Chapter 9

Brahma Dathan Sarnath Ramnath

November 8, 2018

Contents

Chapter 9

Interactive Systems and the MVC Architecture

9.1 Introduction

So far we have seen examples and case studies involving relatively simple software systems. This simplicity enabled us to use a fairly general step-by-step approach, viz., specify the requirements, model the behavior, find the classes, assign responsibilities, capture class interactions, and so on. In larger systems, such an approach may not lead to an efficient design, and it would be wise to rely on the experience of software designers who have worked on the problem and devised strategies to tackle the problem. This is somewhat akin to planning our strategy for a game of chess. A chess game has three stages - an opening, a middle game and an endgame. While we are opening, the field is undisturbed and there is an immense number of possibilities; toward the end, there are fewer pieces and fewer options. If we are in an endgame situation, we can solve the problem using a fairly direct approach using first principles; to decide how to open is a much more complicated operation and requires knowledge of "standard openings." These standard openings have been developed and have evolved along with the game, and provide a framework for the player. Likewise, when we have a complex problem, we need a framework or structure within which to operate. For the problem of creating software systems, such a structure is provided by choosing a **software architecture** (sometimes referred to as an **architectural pattern**) .

In this chapter, we start by describing a well-known software architecture called the **Model-View-Controller** or **MVC** pattern. Next, we design a small interactive system using such an architecture, look at some problems that arise in this context and explore solutions for these problems using design patterns. Finally, we discuss pattern-based solutions in software development and some other frequently employed architectural patterns.

9.2 The Model View Controller Architectural Pattern

The Model View Controller is a relatively old pattern that dates back to the early days of the Smalltalk programming language. As one might suspect from its name, the pattern divides the application into three subsystems: Model, View, and Controller. The pattern separates the application object or the data, which is termed the model, from the manner in which it is rendered to the end-user (view) and from the way in which the end-user manipulates it (controller). In contrast to a system where all of these three functionalities are lumped together (and thus has a low degree of cohesion), the MVC pattern helps produce highly cohesive modules with a low degree of coupling. This facilitates greater flexibility and re-use. MVC also provides a powerful way to organize systems that support multiple presentations of the same information.

The structure is shown in Figure 9.1. The model stores the data and any object can play the role of model. To a great extent, the model is unaware of the existence of either the view or the controller and does not explicitly participate in the proceedings. The view displays the data stored in the application in a format suitable for the end user. Moreover, the data may be displayed at an appropriate level of abstraction. For instance, if the model stores information about bank accounts, a certain view may choose to display only the number of accounts and the total of the account balances. The controller captures the user inputs. In contrast to its relationship with the model, a view is fairly closely connected to a controller and vice-versa.

In a typical application, the model changes only when user input causes the controller to inform the model of the changes. The view must be notified when the model changes. Instance variables in the controller refer to the model and the view. Moreover, the view must communicate with the model, so it has an instance variable that points to the model object.

The MVC model provides a broad division of responsibilities, but the design and implementation phases require a mapping of these three subsystems to physical classes. Usually, a GUI functions as the vehicle employed by the view to display the data in the model. There is a bit of confusion with the mapping because the GUI not just displays the information stored in the model, but also provides controls to allow the user to manipulate the data stored in the model. The code to display the GUI usually involves an intricate arrangement of a host of classes and interfaces and is highly technology dependent. The framework used for GUI design and development changes over time, and for the sake of modifiability, it would be preferable to isolate the code related to the GUI itself (based on JavaFX in this book) from the modules that deal with commands and data at a more "logical" level. As an example, a command to create a rectangle shape in a drawing program may be physically issued through the push of a JavaFX `Button` object and must necessarily be received via JavaFX code, but the code needed to create a rectangle object is independent of the GUI technology and is more stable. Such a division of responsibilities permits modifiability; for example, if and when a new technology replaces JavaFX, it would be easier to replace just the JavaFX modules. Once we adopt this design approach, we can see that at the software level, we have four subsystems: the model, view, controller, and GUI. The GUI, as we just mentioned, contains the essential code for interacting with the user. All user input would be received by the GUI and sent to the controller. The data from the model to be shown to the user goes through the view to the GUI. (The data may be condensed, analyzed, reformatted, etc. within the view, so displaying it is not necessarily a trivial process.)

User-generated events may cause a controller to change a model, or view, or both. For example, suppose that the model stored the text that is being edited by the end-user. When the user deletes or adds text, the controller captures the changes and notifies the model. The view, which observes the model, then refreshes its display, with the result that the end-user sees the changes he/she made to the data. In this case, user-input caused a change to both the model and the view.

On the other hand, the controller may also send messages to the user via the GUI. Con-

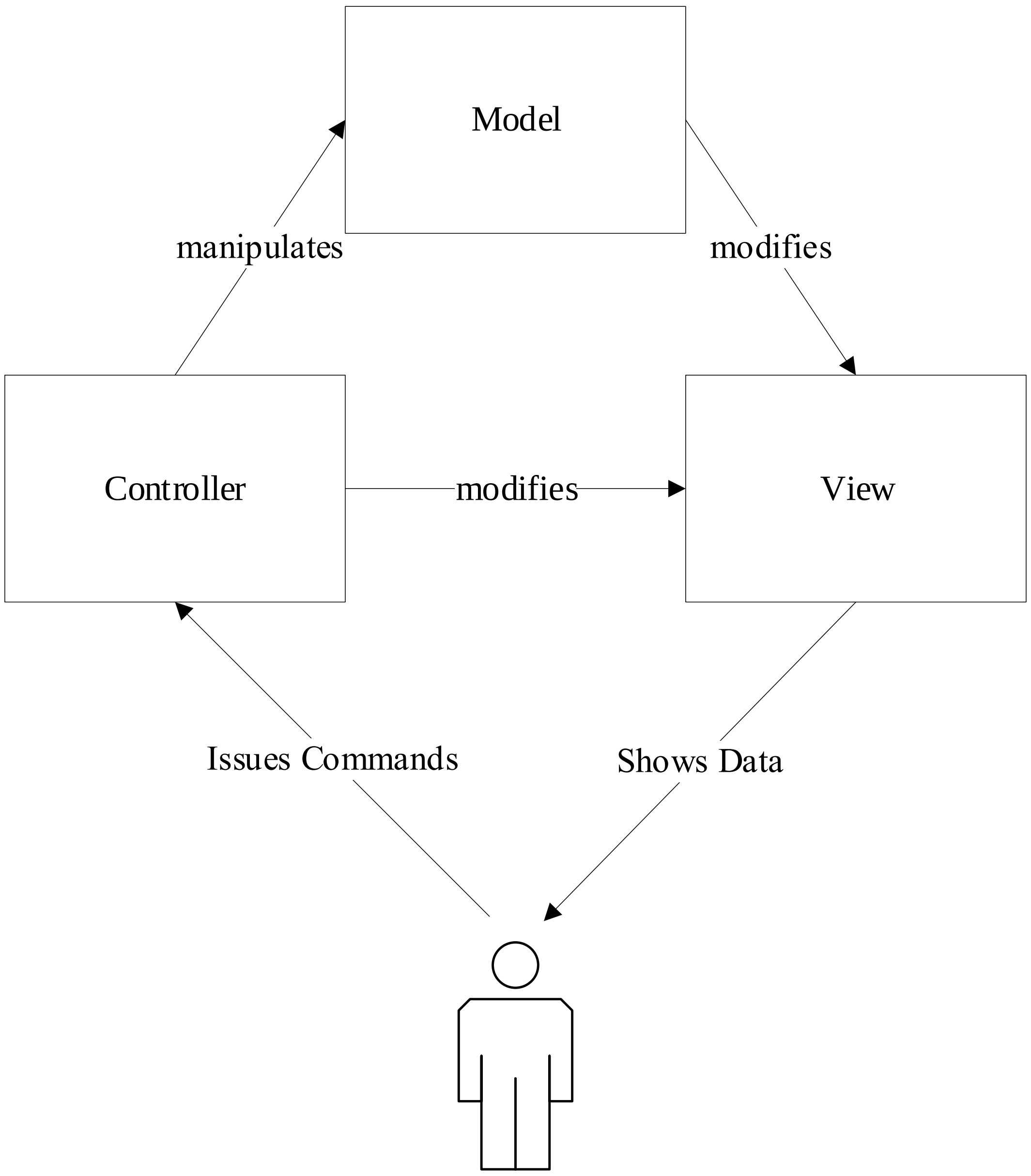

Figure 9.1: The Model-View-Controller Architecture. The model stores the data related to some application. The data is displayed to the user by the view. The user issues commands to manipulate the data through the controller. In practice, the controller may issue commands to the view to modify the display, without an actual modification of the data stored in the model.

sider, for instance, a user scrolling the data. Since no changes are made to the data itself, the model does not change and need not be notified. But the view now needs to display previously-hidden data, which makes it necessary for the view to contact the model and retrieve information.

More than one view-controller pair may be associated with a model. Whenever user input causes one of the controllers to notify changes to the model, all associated views are automatically updated.

It could also be the case that the model is changed not via one of the controllers, but through some other mechanism. In this case, the model must notify all associated views of the changes.

9.2.1 Examples

A library system could have a view that shows information about a book: the title, id, number of copies, and whether a copy is available for checkout. This view could be connected to two GUI screens, which are connected to two different controllers, which manipulate the book data in different ways. Suppose the first GUI screen allows book checkouts. The second GUI screen permits a library staff member to add copies of books. Suppose a customer uses the first GUI screen to see whether a certain book is available for checkout. Let us assume that at this time all copies are checked out. In parallel, assume a library staff member views the same book information and adds a copy using the second GUI screen. When the new copy is added, the model is updated, the view is notified resulting in an update of the first GUI screen, which informs the customer that a copy is now available for checkout.

A second example is that of a mail sever. A user logs into the server and looks at the messages in the mailbox. In a second window, the user logs in again to the same mail server and composes a message. The two screens form two separate views of the same model.

Suppose that we have a graph-plot of pairs of (x, y) values. The collection of data points constitutes the model. The graph-viewing software provides the user with several output formats - bar graphs, line graphs, pie charts etc. When the user changes formats, the view changes without any change to the model.

9.2.2 Implementation

The view and model interaction is modeled using the Observer pattern. The model, as the observable, maintains references to all of the views (the observers) that are interested in observing it. Whenever an action that changes the model occurs, the model automatically notifies all of these views. The views then refresh their displays.

The definition for the model will be as follows:

```
public class Model {
  // code
  private PropertyChangeSupport propertyChangeSupport =
          new PropertyChangeSupport(this);
  public void changeData() {
    // code to update data and construct a PropertyChangeEvent, event
    this.propertyChangeSupport.firePropertyChange(event);
  }
}
```

Each of the views is observer and implements the `propertyChange()` method.

```
public class View implements PropertyChangeListener {
  // code
  public void propertyChange(PropertyChangeEvent event) {
    // process the event
  }
}
```

If a view is no longer interested in the model, it can be deleted from the list of observers.

Since the controllers react to user input, they may send messages directly to the views asking them to refresh their displays.

9.2.3 Benefits

The MVC paradigm provides a natural division of responsibilities and affords the following advantages.

- *Cohesion of Modules*: Instead of putting unrelated code (display and data) in the same module, we separate the functionality, so that each module is cohesive.
- *Flexibility*: The model is unaware of the exact nature of the view-controller pair(s) it is working with. It is simply an observable. This adds flexibility.
- *Lower coupling.* Modularity of the design improves the chances that components can be swapped in and out as the user or programmer desires. This also promotes parallel development, easier debugging, and maintenance.
- *Adaptability to Changes*: Components can be changed with less interference to the rest of the system.
- *Applicability to Distributed Systems*: Since the modules are separated, it is possible to have the view-controller pair in a system that is geographically separated from the model.

