MODULE 4

Interactive Systems and the MVC Architecture

11.1 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 behaviour, 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 are an immense number of possibilities; toward the end there are few 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**.

In this chapter, we start by describing a well-known software architecture (sometimes referred to as an **architectural pattern**) 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.

B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_11

11.2 The MVC Architectural Pattern

The model view controller is a relatively old pattern that was originally introduced in the Smalltalk programming language. As one might suspect, the pattern divides the application into three subsystems: model, view, and controller. The architecture is shown in Fig. 11.1. 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 (resulting in a low degree of cohesion), the MVC pattern helps produce highly cohesive modules with a low degree of coupling. This facilitates greater flexibility and reuse. MVC also provides a powerful way to organise systems that support multiple presentations of the same information.

The model, which is a relatively passive object, stores the data. Any object can play the role of model. The view renders the model into a specified format, typically something that is suitable for interaction with the end user. For instance, if the model stores information about bank accounts, a certain view may display only the number of accounts and the total of the account balances. The controller captures user input and when necessary, issues method calls on the model to modify the stored data. When the model changes, the view responds by appropriately modifying the display.

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. Both the controller and the view communicate with the user through the UI. This means that some components of the UI are used by the controller to receive input; others are used by the view to appropriately display the model and some can serve both purposes (e.g., a panel can display a figure and also accept points as input through mouseclicks). It is important to distinguish the UI from the rest of the system: beginners often mistake the UI for the view. This is easy error to make for two reasons. In most systems, due to the nature of the desired look and feel and the technologies available, there is a single window in which the entire

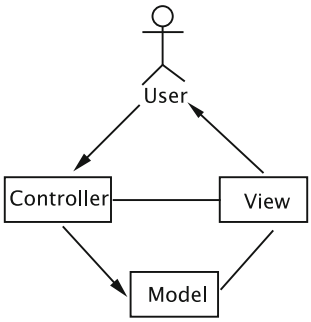


Fig. 11.1 The model–view–controller architecture

application is housed. This means that there has to be a common subsystem that provides the functionality needed both for the view and the user interface. The other source of potential confusion is that the UI presents to the user an image of how the system looks, and this can be mistakenly construed as the view. This interface must include components that are in fact part of the controller (e.g., buttons for giving commands). When we talk of MVC in the abstract sense, we are dealing with the architecture of the system that lies behind the UI; both the view and the controller are subsystems at the same level of abstraction that employ components of the UI to accomplish their tasks. From a practical standpoint, however, we have a situation where the view and the UI are contained in a common subsystem. *For the purpose of designing our system, we shall refer to this common subsystem as the view.* The view subsystem is therefore responsible for all the look and feel issues, whether they arise from a human–computer interaction perspective (e.g., kinds of buttons being used) or from issues relating to how we render the model. Figure 11.2 shows how we might present the MVC architecture while accounting for these practical considerations.

User-generated events may cause a controller to change the 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, consider 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.

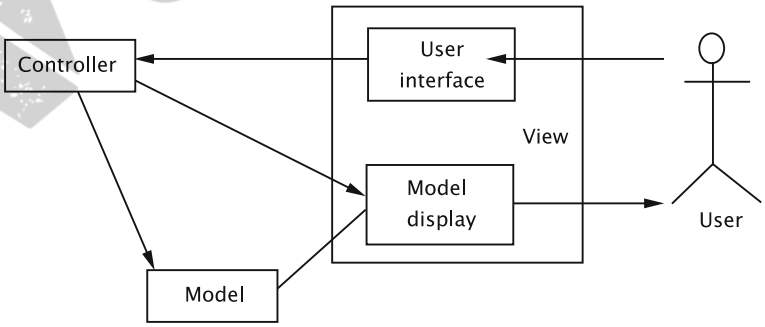


Fig. 11.2 An alternate view of the the MVC architecture

The view–model relationship is that of a subject–observer. The model, as the subject, maintains references to all of the views 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 guiding principle here is that each view is a faithful rendering of the model.*

11.2.1 Examples

Suppose that in the library system we have a GUI screen using which users can place holds on books. Another GUI screen allows a library staff member to add copies of books. Suppose that a user views the number of copies, number of holds on a book and is about to place a hold on the book. At the same time, a library staff member views the book record and adds a copy. Information from the same model (book) is now displayed in different formats in the two screens.

