# COMP 401

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

MVC and Graphics

Our understanding of toolkit widgets allows us to compose WIMP GUIs using predefined toolkit widgets. It does not, however, allow us to draw (custom) 2-D graphics in windows. Some of the predefined widgets such as JSlider and JProgressBar do draw graphics. However, the painting is done by the widget implementer. Here, we will see how to create such widgets ourselves. In this process, we will learn important lessons about window managers, obscuring and exposing of windows, and IS-A vs HAS-A. Again, we will do our implementation using MVC

# Additional CartesianPlane View

Consider the CartesianPlane user-interface we created earlier reproduced in Figure ???.



We defined the model object for this application, letting ObjectEditor do its magic to create the view and controller for the user-interface. Let us expand this application to create another user-interface to interact with the application, shown in Figure ???.



In contrast to the ObjectEditor user-interface, this user-interface does not display a text-field for the AxesLength property, paints the lines and labels in blue rather than black, and draws dashed rather than solid lines. It consists only of a view – it relies on the ObjectEditor user-interface to input the axes length. As before, we want all views to change when the axes length change in the model. We will use it to illustrate the concepts needed to do our own drawing.

Before we determine how to draw and redraw, let us consider when drawing code should be called. To get a complete answer, we need to understand something that might seem unrelated at first, but is crucial to the design of (re) drawing code.

# Exposing and Resizing a Window

Suppose we obscure our view of the Cartesian plane by putting another window in front, as shown in Figure ???. This means that some of the pixels of our view are no longer shown, which in turn means, the display Framebuffer no longer has their values.



Later, we expose the hidden drawing by, say, minimizing the BMISpreadsheet window. How do the hidden pixels get restored in the framebuffer?

The placement of widows is done by a system component called the window manager, which is different from the window system. The window manager provides the commands to move, resize, minimize, and maximize top-level windows. The window system is responsible for drawing in these windows and creating subwindows in them. It is accessed in Java through the user-interface toolkit. When the window system moves a window in from of another window, it could copy the obscured area in the framebuffer into a backing store. Later, when the obscured area is re-exposed, it could reverse the process by copying from the backing store to the framebuffer. The application doing the drawing is not involved in this process.

There are two problems with this approach. First space must be allocated by the window system for backing store, which can be huge, given the high-resolution displays we have today. More important, the approach does not work if the application updates the drawing between the time it was obscured and re-exposed. This is very likely to happen in animation user-interfaces, which change without user input. The restored pixels do not reflect the current state of the drawing, and the backing store space is wasted unnecessarily.

Therefore an alternative approach is to send the application a *paint event* when an obscured window area is re-exposed so that it can respond by redrawing.

Early window managers, especially those supported by Xerox and Sun, supported the backing store approach. Later, the concept of application handling of paint events was introduced by the X window system, which called them *expose events*. Today, many window managers support both options. This means, an application must be ready to handle these events.

What an application draws might depend on the size of the window. Therefore, a paint event is also generated when a window is resized, by the user (through the window manager) or the program.

# When Drawing Code is Called

We can now address our question of when drawing code should be called. In an MVC-based application, it should be called:

1. Initial drawing: when the initial drawing of the model state is to be made.
2. Model changes: when the draw model state changes.
3. Paint events: when the drawn area is re-sized or re-exposed after being obscured.

We will see here a technique that allows us to write a single piece of drawing code to perform all three tasks.

# Drawable Components

The next question is, what is the area into which we draw? As we saw before, a user-interface structure consists of a hierarchy of windows or widgets, with a root top-level window. A window is simply a rectangular area on the screen. A widget is some behavior – I/O processing – added to a window. Thus, a JTextField widget is a window to which we have added capabilities to display and edit text, and a JButton is a window that simulates a button. Different widget classes, then, add different behaviors to windows. As all widgets are windows, these widget classes have a common superclass in Java, called Component. This an area into which text, sliders, and other shapes can be drawn.

Thus, what we want is a component that has not been embellished with I/O behavior so that we can add our own semantics. Based on the reasoning above, we should be able to directly use an instance of Component. Let us assume for now that we can do so, though as we see below, we will have to create our own subclass of this class to respond to receive paint events.

# Drawing Shapes vs. Setting Pixels

Drawing code results, ultimately, in certain pixels being set in the framebuffer. As there are certain pixel patterns that are often set in applications, window systems provide an API that allows the programmer to simply specify the pattern, or shape, and sets the corresponding pixels. To draw a rectangle, oval, line, or String, we can call the method drawRect(), drawOval(), drawLine(), respectively. A large set of shapes can be drawn by a window system – these are the four on which we will focus.