9.3 Creating a Simple Drawing Program

We now apply the MVC architectural pattern to the process of designing a simple program that allows us to create and label figures. The purpose behind this exercise is twofold:

- *To demonstrate how to design with an architecture in mind.* Designing with an architecture in mind requires that we start with a high-level decomposition of responsibilities across the sub-systems. The sub-systems are specified by the architecture, which dictates the broad division of classes.
- *To understand how the MVC architecture is employed.* We will follow the architecture somewhat *strictly*, i.e., we will try to have three clearly delineated sub-systems for Model, View, and Controller. Later on, we will explore and discuss variations on this theme.

As always, our design begins with the process of collecting requirements.

9.3.1 Specifying the requirements

Our initial "wish-list" calls for software that can do the following.

1. Draw lines, rectangles, polygons, and circles.
2. Place labels at various points on the figure; the labels are strings.
3. Save the completed figure in a file. We can open a file containing a figure and edit it.
4. Select the shapes to move or delete them.
5. Undo recent operations of the drawing process. If needed, the operations can be redone as well.

Compared to the kinds of drawing programs we have on the market, this looks too trivial. Nonetheless, the functionality is rich enough to show how the responsibilities can be divided so that the MVC pattern can be applied.

In order to attain this functionality, the software will interact with the user. We need to specify exactly how this interaction will take place. It should, of course, be user-friendly, fast, etc., but as in earlier examples, these non-functional requirements will not be the focus of our attention. Without more ado, let us adopt the following "look and feel:"

- The software will have a simple frame with a display panel on which the figure will be displayed, and a command panel containing the buttons. There will be buttons for each operation, like Line, Circle, Label, etc. The system will listen to mouse-clicks which will be employed by the user to specify points on the display panel.
- The user can abandon the current command by pressing an appropriate key. Let us assume we use the Escape key for this purpose.
- When drawing is in progress, the shape of the cursor when the mouse is on the command panel (called the default cursor shape), is different from the shape when the mouse moves to the display panel (called the drawing cursor shape).
- As a label is entered, a text cursor will be shown to indicate the point at which the next character will be entered.
- The system will support a view-controller pair that allows creation of shapes, their displays, and associated support operations such as undo, redo, and data save and retrieval.
- The program will allow the user to instantiate additional views that display the shapes, but not allow modification. We refer to these views, read-only views.

The look of the view-controller pair of drawing program is shown in Figure 9.2. [1] Since JavaFX is the current technology of choice for building GUI programs, we will use this to implement this part of the system. To illustrate important design concepts, the read-only views will be implemented using both FX and Swing (the GUI technology preceding FX).

[1]The mouse is duplicated to show its shape in the two panels. The shapes are supported in JavaFX. The shapes could be different in a different technology.

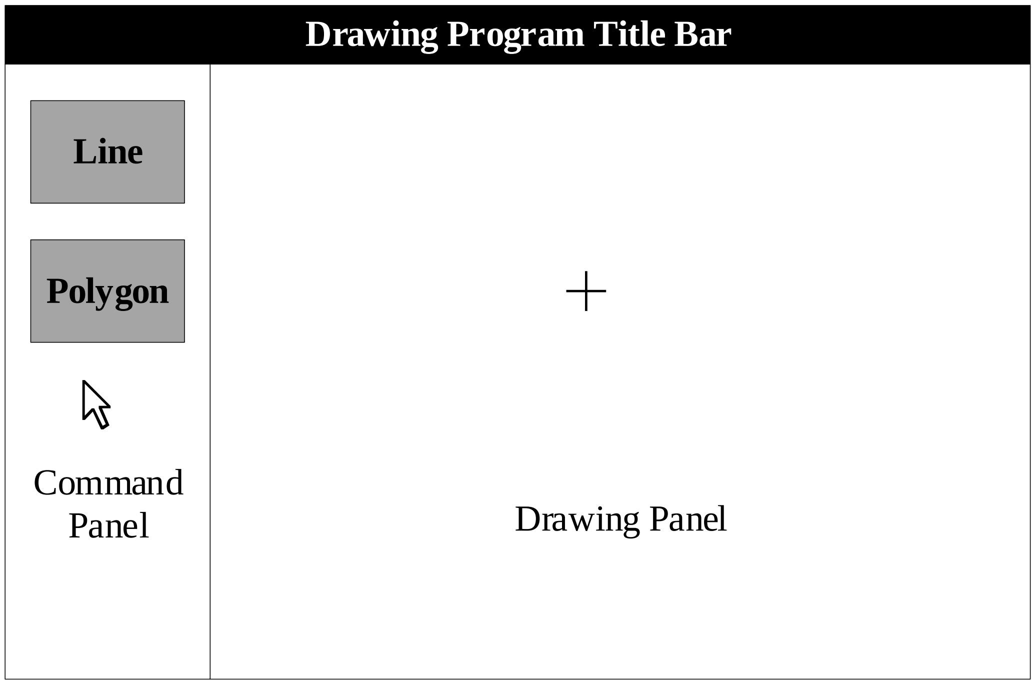

Figure 9.2: The basic layout of the drawing program. Note that besides the customary title bar, the screen is divided into two panels: one for the command buttons (such as Line and Polygon) called the command panel and one for the drawing itself, which we term the drawing panel. The drawing panel has a cross-hair cursor (drawing cursor shape) and the command panel has an arrow cursor (default cursor shape). Obviously, the two cursors cannot coexist, but we draw both to show the reader how the cursors would look in the two panels.

Actions performed by the actor	Responses from the system
1. The user clicks on the Line button in the command panel and moves the cursor to the drawing panel.	
	2. The cursor takes the drawing cursor shape.
3. The user clicks first on one end point and then on the other end point of the line to be drawn.	
	4. The system adds a line segment with the two specified end points and displays the line on the screen.

Table 9.1: Use-case for drawing a line. Note that the user may move the mouse any number of times between the drawing and command panels and the cursor will change the shape accordingly. In Step 2, though, we explicitly write the change in shape of cursor to a cross-hair, in order to emphasize the response of the system to the draw line command.

9.4 First Version of the Drawing Program

In our first version, we have the functionality for drawing lines, labels, and polygons. We will first write the use cases for drawing each of these shapes, then design the system, and finally show the implementation. We will also have the functionality for saving and opening drawings, but we will not spend much time on its design or implementation as the concepts are based on serialization, which we have seen in earlier chapters and does not require any new concepts.

9.4.1 Analysis

We now write the use cases for drawing a line, polygon, and creating labels. In all of the use cases, we assume the following.

- When a command is in progress, if and when the mouse is moved to the drawing panel, the cursor is changed from the default to the drawing cursor shape. With the current technology, the default cursor shape is the arrow and the drawing cursor shape is the cross-hair. These may change as the GUI technology changes.
- If the mouse is moved to the command panel, the cursor takes the default cursor shape.
- Pressing the Escape key abandons the construction of the current cursor shape.

We can now write the detailed use-cases for each operation. The first one, for drawing a line, is shown in Table 9.1.

The use-case for drawing a polygon would naturally begin with a click on a button termed Polygon. The user would click on multiple points to specify the successive vertices of the polygon. As successive vertices are specified, the program would continue to draw the successive edges. We, however, need a mechanism to specify the "last"vertex that would be

Actions performed by the actor	Responses from the system
1. The user clicks on the Polygon button in the command panel and moves the cursor to the drawing panel.	
	2. The cursor takes the drawing cursor shape.
3. The user clicks on a drawing panel point that would be the starting vertex of the polygon to be drawn.	
4. The user then clicks on another point to specify another vertex.	
	5. The system adds a line segment with the last two specified points and displays the line on the screen.
6. If there are no more vertices left in the polygon, the user presses the Enter key. Otherwise, the user goes to Step 4.	
	7. The system adds a line segment between the first and last vertex. The whole polygon is now visible on the screen.

Table 9.2: Use-case for drawing a polygon. We need to have a special way of indicating the N^{th} vertex when drawing an N-vertex polygon. This is specified by a double click of the mouse.

connected to the first vertex, to close the polygon. We assume that the user would indicate this by pressing the Enter key. The use-case is shown in Table 9.2.

To provide an interesting variation, we allow for multiple labels to be added with the same command. To start the process of adding labels, the user clicks on the Label button. This is followed by a mouse-click on the drawing panel, following which the user types in the desired label. After typing in a label, a user can either click on another point to create another label, or press the Enter key, which ends the command. These details are spelled out in the use-case in Table 9.3.

The above use-case does not specify what would happen if a user typed in a non-printable character such as a control character, or if the sequence of characters was empty. The resolution of this left to the way in which the system is implemented. The reader would note that this system is very restrictive in many ways. This has been done for simplicity and will not in any way detract from the design experience. In fact, it will highlight the extendability of the design when we extend the functionality with very little disturbance to the existing code.

In our initial implementation, we will only deal with three drawing commands: draw line, draw polygon, and create labels. We also allow the user to save and open drawings. (The use cases for saving and retrieving drawings are quite straightforward, and the reader is invited to write them.)

Actions performed by the actor	Responses from the system
1. The user clicks on the Label button in the command panel and moves the mouse to the drawing panel.	
	2. The cursor takes the drawing cursor shape.
3. The user clicks at the left end point of the intended label.	
	4. The system places a vertical bar at the clicked location.
5. If there is one more character in the label, the user types that character; if the last character (if there is one) is to be deleted, the user presses the backspace key. If the label is finished, but there is at least one more label to be entered, the user clicks the mouse at the location where the new label has to start. If the label is finished and there are no more labels to enter, the user presses the Enter key.	
	6. If a character is received, the system displays the character at the next position in the label. If the backspace key is pressed, any immediately preceding character in the label is erased. In both cases, the system displays a text cursor (a vertical bar) at the next position where the next character will be appended. If the Enter key is pressed or the mouse is clicked, the system removes the vertical bar from the end of the character sequence; in case of a mouse-click, it goes to Step 4; otherwise it ends the command.

Table 9.3: Use-case table for adding labels. The system allows creating a sequence of labels with this command. After each label, the user clicks at the start point of the next label. After finishing the last label, the user presses the Enter key.

9.4.2 Design

In this section, we design an initial solution to the drawing program, but before we can get into the design details, we first need to see how the three subsystems,the model, view, and controller interact.

The Interaction between Model, View, and Controller

We study the broad division of responsibilities among the model, view, and controller subsystems through an example, by exploring the overall sequence of actions that occur when the Draw Line button is clicked. The sequence diagram is shown in Figure 9.3. Note that many details that would only impede a fundamental understanding are omitted in this diagram. When the user clicks the button to draw a line, the command is received by the controller, which creates a `Line` object and adds it to the model. The view is an observer of the model, so it is notified of the update to the model's collection. The view redraws the drawing, although no real change occurs in the interface itself. The user then clicks the first end point, which is also received by the controller, which adds to the `Line` object and notifies the view that a change has occurred. The view once again updates the interface. Subsequently, when the second end point is clicked, the controller receives that end point as well, updates the `Line` object, and notifies the view once more. At this time, the drawing of the line is complete, and the new line is visible on the user interface.

What the reader should observe in the diagram are the following.

- The controller receives all input.
- The controller creates new drawing objects and adds them to the collection.
- When the model changes, it notifies the view.
- On occasions, the controller notifies the view without going through the model.

Defining the Model

The model keeps a collection of lines, polygons, and label objects. The details of how each of these shapes are represented are unimportant at this stage. The collection is accessed by the view from the model when the drawing is to be rendered on the screen. The model also provides mechanisms to access and modify its collection objects.

The fact that the model is a collection of shapes, implies the following members

- A collection field to store the shapes.
- Methods to add and delete shapes.

The model should also be able to communicate with one or more views. This means it should be implemented as an observable of the Observer pattern with methods to add and remove views and update them when needed.