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.

11.2.2 Implementation

As with any software architecture, the designer needs to have a clear idea about how the responsibilities are to be shared between the subsystems. This task can be simplified if the role of each subsystem is clearly defined.

- The view is responsible for all the presentation issues.
- The model holds the application object.
- The controller takes care of the response strategy.

The definition for the model will be as follows:

```
public class Model extends Observable {
  // code
  public void changeData() {
    // code to update data
    setChanged();
    notifyObservers(changeInfo);
  }
}
```

Each of the views is an `Observer` and implements the `update` method.

```
public class View implements Observer {
  // code
  public void update(Observable model, Object data) {
    // refresh view using data
  }
}
```

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.

For each feature, we start with a detailed list of specifications, stated clearly enough so that they can be classified as belonging to one of the three categories. In general, there is always an initiation step for each operation; the manner in which the user is to be shown the feature and the manner in which it is invoked are part of the presentation. What the system should do when the request is made is a part of the response strategy, and the controller manages this part of the show. This strategy may involve interacting with the user in tandem with making changes to the application object. What is needed from the user is part of the response strategy, but how the system communicates with the user is a presentation issue. Changes to the application object are made by invoking the methods of model. As the application object is modified, the display needs to be modified to reflect the changes. Modifying the display is again a matter for presentation.

Clearly, there is a lot of entanglement here between the three parts, and it is a challenge to keep everything separate. The controller invokes the methods provided by the model so that the separation is relatively easy to implement. There can be confusion around drawing a line between the responsibilities of the view and the controller for reasons explained earlier. Likewise, keeping the business logic away from the display (or model–view separation) can be tricky in situations where there is a close relationship between the stored data and the methods for rendering it. As we design and implement a case-study in the following pages, we make decisions as various situations arise. Although the philosophy behind this architecture is easily stated, the details are best explained by example.

This means that it is not always possible to have a clean division of the components such that some components are designated for data input and the rest are for data display. Therefore, it is quite difficult to decide which components belong to the controller and which ones are part of the view. Surely, the view has to display data and, in general, some of its components end up as mechanisms for user input.

The approach we use to resolve this is to create a UI with functionality to serve the purpose of both the view and the controller. Display components will be available to the view, which invokes the appropriate display commands. Components which capture events generated by user inputs are configured to pass on the message to the appropriate subsystem; note that events for some operations (like scrolling) are handled by the view, whereas others (like add, delete) are sent to the controller.

11.2.3 Benefits of the MVC Pattern

1. Cohesive modules: Instead of putting unrelated code (display and data) in the same module, we separate the functionality so that each module is cohesive.
2. Flexibility: The model is unaware of the exact nature of the view or controller it is working with. It is simply an observable. This adds flexibility.
3. Low 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.
4. Adaptable modules: Components can be changed with less interference to the rest of the system.
5. Distributed systems: Since the modules are separated, it is possible that the three subsystems are geographically separated.

11.3 Analysing 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 subsystems. The subsystems are specified by the architecture. The designer gets to decide which classes to create for each subsystem, but the the responsibilities associated with these classes must be consistent with the purpose of the subsystem.
- *To understand how the MVC architecture is employed* We shall follow the architecture somewhat *strictly*, i.e., we will try to have three clearly delineated subsystems 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.

11.3.1 Specifying the Requirements

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

1. Draw lines and circles.
2. Place labels at various points on the figure; the labels are strings. A separate command allows the user to select the font and font size.
3. Save the completed figure to a file. We can open a file containing a figure and edit it.
4. Backtrack our drawing process by undoing recent operations.

Compared to the kinds of drawing programs we have on the market, this looks too trivial! Nonetheless, it is sufficient to show how the responsibilities can be divided so that the MVC pattern can be applied. What we shall also see, later on, is how new features can be added without disrupting the existing classes.