# Graphics Context

How do we specify the attribute such as stroke (dashed, solid, ..) and color of the shapes we want to draw? For every combination of attributes, we do not want a separate call such as drawLineBlueDashed(), and drawLineGreenSolid(). We could pass these attribute values as arguments to the draw calls. However, often we want to draw different kinds of shapes with the same attributes. For instance, we want to draw all lines and rectangles with Blue. In these situations, we would have to make sure that all of these drawing calls, made perhaps by different methods and classes, use the same attribute values. A more fundamental problem is that a shape has a multitude of attributes. Requiring each drawing call to provide values for each possible attribute makes the call cumbersome.

A solution to this problem is to simulate the notion of a physical pen. Each drawing call is made using this pen. Our pen simulation comes with default attributes, which can be changed, much as a fountain pen comes with default ink that can be changed. As it is a collection of drawing attributes providing the context for a drawing call, the pen simulation is called a *graphics context*.

In Java there are two classes, provided by the java.awt package, that implement a graphics context: Graphics and Graphics2D. Graphics was the original class, and Graphics2D is a later subclass that provides additional attributes such as stroke. As a graphics context is tied to the underlying operating system, it is manufactured by the window system and not the programmer.

The following view code to draw the CartesianPlane illustrates these concepts:

BasicStroke dotted = new BasicStroke(1f, sicStroke.*CAP\_ROUND,*

 BasicStroke.*JOIN\_ROUND,* 1f, new float[] {2f}, 0f);

CartesianPlane cartesianPlane;

public ACartesianPlaneView(

 CartesianPlane aCartesianPlane) {

 cartesianPlane = aCartesianPlane;

}

public void paint(Graphics g) {

 Graphics2D g2 = (Graphics2D) g;

 g2.setStroke(dotted);

 g.setColor(Color.*BLUE);*

 *draw(g, cartesianPlane);*

}

public static void draw(Graphics g,

 CartesianPlane aCartesianPlane) {

 *draw(g, aCartesianPlane.getXLabel());*

 *draw(g, aCartesianPlane.getYLabel());*

 *draw(g, aCartesianPlane.getXAxis());*

 *draw(g, aCartesianPlane.getYAxis());*

 }

 public static void draw(Graphics g, Line aLine) {

 g.drawLine(aLine.getX(), aLine.getY(), aLine.getX() +

 aLine.getWidth(), aLine.getY() + aLine.getHeight());

 }

 public static void draw(Graphics g, Label aLabel) {

 Point location = aLabel.getLocation();

 String s = aLabel.getText();

 g.drawString(s, location.getX(), location.getY());

 }

Here, the drawing process is started in the paint() method which gets a graphics context as an argument. It sets the stroke and color of the graphics context, and then calls auxilliary methods to draw the rest of the model object, it gets as a constructor argument. The auxiliary methods are oblivious to the set attributes, and simply use the passed graphics context to convert properties of the model shapes to corresponding drawing calls.

The properties of a model label shape translate directly to arguments of the drawString() call. The drawLine() call takes as arguments the two end points of the line. Hence, the width and height of the model line shape are used to calculate the lower left corner.

Given a paint event and a graphics context, we know now how to draw on the screen. There are, however, several questions we still need to answer about drawing graphics.

# Unified But Possibly Inefficient Painting

One of these is, how to write a single piece of code that handles all three of these conditions:

1. Initial drawing: when the initial drawing of the model state is to be made.
2. Model changes: when the draw model state changes.
3. Paint events: when the drawn area is re-sized or re-exposed after being obscured.

One way to unite these conditions is to draw the entire model even when parts of it have changed. Thus, we will draw the entire model to both create the initial drawing and to react to changes in the model. Similarly, to unite the third condition, we can draw the entire component, even when parts of it are exposed. Our paint method, above, would work as long as the drawable component is cleared before it is called. This approach is convenient, and it is inefficient only when complex computations have to be performed to make the drawing. In a course on graphics, you will learn more efficient techniques. For this course, we will use this simple approach, which will be sufficiently practical in your assignments.

# Intercepting Paint Events by Overriding Paint Methods

Another issue is how we intercept paint events. Recall that these events are generated by system components and handled by our application code. Thus, we need callbacks in our code to handle these events, much as we needed callbacks to handle action events from button and text-field widgets. As we saw before, in the case of action events, the observer pattern is used in Java 1.1 and later versions However, the Java AWT/Swing toolkits use the inheritance model for intercepting paint events.