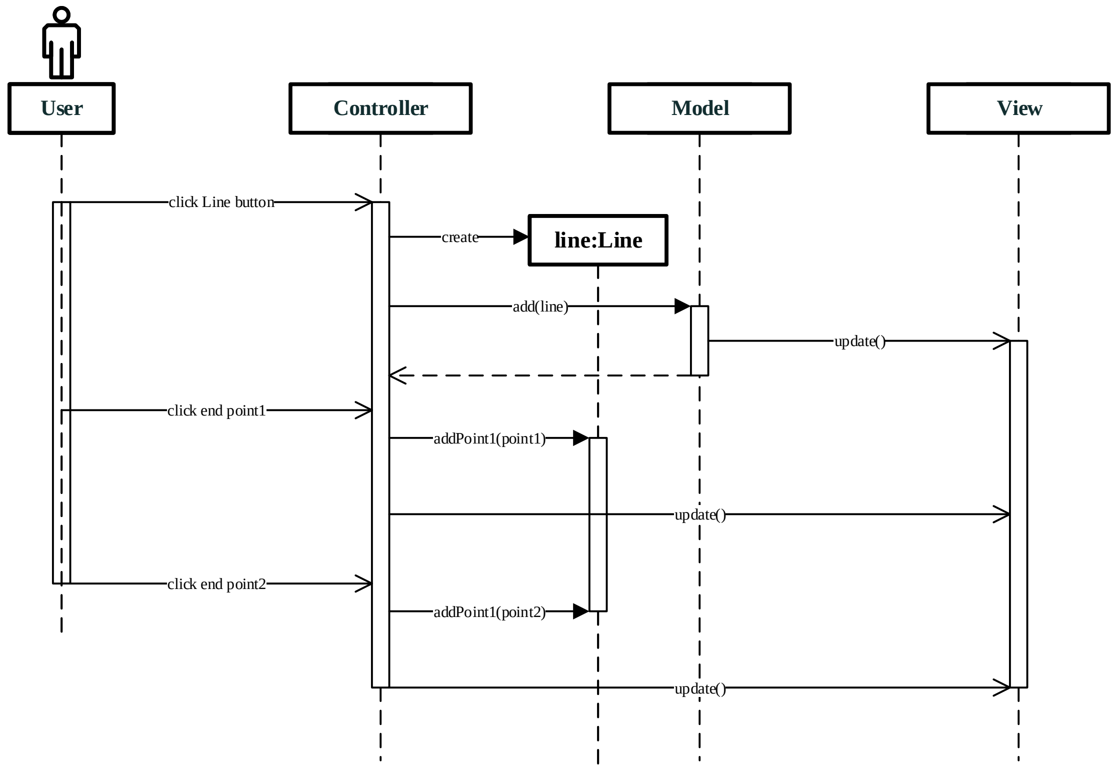

Figure 9.3: The Interactions between the Model, View, and Controller Subsystems. The diagram shows the broad division of responsibilities and interactions. Observe that: the controller receives all input and creates new drawing objects and adds them to the collection; when the model changes, it notifies the view; on occasions, the controller notifies the view without going through the model.

The Events
Line button pressed
Mouse enters the drawing panel.
Moves leaves the drawing panel.
Mouse click
Polygon button pressed
Label button pressed
Character typed
Backspace key pressed
Escape key pressed
Enter key pressed

Table 9.4: List of Events. We have removed the context from the events. The controller should be the one to decide what the context is and determine what actions should be taken for the input. For example, we simply say that clicking the mouse in the drawing panel is an action; it could be for one of several different purposes: specifying the first end point of a line, the leftmost position of a label, and so on.

Defining the Controller

An issue that immediately comes up when thinking about the design and implementation of the view and the controller is the separation of the technology for their implementation and the logic associated with responding to the inputs and the drawing of the shapes. To understand the issues, consider the use case for drawing a line, which we list below, ignoring the columns. We highlight phrases that correspond to an action by the user on the GUI or by the system.

1. The user **clicks on the Line button** in the command panel and **moves the cursor to the drawing panel**.
2. The cursor takes the drawing cursor shape.
3. The user **clicks first on one end point and then on the other end point of the line** to be drawn.
4. The system adds a line segment with the two specified end points and displays the line on the screen.

Let us consider the first highlighted action: **clicks on the Line button** in the command panel. This action is initially handled by some object to be specified by the GUI code. As we just illustrated, we can get a list of all user inputs, by examining the use cases. The user actions from the three use cases and the three general statements we listed at the beginning of Section 9.4.1 can be viewed as events as in the microwave example in the last chapter, and are shown in Table 9.4. We have removed the context from the actions. The controller should be the one to decide what the context is and determine what actions should be taken for the input. For example, we simply say that clicking the mouse in the drawing panel is an event; it could be for one of several different purposes: specifying the first end point of a line, the leftmost position of a label, and so on.

The controller is responsible for handling these events. Let us see how the controller could be structured.

Option 1 We could have a single class that responds to all events, with one method per event.

Option 2 Construct a separate class for drawing each of the shapes. When the program starts, there is no drawing; also when the user completes the drawing of a shape, the program is idle and waits for the drawing of a shape or some other event. The state when no drawing is in progress is a quiescent state and will be called `QuiescentState`. We could have a separate class to represent the drawing of each of the shapes. Thus, there will be states such as `DawLineState`, `DrawPolygonState`, and so on.

Option 3 Extend Option 2 by dividing up the drawing of a specific shape into multiple states. For example, consider the drawing of a line. After the user clicks the Draw Line button, the program goes into a state in which more input is expected to specify the line. First it expecting the first end point and this is the state to which the quiescent state will transition (when the Draw Line button is clicked.) We term this state `LineFirstInputState` to mean the state in which the first input is expected. After the first endpoint of the line is received, the system transitions to a state in which the other endpoint is expected. This state will be called `LineSecondInputState`.

What are the relative merits of these approaches? Option 1 requires just one class, but we can see that an embellishment of the system so support more events and shapes would increase the code size. Moreover, the code would potentially have complicated nested selection structures, complicating implementation and maintenance. Option 2 offers a clear separation of concerns. Each state will have some conditionals to correctly handle the events. For example, the method to handle mouse clicks in `DawLineState` needs to distinguish between the first and second mouse clicks. Option 3 has the advantage of avoiding these conditionals, at the cost of requiring two or more states for the drawing of most shapes.

Option 2 is a reasonable compromise among the options discussed above. Thus the drawing process can be outlined as below. At any given time, the controller could be in one of the following states.

- It is not drawing any shapes and is in the `QuiescentSate`. It could respond to some events such as the command to draw a shape, undo the last command, save the drawing, and so on.
- It is drawing a shape and is in `DrawLineState`, `DrawPolygonState`, etc. All drawing processes have some common features (such as the ability to abandon the drawing of that shape), so we expect all these states to inherit some common functionality. Otherwise, the set of events the state responds to depends on the specific shape.

From the above examples, we can see that the FSM approach is an appropriate way to analyze and construct the system. Obviously, the events can be organized into a hierarchy as shown in Figure 9.4.

Separating the Physical and Logical Controllers

We are using JavaFX, which has its specific ways of creating and displaying buttons and responding to clicks on them. It is conceivable that this technology (JavaFX) would be replaced sometime in the future by something more attuned to changes in hardware and user

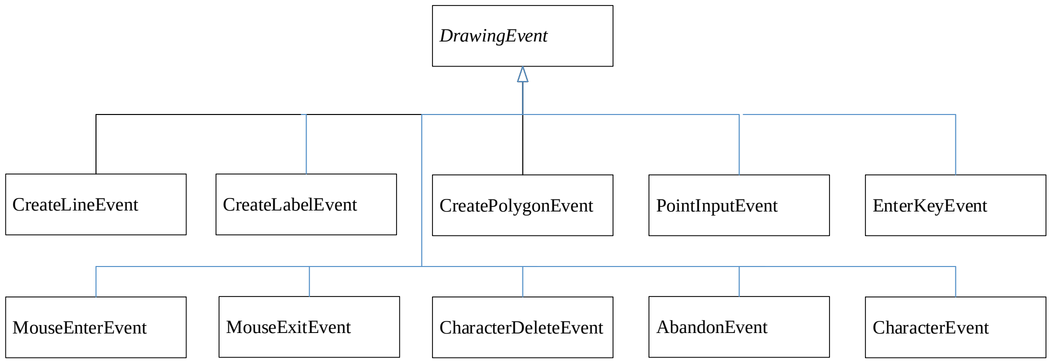

Figure 9.4: The Event Hierarchy for the Initial Version of the Drawing Program. The approach here is similar to that of the example of the microwave in the previous chapter. To add more functionality to the program, we need to add more events to this hierarchy.

interface considerations. If and when that happens, it should be possible to replace the code without too much hassle.

What this means is that we should design the controller taking into account the fact that the technology for GUI construction could change in the future. To handle this, we use an approach we have used before in the future: *Determine what could change and isolate the related functionality in one or more modules.* Our idea is to divide the responsibility of handling the user input into two distinct sets of Java classes. This is shown in Figure 9.5. Note the separation of the subsystem into a physical GUI controller and a logical controller. This set of actions on the GUI might be captured by a set of m physical components. The division of responsibility would depend on, among other things, the GUI technology being employed. For example, a component might receive all text characters, a second component might handle all other key strokes, a third component could be the receiver of single and double clicks on the mouse and all mouse movements, and so on. The sole responsibility of these classes is receiving user input and delivering in a relatively technology-independent way to the components shown in the box labeled "Logical Controller." There may be n different components here, with the division of responsibility being determined completely independent of the components in the physical GUI controller.

The physical GUI classes do not know the organization of the classes that react to the user inputs and vice-versa. All that the physical classes know is that the logical components are capable of reacting to the inputs listed in Table 9.4. This calls for the creation of an interface.

```
public interface Controller {
  public void handleEvent(CreateLineEvent event);
  public void handleEvent(MouseEnterEvent event);
  public void handleEvent(MouseExitEvent event);
  public void handleEvent(PointInputEvent event);
  public void handleEvent(CreatePolygonEvent event);
  public void handleEvent(CreateLabelEvent event);
  public void handleEvent(CharacterEvent event);
  public void handleEvent(DeleteCharacterEvent event);
```

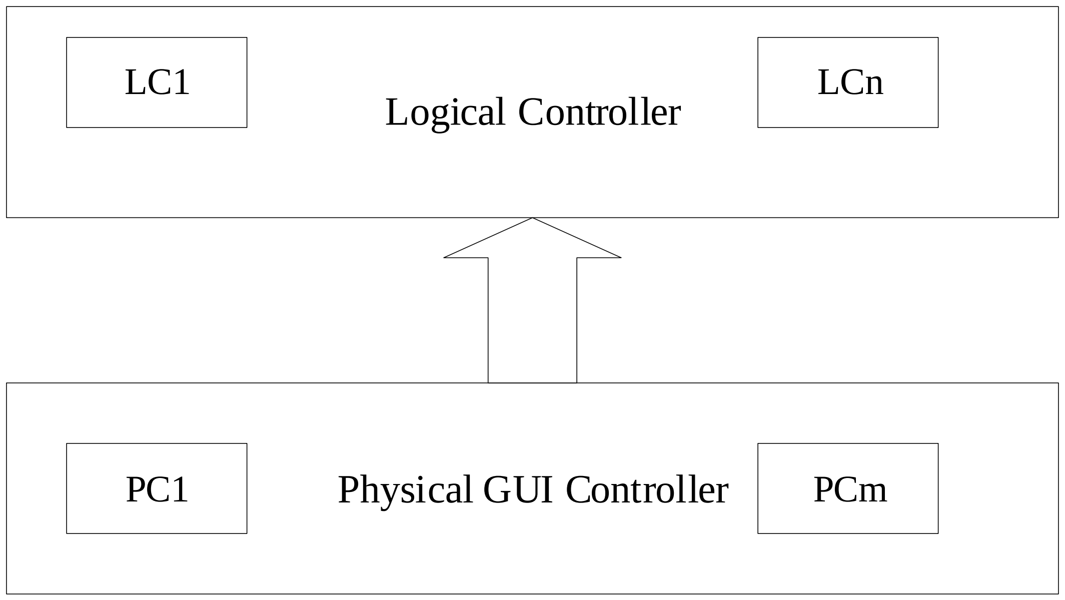

Figure 9.5: The Interface of the Controller Subsystem. Our plan is to separate the GUI components from the logic associated with handling user input. There could be many (say, m) GUI components (PC1 through PCm) that receive actions on the GUI. These m different inputs, could, in general, be handled by n components (LC1 through LCn).

```
  public void handleEvent(AbandonEvent event);
  public void handleEvent(ConfirmEvent event);
}
```

Defining the View

Just as in the case of the controller, the logical steps of rendering should be separated from the physical process. Moreover, we may require multiple views of the model. For example, we could require three different views.