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, which are labeled like Draw Line, Draw Circle, Add Label, etc. The system will listen to mouse-clicks which will be employed by the user to specify points on the display panel.
- The display panel will have a cross-hair cursor for specifying points and a_ (underscore) for showing the character insertion point for labels. The default cursor will be an arrow.
- The cursor changes when an operation is selected from the command menu. When an operation is completed, the cursor goes back to the default state.
- To draw a line, the user will specify the end points of the line with mouse-clicks. To draw a circle, the user will specify two diametrically opposite points on the perimeter. For convenient reference, the center of each circle will be marked with a black square. To create a label, the starting point will be specified by a mouse-click.

11.3.2 Defining the Use Cases

We can now write the detailed use cases for each operation. The first one, for drawing a line, is shown in Table 11.1.

Table 11.1 Use-case table for Drawing a line

Actions performed by the actor	Responses from the system
1. The user clicks on the Draw Line button in the command panel	
	2. The system changes the cursor to a cross-hair
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 to the figure being created. The cursor changes to the default

Table 11.2 Use-case table for Adding a Label

Actions performed by the actor	Responses from the system
1. The user clicks on the Add Label button in the command panel	
	2. The system changes the cursor to a cross-hair cursor
3. The user clicks at the left end point of the intended label	
	4. The system places a_ at the clicked location
	5. The system waits for the user response
5. The user types a character or clicks the mouse at another location	
	6. If the character is not a carriage return the system displays the typed character followed by a_, and the user continues with Step 5; in case of a mouse-click, it goes to Step 4; otherwise it goes to the default state

The use case for drawing a circle can be done analogously.

To give the system better usability, we allow for multiple labels to be added with the same command. To start the process of adding labels, the user clicks on the command 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 type a carriage return, which returns the system to the default state. These details are spelled out in the use case in Table 11.2.

The system will ignore almost all non-printable characters. The exceptions are the Enter (terminate the operation) and Backspace (delete the most-recently entered character) keys. A label may contain zero or more characters.

We also have use cases for operations that do not change the displayed object. An example of this would be when the user changes the font, shown in Table 11.3.

The requirements call for the ability to save the drawing and open and edit the saved drawings. The use cases for saving, closing and opening files are left as exercises. In order to allow for editing we need at least the following two basic operations: *selection* and *deletion*. The use case Select an Item is detailed in Table 11.4.

There are some details here that need to be fleshed out in later stages. We have not specified how the system would indicate the change to the *selection mode*. We could do this by changing the cursor or altering the display in some other way. This use case requires that the display should indicate which items have been selected. This can be done by drawing these items in a different colour.

It is possible that the user's click does not fall on any item; in that case, the system simply ignores the mouseclick and returns to the default mode.

Table 11.3 Use-case table for Change Font

Actions performed by the actor	Responses from the system
1. The user clicks on the Change Font button in the command panel	
	2. The system displays a list of all the fonts available
3. The user clicks on the desired font	
	4. The system changes to the specified font and displays a message to that effect

Table 11.4 Use-case table for Select an Item

Actions performed by the actor	Responses from the system
1. The user clicks on the Select button in the command panel	
	2. The system changes the display to the *selection mode*
3. The user clicks the mouse on the drawing	
	4. If the click falls on an item, the system adds the item to its collection of selected items and updates the display to reflect the addition. The system returns the display to the default mode

Deletion will be done by having a button in the GUI that the user can click; whenever this button is clicked, all the selected items are deleted. The use case for this is left as an exercise.

The reader would note that this system is 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.

11.4 Designing the System

The process of designing this system is somewhat different from our earlier case studies owing to the fact that we have selected an architecture. Our architecture specifies three principal subsystems, viz., the Model, the View and the Controller. We have a broad idea of what roles each of these play, and our first step is to define these roles in the context of our problem. As we do this, we look at the individual use cases and decide how the responsibilities are divided across the three subsystems. Once this is taken care of, we look into the details of designing each of the subsystems.

11.4.1 Defining the Model

Our next step is to define what kind of an object we are creating. This is relatively simple for our problem; we keep a collection of line, circle, and label objects. Each line is represented by the end points, and each circle is represented by the *X*-coordinates of the leftmost and rightmost points and the *Y*-coordinates of the top and bottom points on the perimeter (see Fig. 11.3).