The paint() method used in the drawing code above is implemented in the system class, Component. When the window manager realizes that a Component instance needs to be repainted, it calls this method. If a subclass overrides this method, then the more specific implementation is called instead. This is illustrated by showing the outline of our graphics view class:

public class ACartesianPlaneView extends Component

 implements PropertyChangeListener {

 …

 public void paint(Graphics g) {

 …

 }

}

Like other view objects, it is a listener of model changes. It is also a subclass of Component that overrides the superclass paint() method to intercept paint events.

Unfortunately, now our view inherits from Component, even though logically it is not a Component (window) – it has a Component. In fact, we might want a view to manage the drawing of multiple Component instances – something that is precluded by this inheritance-based callback approach. A later Java toolkit, SWT, which is used to implement Eclipse, does not have this limitation, and uses the observer pattern for also intercepting paint events.

# Repaint

Now consider the task of making the initial drawing and updating it. Both require a graphics context. Recall that this object is created by the window system and not manufactured by the application. In the case of paint events, the system can pass this object to the paint method. But when the drawing has to be initiated by the application, it needs a way to pass the graphics context to the paint() method, to which it does not have direct access.

Java used indirection, a solution to many computer science problems, to make this happen. Whenever a window has to be redrawn, whether in response to a paint event or a model state change, the parameterless repaint() method is called in Component. This system method clears the contents of the Component and calls the paint() method, passing it the graphics context, to which it has access. The object passed is an instance of Graphics2D. Therefore it is safe to cast it to this class. We could have also declared the paint method argument as Graphics2D instead of Graphics, which would have required no casting.

The following notification method in our graphics view illustrates this call:

 public void propertyChange(PropertyChangeEvent evt) {

 repaint(); // causes paint to be called

 }

Recall that this method is called by the model both when the view is initially registered and when the model is updated. This, this single repaint() call handles painting in both cases.

In a graphics view, it makes no sense to examine the notification event to determine the changed property, as the entire model is redrawn. Other implementations of this method can use the event to determine the model that changed – they need not store the models in instance variables. In a graphics view, repainting can also be done in response to paint events sent by the window manager, which have no knowledge of the model of view. Thus, a graphics view must store references to all of its models in its instance variables. This is why our graphics view has the following constructor:

 public ACartesianPlaneView(

 CartesianPlane aCartesianPlane) {

 cartesianPlane = aCartesianPlane;

 }

This time our composer must pass the model to the view, rather than relying on the view to learn about the model from notification methods.

# Listening to Notifications from a Composite Model

This is the first view we have seen for a composite model, that is, a model that itself has object properties. In general, such a view must listen to not only changes to properties of the model, but also the properties of its object descendants. This is illustrated in the composer code below:

public class CartesianPlaneComposer {

 public static void main(String[] args) {

 CartesianPlane cartesianPlane =

 new AnObservableCartesianPlane(100, 100, 100);

 PropertyChangeListener view = new

 ACartesianPlaneView(CartesianPlane);

 cartesianPlane.addPropertyChangeListener(view);

 ((ObservableLine) cartesianPlane.getXAxis()).

 addPropertyChangeListener(view);

 ((ObservableLine)

 cartesianPlane.getYAxis()).addPropertyChangeListener(view);

 ((ObservableStringShape)

 cartesianPlane.getXLabel()).addPropertyChangeListener(view);

 ((ObservableStringShape)

 cartesianPlane.getYLabel()).addPropertyChangeListener(view);

 JFrame frame = new JFrame(" Cartesian Plane");

 frame.add((Component) view);

 frame.setSize(300, 300);

 frame.setVisible(true);

 ObjectEditor.*edit(CartesianPlane);*

 }

}

The view is registered as a listener of not only the CartesianPlane, but also the two axes and labels in it. In this problem, this is not strictly necessary, as the only independent property is AxesLength in the CartesianPlane instance. Our view redraws the entire model tree whenever any part of the model changes. Thus, when AxesLength changes, it will draw changes to all the dependent properties in the component objects. In your assignments, however, you will have many independent properties in the model tree. A graphics view must register itself as a listener of each object in the tree that has a component that can change independently.

# Dynamic Dispatch

Consider again what we know about the Component class. It implements both repaint() and paint(), with the repaint() calling paint(). Yet, if we subclass Component and override paint(), then the repaint() call on an instance of the subclass calls the overridden method and not the one implemented in the class of the called repaint(). This call semantics is called dynamic dispatch – when a method is called on an object, the implementation of the method in the most specific class of the object is called. Some of you might be confused by these semantics, but you will see they give you necessary flexibility. We will study this concept further when we look at abstract methods.