1. A view that only shows the rectangles
2. A view that only shows the last ten shapes
3. A view that shows all the shapes

A natural way to structure the system would be to have three different classes, one for each of the above cases. However, the physical process for each of these is dependent on the technology. Thus, in a very generalized case, we could have multiple “logical views,” each of which implements a specific policy od displaying the model and a set of ”physical views,” with any number of physical views attached to a logical view. When the model changes, all attached logical views draw the shapes, which result in corresponding physical rendering of the shapes by the logical views.

Cast in terms of the user requirements, our system has a single logical view in the entire system: it displays all shapes all the time. But there could be multiple physical views attached to this logical view.

What should be the functionality of a logical view? To answer this question, we find it instructive to examine the use cases again. Here is, for example, the use case for drawing a line, without the columns and the display-related actions highlighted.

1. The user clicks on the Line button in the command panel and moves the cursor to the drawing panel.
2. **The cursor takes the drawing cursor shape.**
3. The user clicks first on one end point and then on the other end point of the line to be drawn.
4. The system adds a line segment with the two specified end points and **displays the line on the screen.**

We now pick out the displays made on the GUI, most of which, by definition of the MVC pattern, are done by the view. These are:

1. The cursor takes the drawing cursor shape.
2. Displays the line on the screen.

One could reason that changing the cursor shape is an activity of the controller, because mouse clicks are inputs. On the other hand, we could also say that changing the cursor occurs when the user moves the mouse to the drawing panel when a command is in progress, so it is a system response and could be thought of as an action by the view. We take the view point that the logical controller makes the decision that the cursor appearance in the drawing panel be changed when a command is in progress and that the view is responsible for making that happen.

In addition, we can extrapolate the use case to add the requirement that the view should also be made to change the cursor to the default shape when no command is in progress.

To be able to draw the shapes, the logical view would be an observer of the model and the physical views would be observers of a logical view. Thus, the following be included in the functionality of the logical view.

1. Listen to property changes in the model.
2. Change the cursor to default.
3. Change the cursor to indicate that drawing is in progress.
4. Draw the shapes in the model, as specified by the user requirements.
5. Function as an observable because the physical views are observers of logical views

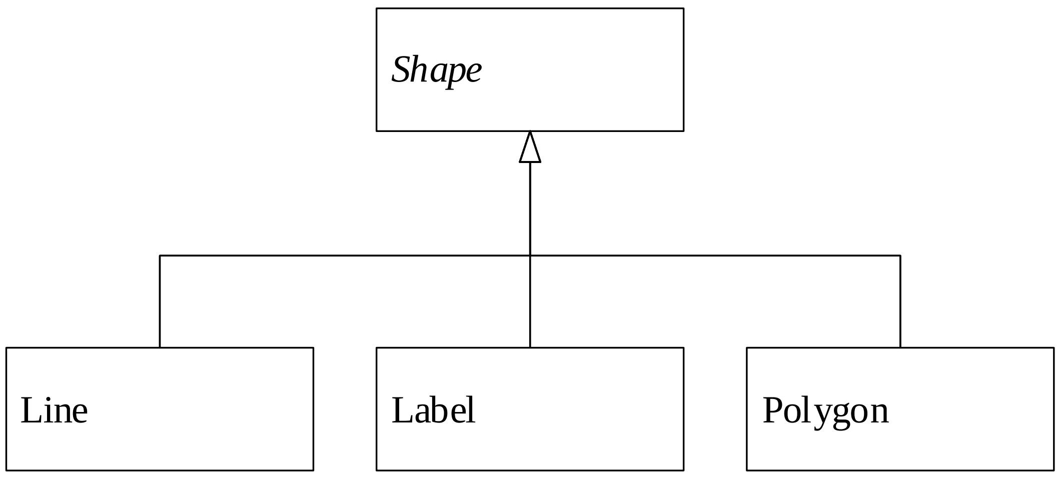

Figure 9.6: The Shape Hierarchy. Since lines, labels, and polygons all are items that can be drawn by the end user, they naturally fit into a hierarchy.

To keep matters relatively simple, we will have a single class called `View` that implements the above functionality of a logical view. Extensions to multiple views would be addressed through a programming project at the end of the chapter.

Representation of the Shapes

Clearly, the various items that the drawing program allows the user to draw, are likely to have common properties and behavior, and from the object-oriented design principles, we know that it is appropriate to have a superclass for all these shapes, which we will call `Shape`. Thus, we have the class hierarchy shown in Figure 9.6.

We expect the individual shapes to have a unique set of fields. For example, the `Line` class will have two fields to store the endpoints. The `Polygon` class could store an array holding the vertices that make up that shape, and so on.

A trickier issue is, however, determining the behaviors of the shapes. One common action that we want to perform on any shape is **rendering**, which is the process by which the data stored in the model is displayed by the view. Regardless of how we implement this, the actual details of how the drawing is done are dependent on the following two parameters:

- *The technology and tools that are used in creating the GUI.* For instance, if we use the Java's `Swing` package, the drawing could be on a `JPanel` and the rendering methods will then have to be invoked on the associated `Graphics` object. If we create a `JavaFX` GUI, we might use that technology's `Canvas` class, and rendering in that case will be done through a `GraphicsContext` object.

- *The shape itself.* If a line is stored by its equation, the code for drawing it would be very different from the line that is stored as two end points.

The technology and tools are known to the author of the physical view, whereas the structure of the shape is known to the author of the shape. Since the needed information is in two different classes (or subsets of classes), we need to decide which class will have the responsibility for implementing the rendering. We have the following options:

- **Option 1.** Let us say that the physical view is responsible for rendering, i.e., there is code in the view that accesses the fields of each item and then draws them. Since the model is storing these items in a polymorphic container, the view would have to query the type of each item returned by the enumeration in order to choose the appropriate method(s).
- **Option 2.** If the shape were responsible, each shape would have a `render()` method that accesses the fields and draws the shape. The problem with this is that the way an object is to be rendered depends on at least two things. First, the functionality available in the toolkit has to be considered. For instance, consider the problem of rendering a circle, which we will take up later in this chapter: a circle is almost always drawn as a sequence of short line segments. If the only method given in the toolkit is that for drawing lines, the circle will have to be decomposed into straight lines. Second, as we have seen, the individual shape classes will have to cater to multiple rendering technologies.

At this point it appears that we are stuck between two bad choices. However, a closer look at the first option reveals a fairly serious problem, which is somewhat intractable: *we are querying each object in the collection to determine its type and decide the appropriate methods to invoke.* This is very much at odds with object-oriented philosophy, where we would like methods to be invoked in a polymorphic manner.

This really means that the `render()` method for each shape should be stored in the shape itself, which is in fact the approach for the second option. This simplifies our task somewhat, so we can focus on the task of fixing the shortcomings of the second option.

Essentially, what we want is that each shape has to be customized for each kind of GUI, which boils down to the task of having a different `render()` method for each type of technology.

One way to accomplish this is to use inheritance. Say that we have three kinds of rendering technologies, the `AWT` (an old Java rendering technology that is now obsolete), `Swing` (which is also not supported) and `JavaFX`.

```
public abstract class Shape {
// some methods that all Shapes in the system must satisfy
  public abstract void render();
}

public abstract class Line extends Shape {...
// some fields and methods
}

public class AWTLine extends Line {
// Line class for AWT rendering
  public void render() {
```

```
  // code to draw a circle using  AWT
  }
}

public class SwingLine extends Line {
// Line class for Swing
  public void render(){
  // code to draw a circle using  Swing
  }
}

public class FXLine extends Line {
// Line class for FX
  public void render(){
  // code to draw a circle using  FX
  }
}
```

In each case, the `render()` method will invoke the methods available in the respective GUI technology to render the item. In addition, each method would have to get any other contextual information. For instance, in the `SwingLine` class, the `render()` method would get the `Graphics` object from the physical view and invoke the `drawLine()` method. The code for this could look something like this:

```
public class SwingLine extends Line {
// Line class for SwingUI
  public void render(){
    Graphics g = (View.getInstance()).getGraphics();
    g.drawLine(...);
  }
}
```

The system could potentially employ many types of shape, each of which has a corresponding abstract class. Each abstract class ends up with as many subclasses as the types of rendering technologies we have to accommodate.

This solution has some drawbacks. The number of classes needed to accommodate such a solution is given by:

Number of Types of Shapes × *Number of Rendering Technologies* (see Figure 9.7).

This causes an unacceptable explosion in the number of classes. Next, consider the situation where shapes are being created in the controller. Some kind of conditional will be needed to decide which concrete class should be instantiated, and this requires the code in the controller to be aware of the rendering technology in use at that time. A third and more subtle point is that of software upgrades. Suppose we create a version of our drawing program that supports `Swing` and we use that to create a figure. All the items created in the model will belong to the `Swing` subclasses, and can be used only with a system where the `Swing` package is available. If a later version of the software does not support `Swing` , (or we move the files to a system that does not support it) we cannot access the old files anymore. If the objects created in the model were independent of the type of GUI, this problem could be avoided.

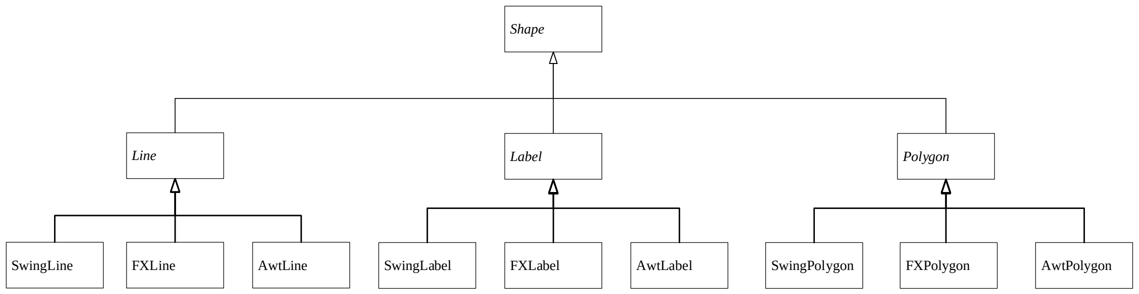

Figure 9.7: Class Explosion Due to Multiple GUI Implementations. We are creating adding a new concrete subclass for each type of renderer to be supported. The three shape classes are abstract because rendering depends on the technology being used.

We now describe an approach that does not result in class explosion. In this approach, we have exactly one class for each shape `Line`, `Polygon`, etc.) and one class for each type of renderer. To understand the principles, observe the following.

- We have two possible ways in which objects can vary: one of shapes and other of rendering technologies. The variation of the rendering technologies can be represented by a second hierarchy. We will use the term internal variation to refer to the differences represented by the `Shape` hierarchy and the term external variation to refer to the variations in the rendering hierarchy.
- Each `Shape` class has a `render()` method, which is responsible for rendering that shape.
- The `render()` method needs to make use of the rendering technology without committing to any specific technology. In other words, the objects of the internal variation needs to make use of the functionality provided by objects in the external variation.

With the two hierarchies in place, we set up a *bridge* between them, which enables The `render()` method needs to make use of the rendering technology without committing to any specific technology. This approach is quite standard in problems of this kind and is called the **Bridge Pattern.**

Figure 9.8 describes the interaction diagram between the classes and visually represents the bridge between the two hierarchies.

The `render()` method needs information from both the GUI and the shape. The GUI information is obtained within the `Renderer` object and the shape is passed in as a reference. We invoke the appropriate drawing methods on the GUI object with which the view has been configured.

Note that the total number of classes is now reduced to

Number of Types of Shapes + Number of Renderers

Since we have only one concrete class for each shape, the creation process is simple. Finally, by factoring out the `render()` method, we are no longer concerned with what kind of GUI is being used or what GUI will be used to edit it at a later stage. Our software for the Model is thus “completely” reusable.