For a label, the model stores the coordinate's starting position, the text, and the style and size of the characters in the string. The collection is accessed by the view when the figure is to be rendered on the screen. The model also provides mechanisms to access and modify its collection objects. These would be methods like `addItem(Item)`, `getItems()`, etc.

11.4.2 Defining the Controller

The controller is the subsystem that orchestrates the whole show and the definition of its role is thus critical. When the user attempts to execute an operation, the input is received by the view. The view then communicates this to the controller. This communication can be effected by invoking the public methods of the controller. Let us examine in detail the various implementation steps for the processes described in the use cases.

Drawing a Line

1. The user starts by clicking the Draw line button, and in response, the system changes the cursor. Clearly, changing the cursor should be a responsibility of the view, since that is where we define the look and feel. This would imply that the view system (or some part thereof) listen to the button click. The click indicates that the user has initiated an operation that would change the model. Since such operations have to be orchestrated through the controller, it is appropriate that the

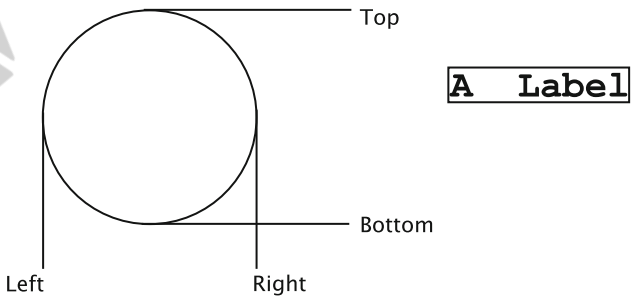


Fig. 11.3 Representing a circle and a label

controller be informed. The controller creates a line object (with both endpoints unspecified).

2. The user clicks on the display panel to indicate the first end point of the line. We now need to designate a listener for the mouse clicks. This listener will extract the coordinates from the event and take the necessary action. Both the view and the controller are aware of the fact that a line drawing operation has been initiated. The question then is, which of these subsystems should be responding to the mouse-click? Having the controller listen directly to the mouse-clicks seems to be more efficient, since that will reduce the number of method invocations. However there are several reasons why this is not a good choice. First, the methods/interfaces (e.g., `MouseListener` in Java) to be implemented depend on the manner in which the view is being implemented. This means that the *controller is not independent of the view*, thus hurting reuse. A second reason is that we can have multiple ways to input the points. For instance, when trying to draw a precise figure, a user may prefer to specify the points as coordinates through some kind of dialog, instead of clicking the mouse. *These accommodations are part of the look and feel, and do not belong in the controller.* Finally, we have the problem of *reading and interpreting the input.* In our particular situation, this manifests itself as the process of mapping device coordinates to the image coordinates. Most of the graphical display tools available nowadays use a coordinate system where the origin corresponds to the top-left corner of the display rectangle, with X coordinates increasing from left to right and Y coordinates increasing from top to bottom (also known as *device coordinates*). Programs that generate and use graphics often prefer the standard Cartesian coordinate system. Thus we might have a situation where the model is being created with Cartesian coordinates, whereas mouse clicks and graphical output must use device coordinates and points have to be mapped from one system to the other. The conversion of Cartesian coordinates to device coordinates is best done in the view since it knows and is responsible for the *nature and format of the output* (points specified as device coordinates). The reverse operation of converting device coordinates of input points to Cartesian coordinates must also, therefore, be done by the view, which means that the view must capture the input. Therefore, although a performance penalty is incurred, we favour the implementation where the mouse-click is listened to in the view. The view then communicates these coordinates to the controller, after performing any transformation or mapping that may be needed. At this point we need to decide how the system would behave during the period between the clicks. For instance, should the point for the first click be highlighted in any way? Since the use case does not specify anything, we can ignore this issue for the time being, i.e., no change happens until both end points are clicked.
3. The user clicks on the second point. Once again, the view listens to the click and communicates this to the controller. On receiving these coordinates, the controller recognises that the line drawing is complete and updates the line object.
4. Finally, the model notifies the view that it has changed. The view then redraws the display panel to show the modified figure.