# Plotted Shuttle

To further understand interaction with Component objects, let us consider creating another graphics user-interface, this time of the PlottedShuttle type we saw earlier. Recall that ObjectEditor created the following user-interface for it:



The use-interface we create is an extension of the graphics view we created above that draws a shuttle image in the view. Later, we will embellish this user-interface to add mouse and keyboard input.

# Subclassing Views and Drawing Images

Naturally, we want to reuse the code we wrote for the Cartesian plane graphics view. Java requires us to use inheritance for this reuse. So we will create a subclass of the previous graphics and override the paint method to draw the additional image. The following view code shows how this is done:

public class APlottedShuttleView

 extends ACartesianPlaneView implements BeanView {

ObservablePlottedShuttle plottedShuttle;

 public APlottedShuttleView (

 ObservablePlottedShuttle aPlottedShuttle) {

 super((ObservableCartesianPlane) aPlottedShuttle.getCartesianPlane());

 plottedShuttle = aPlottedShuttle;

}

public void paint(Graphics g) {

 super.paint(g);

 Graphics2D g2 = (Graphics2D) g;

 draw(g2, plottedShuttle.getShuttleImage());

 }

 public void draw(Graphics2D g, ShuttleImage anImage) {

 Image img =

 Toolkit.*getDefaultToolkit().getImage(anImage.getImageFileName());*

 g.drawImage(img, anImage.getX(), anImage.getY(), this);

 }

 public void draw(Graphics2D g, PlottedShuttle aShuttleLocation) {

 draw(g, aShuttleLocation.getCartesianPlane());

 draw (g, aShuttleLocation.getShuttleImage());

 }

}

The constructor passes the Cartesian plane property of the plotted shuttle to the super class. The paint method first asks the superclass to draw this property, and then draws the shuttle image. (What if we reverse the steps?) Thus, this class and its superclass draw different properties of the model in the same window.

The drawImage() called provided by Graphics takes as an argument an object of type Image. Our model image object identifies the image by the name of the file containing it. The static getImage() method of Toolkit provides a way to translate an image-file name to the corresponding Image instance.

# Mouse and Keyboard Input

What we have learned so far allows us to add output behavior to a widget. Like the predefined widgets provided to us, we might also want our widget to respond to input in the drawn window. At the widow level, there are two kinds of input that apply to all widgets – keyboard and mouse input. A widget receives these events from the window system and transforms them to widget-specific input such as a button presses. To add widget-specific input behavior to a window, then, the widget should be able to intercept mouse and key input.

Fortunately, Java used the familiar observer pattern to notify our application code. A Component instance is an observable providing two observer registration methods:

public void addMouseListener(MouseListener aListener);

public void addKeyListener(KeyListener aListener);

This, it is the first observable we have seen that implements two kinds of registration methods, to allow observers to listen to relevant subsets of all possible notifications sent by the observable.

Suppose we want to embellish the plotted shuttle user-interface with the capability to position the shuttle at the origin (of the Cartesian plane in which it is displayed) whenever a mouse click occurs in the window. We can define the following controller for it:

public class APlottedShuttleMouseController implements MouseListener {

 Component shuttleComponent;

 PlottedShuttle plottedShuttle;

 public APlottedShuttleMouseController (PlottedShuttle aPlottedShuttle,

 Component aShuttleComponent) {

 shuttleComponent = aShuttleComponent;

 plottedShuttle = aPlottedShuttle;

 shuttleComponent.addMouseListener(this);

 }

 public void mouseClicked(MouseEvent e) {

 plottedShuttle.setShuttleX(0);

 plottedShuttle.setShuttleY(0);

 }

 public void mouseEntered(MouseEvent e) {}

 public void mouseExited(MouseEvent e) {}

 public void mousePressed(MouseEvent e) {}

 public void mouseReleased(MouseEvent e) {}

}

The class shows the methods that must be implemented by a MouserListener to listen to different kinds of mouse notifications. It is only interested in the mouseClicked() notifications – so all other methods do nothing. The mouseClicked() method ignores the MouseEvent describing the position of the mouse, and simply sets the mouse location to the origin . Both the component in which the mouse is clicked and the model are passed to this controller in the constructor. The constructor registers the instance as a mouse listener of the component.

