint()
```

`repaint` calls Component’s `update` method to clear the background of any previous drawing, and then calls `paint`.

*Drawing lines*
Java calls the `paint` method whenever it determines that a window should be drawn or redrawn.

A line can be drawn on a graphics context `g` like this:

```java
g.drawLine(0, 50, 100, 50);
```

This draws a line from (0, 50) to (100, 50).

**Writing text**

Text can be written to a canvas using the `drawString` method of its graphics context.

```java
String title = "Hello, CSC 517!";
...
g.drawString(title, 100, 100);
```

The last two coordinates tell where the lower left corner of the string is to be placed.

If you draw a string at (0, 0), it will not be visible.

The string is written in a font specified by the `setFont` method.

```java
public void paint(Graphics g) {
    g.setFont(new Font("Helvetica", Font.BOLD, 12));
    g.drawString(title, 100, 100);
}
```

You can also specify `font.PLAIN` or `font.ITALIC`.

Now we are ready to present the class `AccountBarGraphView`. 
import java.awt.*;
import javax.swing.*;

public class AccountBarGraphView
    extends AbstractAccountView {

    public AccountBarGraphView(Account account) {
        super(account);
    }

    public void paintComponent(Graphics g) {
        // Draw account balance as a bar graph
        super.paintComponent(g);
        double balance = getAccount().getBalance();

        // Bar axis will be 200 pixels wide & represent
        // balances between $-10,000 and $10,000. Here
        // we calculate the length of the bar.
        int barLength = (int) ((balance/20000.0)*200);

        if (balance >= 0.0) {
            g.setColor(Color.black);
            g.fillRect(105, 15, barLength, 20);
        } else {
            g.setColor(Color.red);
            g.fillRect(105+barLength, 15, -barLength, 20);

        // Draw vertical & horizontal axes.
        g.setColor(Color.black);
        g.drawLine(5, 25, 205, 25);
        g.drawLine(105, 5, 105, 45);

        // Draw graph labels.
        g.setFont(new Font("SanSerif", Font.PLAIN, 10));
        g.drawString("-$10,000", 5, 10);
        g.drawString("$0", 110, 10);
        g.drawString("$10,000", 166, 10);
    }

    public void updateDisplay() {
        repaint();
    }

    // Get minimum, maximum, & preferred sizes for view.
    public Dimension getPreferredSize() {
        return new Dimension(210, 50);
    }

    public Dimension getMinimumSize() {
        return getPreferredSize();
    }

    public Dimension getMaximumSize() {
        return getPreferredSize();
    }
}
The point of defining `getMinimumSize` and `getMaximumSize` is to override the methods defined in `JComponent` and thereby prevent the layout manager from resizing the graph.

Finally, we come to the `AccountController` class. It extends `JPanel` because it provides a GUI interface for depositing and withdrawing funds.

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```
public class AccountController extends JPanel {

    private Account account; // Account that it controls.
    private JTextField amountField; // Enter amount here

    public AccountController(Account controlledAccount) {
        super();
        account = controlledAccount;
        amountField = new JTextField(10);

        JButton depositButton = new JButton("Deposit");
        depositButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent ev) {
                    try {
                        // Deposit the amount specified by user.
                        account.deposit(Double.parseDouble(
                            amountField.getText()));
                    } catch (NumberFormatException e) {
                        JOptionPane.showMessageDialog(
                            AccountController.this,
                            "Please enter a valid amount.",
                            "Error", JOptionPane.ERROR_MESSAGE);
                    }
                }
            });

        JButton withdrawButton = new JButton("Withdraw");
        withdrawButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent ev) {
                    try {
                        // Withdraw the amount specified by user
                        account.withdraw(Double.parseDouble(
                            amountField.getText()));
                    } catch (NumberFormatException e) {
                        JOptionPane.showMessageDialog(
                            AccountController.this,
                            "Please enter a valid amount.",
                            "Error", JOptionPane.ERROR_MESSAGE);
                    }
                }
            });

        // Lay out components
        setLayout(new FlowLayout());
        add(new JLabel("Amount: "));
        add(amountField);
        add(depositButton);
        add(withdrawButton);
    }
}
What method actually makes the deposits and withdrawals in this class?

**MVC in Swing**

[Core Java, pp. 379–380] Most Swing components have a model implemented by an interface whose name ends in `Model`. E.g., the `Button` class has the interface `ButtonModel`.

`ButtonModel` is implemented in only one class, `DefaultButtonModel`.

It contains a number of methods for finding out certain things about the button.

These include—

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getActionCommand()</code></td>
<td>The action command string associated with this button.</td>
</tr>
<tr>
<td><code>getMnemonic()</code></td>
<td>The keyboard character that selects this button</td>
</tr>
<tr>
<td><code>isArmed()</code></td>
<td>True if the button was pressed and the mouse is still over the button.</td>
</tr>
<tr>
<td><code>isEnabled()</code></td>
<td>True if the button is selectable</td>
</tr>
<tr>
<td><code>isPressed()</code></td>
<td>True if the button was pressed but the mouse button hasn’t yet been released.</td>
</tr>
<tr>
<td><code>isRollover()</code></td>
<td>True if the mouse is over the button.</td>
</tr>
<tr>
<td><code>isSelected()</code></td>
<td>True if the button has been toggled on (used for checkboxes and radio buttons)</td>
</tr>
</tbody>
</table>
We saw most of these features at the end of the last class in the “How to Use Buttons” page at http://java.sun.com/docs/books/tutorial/uiswing/components/button.html.

The button model maintains three pieces of state information:

- pressed/not-pressed,
- armed/not-armed, and
- selected/not-selected.

These are boolean quantities.

A button is pressed if the user has pressed the mouse button while the mouse cursor is over the button, but has not yet released the mouse button. The button is pressed even if the user drags the mouse cursor off the button.

A button is armed if the button is pressed and the mouse cursor is over the button.

Some buttons may also be selected. This value is typically toggled on and off by repeatedly pressing the button.

Each JButton stores a ButtonModel object, which you can retrieve:

```java
JButton button = new JButton("Blue");
ButtonModel model = button.getModel();
```

Rarely do you have to query the ButtonModel directly. The JButton class (actually, AbstractButton) provides access to most items of interest, such as button state.

When using the Metal look and feel, the JButton class that uses ButtonModel for a model uses BasicButtonUI for the view; this is in package javax.swing.plaf.

The various listeners, ChangeListener, ActionListener, etc. can be thought of as making up the controller. Not all Swing components have dedicated controller objects.
Thus, most of the functionality of a JButton is actually located in other classes. A button is just a wrapper class that inherits from JComponent. It holds—

- a DefaultButtonModel object,
- some view data (e.g., the button label and icon), and
- a BasicButtonUI object that is responsible for the button view.

One of the strengths of the MVC framework supported by Swing is the ability to use multiple views for the same model. This is easy when they share the same model.

For example, JComboBox is a widget that allows the user to press the first key of a list item and have the list automatically scroll to the right place.

The same model can easily be used for both a JList and JComboBox because ComboBoxModel extendsListModel.

Layout

(See http://java.sun.com/docs/books/tutorial/uiswing/overview/layout.html)

In order to produce a visually pleasing user interface you must—

- Select appropriate JComponents and
- Organize them within your Window(s)

This organizing or layout task involves—

- Inserting each component into a potentially visible container (e.g., a JFrame or an JApplet).
- Setting the size of the component and its placement within the container.

Insertion of components into containers establishes a containment relation.
Since containers are themselves components, it is possible to insert containers within containers.

Thus for a given user-interface “screen,” a containment tree is typically constructed, which is rooted in the top-most container (either JApplet, JFrame or JDialog).

Each instance of Container (and thus each JComponent) is associated with a layout manager that says how the attached components should be arranged.

The default layout manager for a JFrame is the border layout manager, which allows five components to be added at positions called Center, North, South, East, and West.

Because the default layout manager for a Frame is BorderLayout, it is not necessary to specify it. But optionally one may specify, just before adding a component:

```
setLayout(new BorderLayout());
```

Whenever you iconify, deiconify, move, or expose a window, Java calls the paint method of all canvases attached to the corresponding frame.

**Placement and Coordinates**

When a Component is placed within a Container, its placement is relative to its Container.

Thus, if a JButton is added to a JFrame and then reshaped, e.g., by

```
b.setBounds(10, 50, 100, 100)
```

this will place it at x,y position (10,50) relative to the origin of the Frame.

For working with positioning and insertion; here are some key methods of java.awt.Component and java.awt.Container:

```
For java.awt.Component:
```
Container getParent() — returns a reference to this Component’s immediate “container” or parent.

Rectangle getBounds() — returns the bounding box of the component (relative to parent). This includes its origin, width, and height.

The bounding box can be modified by:

```java
void setBounds(int x, int y,
               int width, int height)
```

Dimension getSize() — gives access to just the width and height of the component. This can be modified by:

```java
void setSize(int width, int height)
```

void setLocation(Point p) — moves a component to a new location. The top-left corner of the new location is specified by point p. Point p is given in the parent’s coordinate space.

For java.awt.Container:

Component add(Component) — This has many overloaded variants. Performs the insertion of the argument Component into the target Container.

There is a corresponding remove method.

Component[] getComponents() — This retrieves handles to all of the Components this Container contains.

Example Layout

The following explicitly sizes and places two Buttons within an Applet.

```java
import java.awt.*;

public class RawLayout extends java.applet.Applet
{
   private Button b1_ = new Button("Upper-Left"),
          b2_ = new Button("Lower-Right");

   public void init()
```


```java
{  
    setLayout(null);  // Needed for no layout mgr.
    add(b1_);
    add(b2_);
    b1_.setSize(100,100);
    b2_.setBounds(size().width-150,
                   size().height-175, 150, 175);
}  
}
```

**Layout managers**

A layout manager is responsible for sizing and arranging its Component(s) (setting their origin coordinates, etc.).

Each LayoutManager has a particular “style” of arranging all of the immediate Component(s) of the corresponding Container.

Using a layout manager eliminates the need for the programmer to worry too much about coordinates.

When a container is resized, or when a component is added or removed, a protocol between the container and the layout manager redoes the layout in the appropriate fashion.

Most Layout Managers will resize components so as to “fill” the bounds of the Container.

Because of this, it is often a good idea to place lone Components within a Panel and insert the Panel into the Container. This way the LayoutManager will stretch the Panel and not the Component.

All Containers have a default LayoutManger associated with it upon creation. For most Containers this is FlowLayout, which does not resize Component(s added to it.)

There are several predefined layout managers defined in Swing:

**FlowLayout** — The simplest layout manager. It lays components out in a single row from left to right, wrapping around to additional rows if necessary.
BorderLayout — Arranges components around the border of the container. You can specify insertion to the North, East, South, West, and Center. The center area is resized to take up any excess space.

CardLayout — Stacks components on top of one another, allowing only to top component to show. It allows for flipping to a particular “card.”

GridLayout — Arranges components in a row-column grid. This enforces regular sizing of Components.

GridBagLayout — A complex, highly flexible layout manager which permits irregularly sized components which can span multiple rows and columns.

Once the container is instantiated, you typically do a `setLayout()` and then insert (add) your components.

Here is another example of how panels can be placed inside of containers to gain more control over the layout:

```java
import java.awt.*;

public class ManagedLayout extends java.applet.Applet {
    private Button b1_ = new Button("Lower-Left"),
                   b2_ = new Button("Lower-Right"),
                   b3_ = new Button("Top-Center");

    public void init() {
        // Choose Layout Manager for applet
        setLayout(new BorderLayout());
        Panel p1 = new Panel();
        p1.add(b1_); // Panel uses FlowLayout
        p1.add(b2_);
        add("South", p1); // Add panel to Applet

        Panel p2 = new Panel();
        p2.add(b3_);
        add("North", p2);
    }
}
```