As is often the case in object-oriented design, one price we pay is through a loss of performance. In this case, this is seen in the increased number of method calls. Every time we

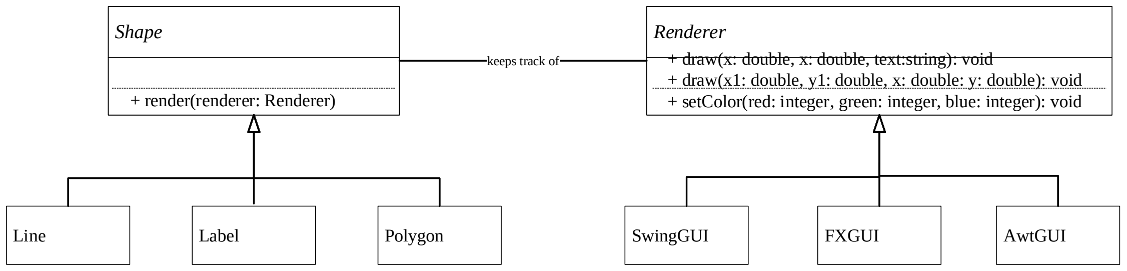
Shape
+ render(renderer: Renderer)
keeps track of
Renderer
+ draw(x: double, x: double, text:string): void
+ draw(x1: double, y1: double, x: double: y: double): void
+ setColor(red: integer, green: integer, blue: integer): void
Line
Label
Polygon
SwingGUI
FXGUI
AwtGUI

Figure 9.8: The Bridge Pattern as Employed in the Drawing Program. The basic principle is that we have two independent hierarchies, with the one on the right allowed to vary. As new GUIs are created, we create appropriate subclasses to the `Renderer` class. To allow for dynamically changing the supported GUI, we chose to make the renderer a parameter of the `render()` method. In a more traditional implementation, we would store the renderer as a field of `Shape`.

invoke the `render()` method, we have to get the `Renderer` object and invoke its functionality. What is left in the design of the Bridge pattern is the following:

- The methods of the `Renderer` hierarchy.
- The flow of control to invoke object rendering.

To see what methods should go into the `Renderer` hierarchy, we first observe the need for an interface named `Renderer`. Clearly, we need to draw lines, circles, draw polygons, and write text. If we assume that any shape with straight lines is implemented as a sequence of coordinates, drawing them could be facilitated with the use of a method to draw a line as below.

```
public abstract void draw(double x1, double y1, double x2, double y2);
```

To draw a circle, we could have a method to draw a full circle, or perhaps, provide a more general method to draw an arc.

```
public abstract void draw(double x, double y, double radius, double startAngle, double endAngl
```

To write a label, we could have the method,

```
public abstract void draw(double x, double y, String text);
```

If a shape were to store the information as a set of points, as opposed to a sequence of coordinates, the above set of methods may not be convenient. We could provide a method to draw a pixel.

```
public abstract void draw(double x, double y);
```

Since we would like to vary the shape color, we add the following method to the interface.

```
public abstract void setColor(int red, int green, int blue);
```

While colors could be represented differently in different GUI technologies, we could adopt the convention that each of the parameters represents the respective color value on a s]scale from 0 to 255.

Finally, as a catch all approach, we add the following method, which helps us admit defeat when the above functionality is inadequate to render a certain shape. We hope there would not be a need to use it (using it could be a signal a weak interface), but it does not harm to have it in the interface.

```
public abstract void draw();
```

Thus, the interface would be represented by the UML diagram in 9.9.

9.4.3 The Physical Controller and Views

We construct the drawing program controller and view using JavaFX. We have read-only views built using both FX and Swing technologies.

Controller

The controller contains the following devices:

- A set of buttons for selecting the shape to be drawn,
- Handlers for mouse clicks, mouse movements, key strokes, etc.
- Menus for selecting the different read-only views.

Table 9.11 shows the classes involved in handling the event and the correspondng methods in DrawingContext that routes these events.

The Button Classes The buttons form a hierarchy as shown in Figure 9.12. The design follows the principles described in Chapter 8. There is a separate `handleEvent()` method in `DrawingContext` for each button click event.

Dealing with Key Strokes Key strokes are classified into two types and are handled as follows.

- The typing of ‘printable” characters such as alphanetic characters, digits, punctuations, etc. These are received in the `handle()` method of `KeyTypedHandler` and control is transferred to `handleEvent(CharacterKeyEvent)` method of `DrawingContext`. The character typed is stored in the event object.
- The pressing of keys such as the Enter key is handled by the `handle()` method of `KeyPressedHandler`. As in other cases, this calls the appropriate method of `handleEvent()` method of `DrawingContext`. The possible parameters are `BackSpaceKeyEvent`, `EnterKeyEvent`, and `EscapeKeyEvent`.

Handling Mouse Clicks The class `MouseClickHandler` deals with mouse clicks. Its `handle` method calls the `handleEvent(MouseClickEvent)` method of `DrawingContext`.

Monday, June 29, 2020 3:58 PM

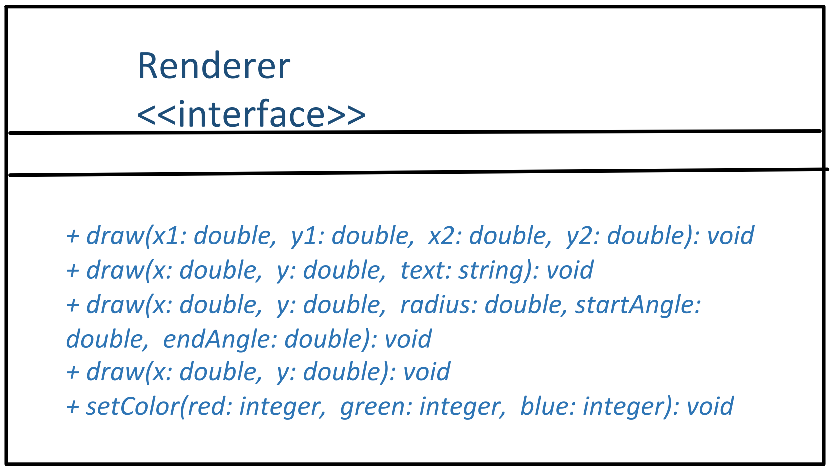
Renderer
<<interface>>
+ draw(x1: double, y1: double, x2: double, y2: double): void
+ draw(x: double, y: double, text: string): void
+ draw(x: double, y: double, radius: double, startAngle: double, endAngle: double): void
+ draw(x: double, y: double): void
+ setColor(red: integer, green: integer, blue: integer): void

Figure 9.9: The Renderer interface. The primary methods are to draw a line, an arc, and write a string. The expectation is that most drawing needs can be accomplished by invoking these methods in some combination. To cater to rendering shapes whose coordinates are stored by an equation or as a set of pixels, we provide the `draw(double x, double y)` method. The color can be changed using the `setColor() method. There should be no need to us ethe draw() method without parameters.`

Wednesday, July 1, 2020 1:28 PM

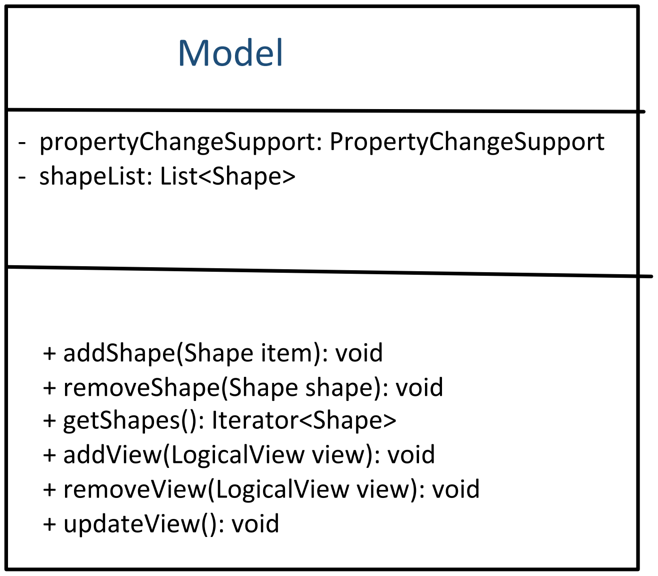
Model
- propertyChangeSupport: PropertyChangeSupport
- shapeList: List<Shape>
+ addShape(Shape item): void
+ removeShape(Shape shape): void
+ getShapes(): Iterator<Shape>
+ addView(LogicalView view): void
+ removeView(LogicalView view): void
+ updateView(): void

Figure 9.10: The Model. It is essentially a collection of `Shape` objects and an Observable in the Observer pattern. Thus it has methods to add and remove `Shape` objects and `LogicalView` objects. The `updateView()` method sends an update notification to every view connexted to the Model.

JavaFX Class Handling the Event	Method in `DrawingContext`
CreateLineButton	handleEvent(CreateLineEvent event)
CreatePolygonButton	handleEvent(CreatePolygonEvent event)
CreateLabelButton	handleEvent(CreateLabelEvent event)
KeyTypedHandler	handleEvent(CharacterEvent event)
KeyPressedHandler	handleEvent(DeleteCharacterEvent event)
KeyPressedHandler	handleEvent(AbandonEvent event)
KeyPressedHandler	handleEvent(ConfirmEvent event)
MouseClickHandler	handleEvent(PointInputEvent event)
SwingViewMenuItem	handleEvent(SwingViewMenuItem event)
DraingPanel	handleEvent(MouseEnterEvent event)
FXViewMenuItem	handleEvent(FXViewMenuItem event)
DraingPanel	handleEvent(MouseExitEvent event)

Figure 9.11: List of classes and methods in DrawingContext.

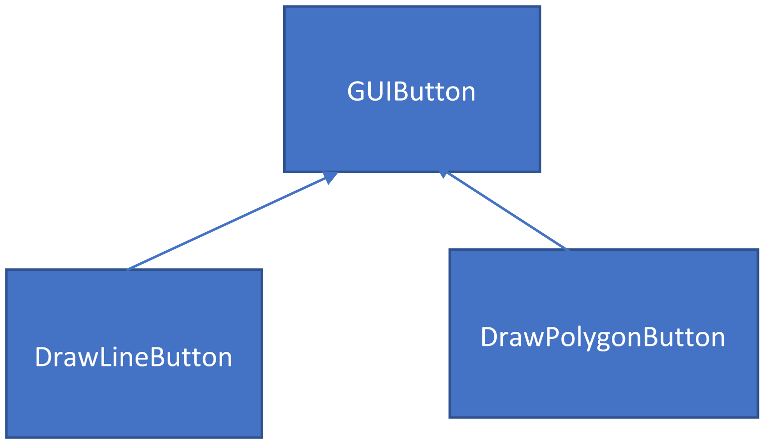

Figure 9.12: The Button Hierarchy. For each command issued through a button click, we have one class. The design follows the same principles described in Chapter 8. The `GUIButton` class is the abstract superclass that configures a button object. Each subclass overrides the `handle` method to call the appropriate method in `DrawingContext`
.

Processing Clicks on Menu Items There are two menu items: one each for instantiating a read-only view implemented using JavaFX and Java Swing. The two classes FXViewMenuItem and SwingViewMenuItem both extend `MenuItem` (a JavaFX class), override the `handle()` method to handle the click on the menu item and notify `DrawingContext` via the respective `handleEvent()` method.

Sensing Mouse Movements Since the shape of the cursor changes as the mouse moves between the drawing and button panels, these events have to be sensed. This is conveniently done in the `DrawingPanel` class, which is an extension of `FXView` discussed in the next subsection.

9.4.4 View

Unlike the controller, there are multiple views:

- A view associated with the controller, which is always visible. As we discussed earlier, this is constructed using JavaFX.
- Multiple read-only views, which are displayed optionally.

For each technology we wish to use for a view (read-only or not), we have a corresponding panel class, which displays the shapes in a technology dependent fashion. Thus there are multiple "physical view" objects.