Suppose we further embellish the user-interface to change the X (Y) coordinate of the shuttle to a predefined value when the user enters the ‘x’ (y) character. The process is very similar to the one we saw above except that our new controller listens to key rather than mouse events. In general, multiple top-level windows can be displayed on the screen, but only one of them actually receives key input. This is the current focus window. A component of a focus window can notify its listeners of key events only if the setFocusable(**true**) call is made on it. Our key controller makes this call:

public class APlottedShuttleKeyController implements KeyListener {

 Component shuttleComponent;

 PlottedShuttle plottedShuttle;

 public static final int *TARGET\_X = 100;*

 public static final int *TARGET\_Y = 100;*

 public APlottedShuttleKeyController (PlottedShuttle aPlottedShuttle,

 Component aShuttleComponent) {

 shuttleComponent = aShuttleComponent;

 plottedShuttle = aPlottedShuttle;

 shuttleComponent.addKeyListener(this);

 shuttleComponent.setFocusable(true);

 }

}

public void keyTyped(KeyEvent e) {

 char typedChar = e.getKeyChar();

 switch (typedChar) {

 case 'x' :

 plottedShuttle.setShuttleX(*TARGET\_X);*

 break;

 case 'y':

 plottedShuttle.setShuttleY(*TARGET\_Y);*

 break;

 }

 }

 public void keyPressed(KeyEvent e) {}

 public void keyReleased(KeyEvent e) {}

}

Again, here are multiple kinds of key event notifications, but our controller processes only the keyTyped notification. This time, it processes the event information, to determine which key has been pressed.

As this example shows, it is possible to create multiple controllers for the same Component. These two controllers are completely independent of each other – so the separation of concerns principle tells us that they should indeed have been separate classes.

# MVC Composition Before UI Composition

Our composer now associates both controllers and the view we saw earlier with the model, besides creating the simple UI structure:

public class PlottedShuttleComposer {

 public static void main(String[] args) {

 ObservablePlottedShuttle plottedShuttle = new

 AnObservablePlottedShuttle(50, 100);

 PropertyChangeListener view = new APlottedShuttleView(plottedShuttle);

 plottedShuttle.addPropertyChangeListener(view);

 MouseListener mouseController =

 new APlottedShuttleMouseController(plottedShuttle, (Component) view);

 KeyListener keyController =

 new APlottedShuttleKeyController(plottedShuttle, (Component) view);

 JFrame frame = new JFrame("Plotted Shuttle");

 frame.add((Component) view);

 frame.setSize(300, 300);

 frame.setVisible(true);

 ObjectEditor.*edit(plottedShuttle);*

 plottedShuttle.setShuttleY(100);

 plottedShuttle.setShuttleX(50);

 }

}

It is strange to add a view to a frame, but our view extends component, so this is legal.This time the code composing the MVC structure appears before the code creating the display structure because the view is part of the display structure.

# Summary of Window-based MVC

As we have seen, the MVC pattern is tied to a set of I/O methods provided by some application-independent user-interface library. In this chapter, the I/O library was not the set of predefined widgets, but the unembellished window object, which in Java is an instance of Component.

Our controllers listen now to raw key and mouse events rather than higher-level events such as text field entries and button presses. But they still use the observer pattern to listen for these events.

A view must now listen also for paint events. In AWT and Swing, it uses inheritance to do so. It redraws the entire window when any part of it changes. As a result, its model notification method does not try to determine which model component has changed. However, it must register itself an observer of every object in a model structure that has a component that can be independently changed.

 View output code is executed not only when the model changes but also when the associated top-level window is exposed. As a result, a view must contain references to the model.

The drawing code is executed in the Component paint() method, which takes a graphics context as an argument. This argument is supplied by the Compoent repaint() method. This method is called by the system when the window is re-exposed or resized. It is also called by the mode notification method to initiate a redrawing. Dynamic dispatch ensures that the most specific implementation of repaint() is called.

# Exercises

1. What advantages does the interactor pattern approach offer over the monolithic main approach?
2. What advantages does the MVC framework offer over the interactor approach? Do you see any disadvantages?
3. Distinguish between models and listenables, and views and listeners.
4. In the MVC pattern, describe the roles of the controller(s), model, and view(s). Classify all of the classes you created for Assignment 9 as a controller, a model, a view, or none of the above.
5. Compare and contrast the three different notifications schemes you learned about. The three schemes are embodied in Java’s java.util.Observer implemention, AWT components, and java.beans package (PropertyChangeListener and PropertyChangeEvent).
1. © Copyright Prasun Dewan, 2015. [↑](#footnote-ref-1)