Recall that for the sake of simplicity, we have a single logical view of the system and a logical view object is instantiated as a client of `Model`. Since the code in the physical view classes is dependent on the technology, we provide a uniform way of accessing the physical views using an interface `PhysicalView`.

`PhysicalView` extends `PropertyChangeListener` and is, therefore, an observer. Every physical view observes the logical view,so when drawing has to take place, the logical view can invoke the `propertyChange()` method of the physical view, which draws the shapes in a way appropriate way for its technology.

We implement each view as two separate classes: a panel where the drawing occurs and the window that stores the panel. We begin with Java Swing. The reader should note some some intricacies involved in the drawing process.

The two classes are `SwingView` and `SwingPanel`. The `SwingPanel` class implements `PhysicalView` and can thus be a client of the logical view. The logical view notifies the physical view by calling the `propertyChange()` method, which invokes the `repaint()` method. This results in a redraw of the screen. All painting occurs in the `paintComponent()` method, using the `Graphics` object passed as parameter to this method. The method sets up a `Renderer` object, it is ready to have the shapes drawn on the panel. Since the responsibility of drawing the shapes rests with the respective `Shape` objects, and the orchestration of this process is done by the logical view, the `paintComponent()` invokes a callback method (which we call `draw()`) in the logical view, which decides which proceeds with the logical drawing. The `draw()` method of the logical view receives a reference to a `Renderer` object and passes this to the `draw()` method of each of the shapes to be drawn. The sequence diagram is shown in Figure 9.13.

The `SwingView` class is a window extending `JFrame` and holds the reference to a `SwingPanel` object.

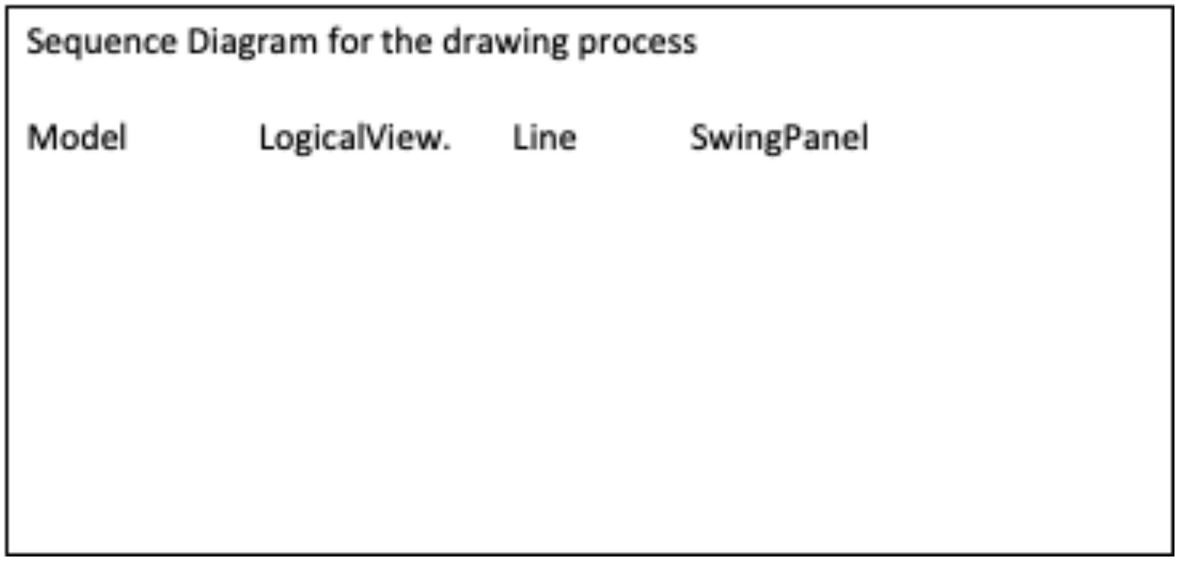

Figure 9.13: The drawing of shapes. All painting has to occur in the `paintComponent()` method. Therefore, this method has to invoke a callback method (which we call `draw()` in the logical view, which decides which proceeds with the logical drawing. The `draw()` method of the logical view receives a reference to a `Renderer` object and passes this to the `draw()` method of each of the shapes to be drawn.

FXPanel and FXView Apart from find differences associated with the two technologies, the two classes parallel `SwingPanel` and `SwingView`. `FXPanel` stores a `Canvas` object on which the shapes are drawn. Upon being notified through the `propertyChange()` method of the need to draw the shapes, the method gets the `GraphicsContext` object associated with the canvas, creates a `Renderer` object using it and, like `SwingView`, invokes the `draw()` method of the logical view.

The `FXView` class extends `Stage` and holds the reference to a `FXPanel` object.

The Drawing Program Class A class named `DrawingProgram`, based on JavaFX, sets up the GUI for drawing and viewing the shapes. It is made up of two objects: a `ButtonPanel` object storung all the buttons and a `DrawingPanel` object that extends `FXPanel` and as desrobed earlier, tracks the entering and exiting of mouse.

9.4.5 Implementation

The system is implemented as a set of packages, whose member classes/interfaces are indicated in the table in 9.14. The dependencies between the packages are shown in Figure 9.15.

We now look at the code for some of the classes and interfaces. We organize the discussion in terms of the four major packages, `model`, `view`, `controller`, and `gui`.

Packaage	Any Subpackaages	Any Classes/Iterfaces
`model`	`shapes`	the `Model` class
`model.shapes`	None	the `Shape` interface and the concrete classes implementing it
`view`	None	The logical view class `View`, the physical view interface `PhysicalView`, `Renderer` interface
`controller`	`events`, `states`	the `Controller` interface
`controller.events`	None	all the event classes and the superclass `DrawingEvent`
`controller.states`	None	`DrawingContext`, `DrawingState`, and the concrete state classes
`gui`	`controller`, `panels`, `view`	the `DrawingProgram` class
`gui.controller`	`buttons`, `handlers`,	None
`gui.controller.buttons`	None	all the button classes, their superclass `GUIButton`, and the two menu items, `SwingViewMnuItem` and `FXViewMenuItem`
`gui.controller.handlers`	None	`KeyPressedHandler`, `KeyTypedHandler`, and `MouseClickHandler`
`gui.panels`	None	the panel classes `FXPanel`, `SwingPanel`, `ButtonPanel`, and `DrawingPanel`
`gui.view`	`renderers`, `views`	None
`gui.view.renderers`	None	`FXRenderer` and `SwingRenderer`
`gui.view.views`	None	the two read-only view classes `FXView` and `SwingView`

Figure 9.14: The list of packages and their members.

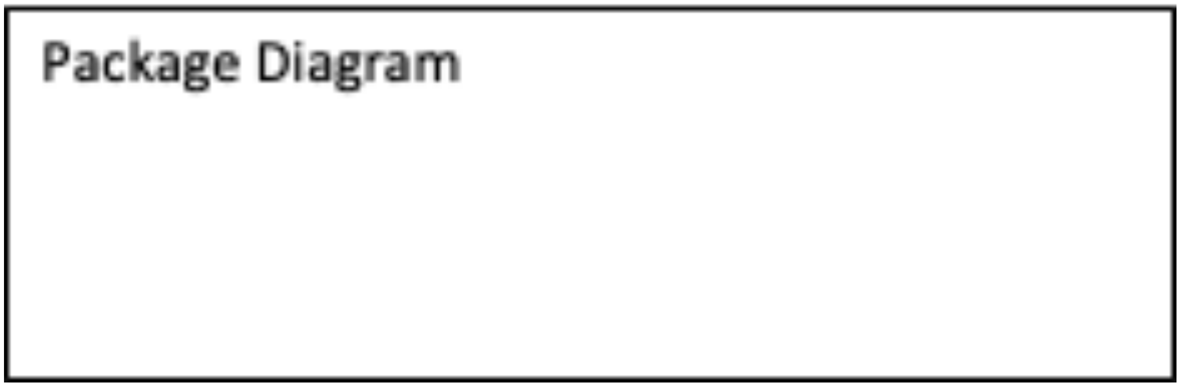

Figure 9.15: The interaction among the packages.

.

Model and Its subpackages

The `Model` class is a singleton collection. It also serves as an observable in the `Observer` pattern, notifying the view when objects are added or removed from the collection. Besides the standard code for implementing singleton, it has the following fields.

```
private PropertyChangeSupport propertyChangeSupport = new PropertyChangeSupport(this);
private List<Shape> shapeList;
```

The first field is for implementing the observable code and the second field stores the shapes.

The methods for adding and removing shapes are straightforward. They call the `updateView()` method to notify the observers.

```
public void addShape(Shape shape) {
  shapeList.add(shape);
  updateView();
}
public void removeShape(Shape shape) {
  shapeList.remove(shape);
  updateView();
}
public void updateView() {
  propertyChangeSupport.firePropertyChange(null, null, null);
}
```

There are methods to add and remove an observer (the views), which are straightforward.

The shapes Sub-Package The `Shape` class is the abstract superclass for all shapes.

```
public abstract class Shape {
  protected int red = 0;
  protected int green = 0;
  protected int blue = 255;
  public void render(Renderer renderer) {
    renderer.setColor(red, green, blue);
  }
```

The class stores the color of the shape. The color values (red, green, and blue) range from 0 to 255. The `render()` method sets the color of the `Renderer` object (depends on the technology) to the desired one by passing the values of `red`, `green`, and `blue`.

As you can see, the default color is blue, but the color of a shape can be changed by invoking the `setColor()` method. The getters for the three colors are not shown.

```
public void setColor(int red, int green, int blue) {
  this.red = red;
  this.green = green;
  this.blue = blue;
}
```

The most complicated subclass of `Shape` is `Polygon`.

```
public class Polygon extends Shape {
  private Point2D[] points;
  private int numberOfPoints;
  public Polygon() {
    points = new Point2D[10];
  }
  public void addPoint(Point2D point) {
    if (this.points.length == numberOfPoints - 1) {
      Point2D[] temp = new Point2D[numberOfPoints * 2];
      System.arraycopy(points, 0, temp, 0, numberOfPoints);
      points = temp;
    }
    points[numberOfPoints++] = point;
    points[numberOfPoints] = point;
  }
  public void end() {
    addPoint(points[0]);
  }
  public void render(Renderer renderer) {
    super.render(renderer);
    for (int index = 0; index < numberOfPoints; index++) {
      renderer.draw(points[index].getX(), points[index].getY(),
                    points[index + 1].getX(), points[index + 1].getY());
    }
  }
}
```

The class maintains an array `points` of `Point2D` objects, which are the vertices of the polygon. The initial array length is 10, but if more vertices arrive a new array is allocated and the vertices copied to the newly-allocated array. The field `numberOfPoints` maintains the number of vertices.

It is important to know how we maintain the vertices. Suppose a polygon has been completely drawn and it has three vertices p1, p2, and p3, in that order. Then we store 4 objects in `points` with cells at index values 0 through 3 storing p1, p2, p3, and p1. The field `numberOfPoints` is 4, not 3. When the polygon is to be drawn, we simply have a loop that iterates `numberOfPoints` times to draw the *line* from each vertex to the next (`points[index]` to `points[index + 1]`.

As each vertex is clicked, the method `addPoint()` should be called.

Observe the code

```
points[numberOfPoints++] = point;
points[numberOfPoints] = point;
```

This ensures that the loop to draw the polygon works correctly. The last line to draw an incomplete polygon would be from the last point to itself.

Also observe that `super.render()` is called before rendering the lines of the polygon to ensure that the color is as set in the fields.

Members of the controller Package

We begin with the event classes. `DrawingEvent` is an "empty" class, which simply serves as a superclass for all events. We divide the subclasses into two categories: singletons and non-singletons. The singleton classes are: `BackspaceKeyEvent`, `EnterKeyEvent`, `EscapeKeyEvent`, `CreateLabelEvent`, `CreateLineEvent`, `CreatPolygonEvent`, `FXViewRequestEvent`, `SwingViewRequestEvent`, `MouseEnterEvent`, `MouseExitEvent`, and `ShiftEnterKeyEvent`. These classes are implemented using the standard approach for implementing the Singleton pattern. They do not hold any data relevant to the application.

The non-singleton event classes are distinguished from the above due to their need to hold event-specific data, which the drawing program needs. We look at one example: `PointInputEvent`, which is instantiated when the mouse is clicked. The class stores the x and y coordinates of the point at which the mouse is clicked and has getters for those values.

```
public class PointInputEvent extends DrawingEvent {
    private int x;
    private int y;
    public PointInputEvent(int x, int y) {
       this.x = x;
       this.y = y;
    }
    public int getX() {
       return x;
    }
    public int getY() {
       return y;
    }
}
```

The other non-singleton classes are implemented in a similar fashion.

The `Controller` interface contains a method for each event. For example, here are a couple of the methods.

```
public void handleEvent(InputPointEvent event);
public void handleEvent(CreatePolygonEvent event);
```

controller.events All events go through `DrawingContext`, which implements `Controller`. In addition, the class maintains the current state and a reference to the current shape (if any) being created. The class is a singleton, so the standard code for making it a Singleton is also part of this class.

```
public class DrawingContext implements Controller {
  private DrawingState currentState;
  private static DrawingContext instance;
  private Shape shape;
  // private constructor and methods
}
```

The methods in `DrawingContext` are quite straightforward. The `changeCurrentState()` method works as in `MicrowaveContext` in Chapter 8. The setter and getter for `shape` are called from the state classes: `DrawingState` and its subclasses.

```
public void changeCurrentState(DrawingState nextState) {
  currentState.leave();
  currentState = nextState;
  currentState.enter();
}
public void setShape(Shape shape) {
  this.shape = shape;
}
public Shape getShape() {
  return shape;
}
\end{verbatimm}

We show one example of a {\tt handleEvent()} method; the others are identical, except for the
\begin{verbatim}
public void handleEvent(CreateLineEvent event) {
  currentState.handleEvent(event);
}
```

Classes in the states package Now onto the state classes. `DrawingState` is an abstract superclass of all other state classes. The class has no fields. It has a corresponding method for each method in `Controller` and the methods `enter()` and `leave()`. The method body is empty in all except three of the methods.

When the Escape key is pressed, any shape that be in construction should be discarded. As you can see in the `handleEvent(AbandonEvent event)` method, the `Shape` reference is retrieved from the context and if it is not `null`, the shape is deleted from the model. The state is then changed to `QuiescentState`.

The `handleEvent(FXViewRequestEvent event)` method simply creates an `FXView` object, which adds itself as a listener of the logical view. A similar comment applies to `handleEvent(SwingViewRequestEve` `event)`.

```
public abstract class DrawingState {
  public void handleEvent(AbandonEvent event) {
    Shape shape = DrawingContext.instance().getShape();
    if (shape != null) {
      Model.instance().removeShape(shape);
      DrawingContext.instance().setShape(null);
    }
    DrawingContext.instance().changeCurrentState(QuiescentState.instance());
  }
  public void handleEvent(FXViewRequestEvent event) {
    new FXView();
  }
  public void handleEvent(SwingViewRequestEvent event) {
    new SwingView();
  }
  // all other methods are empty
}
```

When no drawing is taking place, the system is in the quiescent state (`QuiescentState`), a subclass of `DrawingState`. When we enter this state, the cursor shape in the drawing panel should assume the default shape. This is done by overriding the `enter()` method and make the view set the cursor accordingly.

```
public void enter() {
  View.instance().setCursorToDefault();
}
```

The system leaves the state when a drawing command is issued. At that time, the cursor in the drawing panel has to take a different shape. This shape is set by instructing the view.

```
public void leave() {
  View.instance().setCursorToDrawing();
}
```

When a command to draw a shape is invoked, the state changes. Here is the code to change the state to draw a line.

```
public void handleEvent(CreateLineEvent event) {
  DrawingContext.instance().changeCurrentState(new LineDrawingState());
}
```

We end this part of the implementation discussing two classes: one representing the state for drawing labels and and other for drawing polygons.

Let us start with `LabelDrawingState`. It has to respond to the following events.

1. Specification of the point at which a new label begins.
2. Typing of a character, which becomes part of a label being constructed.
3. `DeleteCharacterEvent` to delete the last character.
4. Confirmation of the end of the creation of labels via the `ConfirmEvent`.

Here is the code for Case 1 in the above list. When the mouse is clicked, we have two possible situations: a label creation is already in progress, in which case, `DrawingContext` holds that shape.

For labels in construction, for simplicity, we indicate the position where the next character would be entered by '—', which is stored as part of the label. Before starting a new label, this character has to be deleted. All of this is done within the block associated with the `if`.

In any case, we have to create a new label at the clicked position. The label is created, the character '—', added, the shape stored in `DrawingContext` and the model.

```
public void handleEvent(PointInputEvent event) {
  Shape shape = DrawingContext.instance().getShape();
  if (shape != null) {
    Label label = (Label) shape;
    label.removeCharacter();
    DrawingContext.instance().setShape(null);
  }
  Label label = new Label(new Point2D(event.getX(), event.getY()));
  DrawingContext.instance().setShape(label);
  label.addCharacter("|");
  Model.instance().addShape(label);
}
```

For Case 2, we need to add the character to the label, assuming one is in construction. If no label is being constructed, the character is ignored via the check at the beginning of the following code. Otherwise, the current label is retrieved from the context, the '—' deleted, and the new character appended, and the '—' reinstated.

```
public void handleEvent(CharacterEvent event) {
  Shape shape = DrawingContext.instance().getShape();
  if (shape == null) {
    return;
  }
  Label label = (Label) shape;
  label.removeCharacter();
  label.addCharacter(event.getCharacter() + "|");
  View.instance().update();
}
```

The following method handles Case 3. If a label is in construction, the last character is deleted, which requires first removing the '—', then the actual character, and finally putting back the '—'. The `removeCharacter()` of label checks its length to ensure that the code does not crash when the label is empty.

```
public void handleEvent(DeleteCharacterEvent event) {
  Shape shape = DrawingContext.instance().getShape();
  if (shape == null) {
    return;
  }
  Label label = (Label) shape;
  label.removeCharacter();
  label.removeCharacter();
  View.instance().update();
}
```

The code for the last case is given below. Since no more labels are created as part of this command, in addition to removing the '—' from this label and setting the shape stored in the context to `null`, we change the current state to `QuiescentState`.

```
public void handleEvent(ConfirmEvent event) {
 Shape shape = DrawingContext.instance().getShape();
 if (shape == null) {
   return;
 }
 Label label = (Label) shape;
 label.removeCharacter();
 View.instance().update();
 DrawingContext.instance().setShape(null);
 DrawingContext.instance().changeCurrentState(QuiescentState.instance());
}
```

The view Package

This package contains a class and two interfaces. The class `View` is a singleton class, a logical view and an observer of `Model`. It therefore implements `PropertyChangeListener`. Besides, the `View` object communicates with the physical views to get the logical rendering operations appear on the screen and for this, it maintains a `PropertyChangeSupport` field and adds and removes physical views via two public methods, which are shown below.

```
public class View implements PropertyChangeListener {
  private PropertyChangeSupport propertyChangeSupport = new PropertyChangeSupport(this);
  public void addPhysicalView(PhysicalView physicalView) {
    propertyChangeSupport.addPropertyChangeListener(physicalView);
  }
  public void removePhysicalView(PhysicalView physicalView) {
    propertyChangeSupport.removePropertyChangeListener(physicalView);
  }
// other code
}
```

Every physical view class implements the following interface.

```
public interface PhysicalView extends PropertyChangeListener {
  public void setCursorToDefault();
  public void setCursorToDrawing();
}
```

When the view gets notified by the model, it calls the `update()` method, which notifies each physical view that it should prepare itself for drawing.

```
public void update() {
  for (PropertyChangeListener listener : propertyChangeSupport.getPropertyChangeListeners()) {
    PhysicalView view = (PhysicalView) listener;
    view.propertyChange(null);
  }
}
```

The physical views are GUI technology dependent and hence reside in subpackages of `gui`. Once a physical view prepares itself for drawing, it calls the following `draw()` method of

`View`, and it is this method that decides the logical aspects of drawing: which shapes should be drawn, what their colors should be, etc. Note that in the following implementation, the view draws all shapes in their default color.

Observe the parameter to the method, which is the `Renderer` object appropriate for the specific physical view. This object is passed to the `Shape` object's `render()` method, for rendering using the Bridge pattern.

```
public void draw(Renderer renderer) {
  for (PropertyChangeListener listener : propertyChangeSupport.getPropertyChangeListeners()) {
    Iterator<Shape> shapes = Model.instance().getShapes();
    while (shapes.hasNext()) {
       Shape shape = shapes.next();
       shape.render(renderer);
    }
  }
}
```

The `Renderer` interface is a straightforward code based on the design we discussed earlier.

```
public interface Renderer {
  public abstract void draw(double x1, double y1, double x2, double y2);
  public abstract void draw(double x, double y, String text);
  public abstract void draw(double x, double y, double radius,
                  double startAngle, double endAngle);
  public abstract void draw();
  public abstract void setColor(int red, int green, int blue);
  public abstract void draw(double x, double y);
}
```

Two remaining methods of this class involve switching the cursor to the appropriate shape.

```
public void setCursorToDefault() {
  for (PropertyChangeListener listener : propertyChangeSupport.getPropertyChangeListeners()) {
    ((PhysicalView) listener).setCursorToDefault();
  }
}

public void setCursorToDrawing() {
  for (PropertyChangeListener listener : propertyChangeSupport.getPropertyChangeListeners()) {
    (PhysicalView) listener).setCursorToDrawing();
  }
}
```

The gui Package and Its Subpackages

All code needed to adapt to a new GUI technology resides in this package and its subpackages. Interfaces such as `Renderer` and `PhysicalView` should be implemented by these GUI classes.

Classes in the `buttons` package The `GUIButton` is the superclass of all button classes and its constructor provides a uniform look to all the buttons. Besides setting specific size and paddings to buttons, note that the code makes each button handle its own clicks.

```
public GUIButton(String string) {
  super(string);
  setPrefWidth(150);
  setPrefHeight(20);
  setPadding(new Insets(10, 30, 10, 30));
  setOnAction(this);
}
```

The `CreateLabelButton` calls the `handlEvent()` method of `DrawingConetxt` when it is clicked.

```
public void handle(ActionEvent event) {
  DrawingContext.instance().handleEvent(CreateLabelEvent.instance());
}
```

Similar code is in place in other buttons.

The Menu Item Classes The two classes `FXViewMenuItem` and `SwingViewMenuItem` are very similar. The following code for `FXViewMenuItem` shows an anonymous class that overrides the `handle()` method being instatntiated and made to call the appropriate `handlEvent()` method of `DrawingConetxt`.

```
public class FXViewMenuItem extends MenuItem {
  public FXViewMenuItem() {
    super("FX View");
    setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent event) {
        DrawingContext.instance().handleEvent(FXViewRequestEvent.instance());
      }
    });
  }
}
```

The `gui.handlers` package The classes here respond to events on the keypad and mouse clicks. All the classes override the `handle()` method.

The `KeyPressedHandler` handles the three events (pressing of Backspace, Enter, and Escape keys) using selection statements.

```
public class KeyPressedHandler implements EventHandler<KeyEvent> {
  @Override
  public void handle(KeyEvent keyPressed) {
    if (keyPressed.getCode().equals(KeyCode.BACK_SPACE)) {
      DrawingContext.instance().handleEvent(DeleteCharacterEvent.instance());
    } else if (keyPressed.getCode().equals(KeyCode.ESCAPE)) {
      DrawingContext.instance().handleEvent(AbandonEvent.instance());
    } else if (keyPressed.getCode().equals(KeyCode.ENTER)) {
      DrawingContext.instance().handleEvent(ConfirmEvent.instance());
    }
  }
}
```

The `KeyTypedHandler` needs to embed the typed character into the event. Thus, the character that was typed is passed to the constructor of `CharacterKeyEvent`. public void handle(KeyEvent keyPressed) DrawingContext.instance().handleEvent(new CharacterEvent(keyPressed.getCharacter

In a way analogous to `KeyTypedHandler`, `MouseClickHandler` needs to store the coordinates of the clicked point ito the event. See the following code.

```
public class MouseClickHandler implements EventHandler<MouseEvent> {
  public void handle(MouseEvent mouseEvent) {
    DrawingContext.instance()
      .handleEvent(new PointInputEvent((int) mouseEvent.getX(), (int) mouseEvent.getY()));
  }
}
```

9.5 Undo and Redo

Clearly, to implement undo, some kind of a stack would be needed to remember the operations that have been completed. When an undo is requested, an element from the top of the stack is popped, and this element has to be "decoded" to find out what the last operation was. This would require some kind of conditional, and the complexity of this method would increase with the number of different kinds of operations that we implement. In earlier chapters we have seen how such complexity can be reduced by *replacing conditional logic with polymorphism.* In the next section we examine some patterns that can help us improve the design of the controller.

In the context of implementing the undo operation, a few issues need to be highlighted.

- *Single-level undo vs multiple-level undo.* A simple form of undo is when only one operation (i.e., the most recent one) can be undone. This is relatively easy, since we can afford to simply clone the Model before each operation, and restore the clone to undo.
- *Undo and redo are unlike the other operations.* If an Undo operation is treated the same as any other operation, then two successive undo operations cancel each other out, since the second undo reverses the effect of the first undo and is thus a redo. The undo (and redo) operations must therefore have a special status as meta-operations, if several operations must be undone.
- *Blocking further undo operations.* This can happen for two reasons. Some operations like "print file" are irreversible, and hence undo-able. Other operations like " save to disk" may not be worth the trouble, to undo, due to the overheads involved.

- *Blocking further redo operations.* It is easy to see that uncontrolled undo and redo can result in meaningless requests. Consider a situation where we have the sequence:

  ```
  select(shape)
  shape.setColor('blue')
  undo
  delete(shape)
  redo
  ```

  A redo means we are trying to undo the last undo operation. In the above example, the last undo operation was to reverse the change of color of `shape` to blue. Therefore, with redo, we are trying to change the color of the deleted object `shape` back to blue. Without undoing the deletion of `shape`, it is logically inconsistent to perform this color change. It may therefore be necessary for a system to disallow all redo operations under certain circumstances.

- *Solution should be efficient.* This constraint rules out naive solutions like saving the model to disk after each operation.

Keeping these issues in mind, a simple scheme for implementing undo could be something like this:

1. Create a stack for storing the history of the operations.
2. For each operation, define a data class that will store the information necessary to undo the operation.
3. Implement code, so that whenever any operation is carried out, the relevant information is packed into the associated data object and pushed onto the stack.
4. Implement an `undo` method in the controller that simply pops the stack, decodes the popped data object and invokes the appropriate method to extract the information and perform the task of undoing the operation.

One obvious approach for implementing this is to define a class `StackObject` that stores each object with an identifying `String`.

```
public class StackObject{
  private String name;
  private Object info;
  public StackObject(String string, Object object) {
    name = string;
    info = object;
  }
  public String getName() {
    return name;
  }
  public Object getInfo() {
    return info;
  }
}
```

Each command has an associated object that stores the data needed to undo it. The class corresponding to the operation of adding a line is shown below.

```
public class LineObject {
  private Line line;
  public Line  getLine() {
    return line;
  }
  public LineObject(Line line) {
    this.line = line;
  }
  public Line getLine() {
    return line;
  }
}
```

When the operation for adding a line is completed, the appropriate `StackObject` instance is created and pushed on the stack.

We define a `Stack` object `history` that is accessible to all states that draw shapes.

```
Stack history;
```

We could do the following in the course of drawing a line.

```
public class LineDrawingState {
  public void handleEvent(PointInputEvent event) {
    Shape shape = DrawingContext.instance().getShape();
    if (shape == null) {
      shape = new Line(new Point2D(event.getX(), event.getY()));
      Model.instance().addShape(shape);
      DrawingContext.instance().setShape(shape);
      history.push(new StackObject("line", new LineObject(line)));
      View.instance().update();
    }
    // rest of the code in the method as before
}
```

Decoding is simply a matter of popping the stack reading the string. We could create a new class `UndoManager` to take care of all undo's.

```
public class UndoManager {
  public  void undo(){
    StackObject stackObject = history.pop();
    String name = stackObject.getName();
    Object info = stackObject.getInfo();
    switch(name) {
    case "line"  : undoLine((LineObject) info);
    case "polygon": undoPolygon((PolygonObject) info);
    // one case for each command
    }
  }
}
```

Finally, undo-ing is simply a matter of retrieving the reference to and removing the line form the Model.

The `undoLine()` method would be along the following lines. The method could be in `UndoManager`.

```
  public void undoLine(LineObject lineObject){
    Line line = lineObject.getLine();
    model.removeShape(line);
  }
```

There are two obvious drawbacks with this approach:

- The long conditional statement in the `undo()` method of `UndoManager`.
- The task of rewriting the code whenever we make changes like adding or modifying the implementation of an operation.

The object-oriented approach for dealing with the first drawback is to subclass the behavior by creating an inheritance hierarchy and *replace conditional logic with polymorphism.* (Recollect that this is accomplished by making the original method abstract and moving each leg of the conditional to an overriding method in the corresponding subclass.) Let us refactor the code to accomplish this. Before replacing the conditional, however, we see that the `undo()` method is mostly working off the data stored in `StackObject` and our

```
public class UndoManager {
  private Stack history;
  public  void undo() {
    StackObject stackObject = history.pop();
    stackObject.undo(this);
  }
  public Stack getHistory() {
    return history;
  }
  // other fields and methods
}

public class StackObject {
  public void undo(UndoManager undoManager) {
    StackObject stackObject = undoManager.getHistory().pop();
    String name = stackObject.getName();
    Object info = stackObject.getInfo();
    switch(name) {
      case "line"  : undoLine((LineObject) info);
      case "delete": undoDelete((DeleteObject) info);
      case "select": undoSelect((SelectObject) info);
      // one case for each command
    }
  }
// other fields and methods
}
```

Now our conditional is in `StackObject` and we are ready to subclass this behavior. Since each kind of data object is associated with an operation our hierarchy will have a subclass corresponding to each operation.

```
public abstract class StackObject{
  public abstract void undo(UndoManager undoManager);
  // other fields and methods
}

public class LineObject extends StackObject {
  private Line line;
  public Line getLine() {
    return line;
  }
  public LineObject(Line line) {
    this.line = line;
    Model.addShape(line);
  }
  public void undo(UndoManager undoManager) {
    undoManager.undoLine(this);
  }
}
```

This code is a lot simpler and cleaner, although we have paid a price by increasing the number of method calls. Note that we no longer "decode" the stored objects and therefore the name field is not required. The code in `LineDrawingState` is modified in the following ways:

- It just creates a `LineObject` and pushes it onto the stack.
- The context stores a reference to `StackObject`, rather than a reference to `Shape`.
- Each subclass of `StackObject` stores the current shape in `Model`.

so it just creates a `LineObject` and pushes it onto the stack.

```
public class LineDrawingState {
  public void handleEvent(PointInputEvent event) {
    StackObject stackObject = DrawingContext.instance().getStackObject();
    if (stackObject == null) {
      stackObject = new LineObject(new Line(new Point2D(event.getX(), event.getY())));
      DrawingContext.instance().setStackObject(stackObject);
      UndoManager.getHistory().push(stackObject);
```

```
        View.instance().update();
    }
    // rest of the code in the method as before
}
```

9.5.1 Employing the Command Pattern

The reader may have noticed the a familiar pattern in the above code. In its `undo()` method, the `UndoManager` passes itself as a reference to the `undo()` method of the `StackObject`. In turn, each subclass of the `StackObject` (e.g., `LineObject`) passes itself as reference when invoking the appropriate undo method of the controller. This results in unnecessarily moving a lot of data around. One of the lasting lessons of the object-oriented experience is the supremacy of data over process, which can be tersely summed up as the following "Law of inversion[2]":

If your routines exchange too many data, put your routines in your data

In this context, this can be accomplished using the **Command Pattern.**

The intent of the Command pattern is as follows [3]: *Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undo-able operations.* We have partially satisfied this intent in our scenario by associating an object with each operation. For instance, whenever we execute an operation to create a line, a `LineObject` is created and pushed onto the stack. What we have failed to recognize so far is that this object need not merely be a repository of associated data but can also encapsulate the routines that manipulate this data.

The Command pattern provides us with a template to address this. The abstract `Command` class has abstract methods to `execute(), undo()` and `redo().` The concrete command classes (such as `LineCommand`) implement these methods and store the associated data needed to undo and redo these operations.

```
public abstract class Command{
  public abstract boolean undo();
  public abstract boolean redo();
  public abstract void execute();
}

public class LineCommand extends Command {
    private Line line;
    public LineCommand(Point2D point1) {
        line = new Line(point1);
        execute();
    }
    public void execute() {
        Model.instance().addShape(line);
    }
    public boolean undo() {
```

[2]see Bertrand Meyer, pg 684

[3]GoF, pg 233

```
        Model.instance().removeShape(line);
        return true;
    }
    public boolean redo() {
        execute();
        return true;
    }
    public void setPoint2(Point2D point) {
        line.setPoint2(point);
    }
}
```

As we saw, the class `UndoManager` that manages undo and redo. The `history` stack stores a list of the commands that have been executed. As a result, the `undo()` method simply gets the command from the stack and invokes its `undo()` method.

The class is a singleton, but in the following code, we omit the code related to the singleton implementation.

```
public class UndoManager {
    private Stack<Command> history;
    private Stack<Command> redoStack;
    // Some of the code for implementing the Singleton pattern is not shown
    private UndoManager() {
        history = new Stack<Command>();
        redoStack = new Stack<Command>();
    }
    public void beginCommand(Command command) {
        redoStack.clear();
        history.push(command);
    }
    public void endCommand() {
        Model.instance().updateView();
    }
    public void undo() {
        if (!(history.empty())) {
            Command command = (history.peek());
            if (command.undo()) {
                history.pop();
                redoStack.push(command);
            }
        }
    }
    public void redo() {
        if (!(redoStack.empty())) {
            Command command = (redoStack.peek());
            if (command.redo()) {
                redoStack.pop();
                history.push(command);
            }
        }
     }
}
```

The superclass of all states `DrawingState` ignores all undo and redo requests, so this behavior is inherited by all state classes. Undo and redo requests must be responded to when the system is quiescent, so this state overrides the `handleEvent()` methods as below.

```
    public void handleEvent(UndoRequestEvent event) {
        UndoManager.instance().undo();
    }
    public void handleEvent(RedoRequestEvent event) {
        UndoManager.instance().redo();
    }
```

The context class, `DrawingContext` now maintains the current `Command` as opposed to the current `Shape`. So the relevant changes are

```
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public Command getCommand() {
        return command;
    }
```

The states for drawing the shapes now deal with the command classes, as opposed to the `Shape` objects. Since adding the new shape to the model is done by the `Command` classes, that code is absent in the satte classes.

Here is the code for handling `PointInputEvent` in `DrawLineState`.

```
   public void handleEvent(PointInputEvent event) {
        Command command = DrawingContext.instance().getCommand();
        if (command == null) {
            command = new LineCommand(new Point2D(event.getX(), event.getY()));
            DrawingContext.instance().setCommand(command);
            UndoManager.instance().beginCommand(command);
            View.instance().update();
        } else {
            ((LineCommand) command).setPoint(1, new Point2D(event.getX(), event.getY()));
            View.instance().update();
            DrawingContext.instance().setCommand(null);
            DrawingContext.instance().changeCurrentState(QuiescentState.instance());
            UndoManager.instance().endCommand();
        }
   }
}
```

To abandon a command that is in progress, `DrawingState` code now uses `UndoManager`.

```
public void handleEvent(AbandonEvent event) {
    Command command = DrawingContext.instance().getCommand();
    command.undo();
    DrawingContext.instance().setCommand(null);
    DrawingContext.instance().changeCurrentState(QuiescentState.instance());
}
```

Index