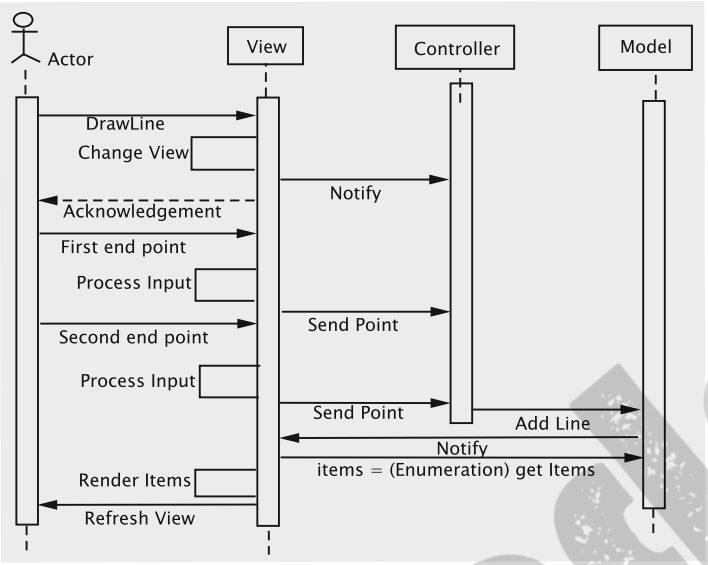


Fig. 11.4 Sequence of operations for drawing a line

This sequence of operations across the three subsystems can be captured by a high-level sequence diagram as shown in Fig. 11.4. Note that unlike the sequence diagrams in earlier chapters, this does not spell out all the classes involved or the names of the methods invoked.

Drawing a Circle

The actions for drawing a circle are similar. However, we now have some additional processing to be done, i.e., the given points on the diameter must be converted to the the four integer values, as explained in Fig. 11.3. Note that this requires a mapping to convert the input to the form required by the model. This can be performed in the controller, since these representations are equivalent.

Adding a Label

This operation is somewhat different due to the fact that the amount of data is not fixed. The steps are as follows:

1. The user starts by clicking the Add Label button. In response, the system changes the mouse-cursor, which, as before is the responsibility of the view.
2. The user clicks the mouse, and the system acknowledges the receipt of the mouse click by placing a_ at the location. This would result in changing what the drawing looks like. As decided earlier, we will maintain the property that the view is a faithful rendering of the model. The view therefore notifies the controller that the operation has been initiated, and the controller modifies the model. One issue

that we have to resolve is that of assigning the appropriate size and style to the characters in the label. To implement this, we have to address the following:

- *Which subsystem 'remembers' the current style and size?* Since the user cannot be expected to specify the size and style with each character, these have to be stored somewhere. For our situation, we shall assume that these are stored in the view and passed on to the controller when the label construction operation is initiated.
- *When do the changes to size and style take effect?* To simplify our system, we assume that these will take effect for the next label that is created. What this means is that the style and size have to be uniform for any given label, and if a change is made to any of these while we are in the process of creating a label, these changes will not take immediate effect.

3. The user types in a character. Once again, the view listens to and gets the input from the keyboard, which is communicated to the controller. Once again the controller changes the model, which notifies the view.
4. The user clicks the mouse or enters a carriage-return. This is appropriately interpreted by the view. In both cases, the view informs the controller that the addition of the label is complete. In case of a mouse click, the controller is also notified that a new operation for adding a label has been initiated.

This sequence of steps is explained in Fig. 11.5. *Note that the view interprets the keystrokes: as per our specifications ordinary text is passed on directly to the controller, control characters are ignored, carriage-return is translated into a command, etc. All this is part of the way in which the system interacts with the user, and therefore belongs to the view.*

Dealing with the Environmental Variables

Most interactive systems need to remember the values of certain parameters to make the system user-friendly. For instance, a word-processing system remembers the size and font of the characters so that the user does not have to specify these with every operation. We refer to these parameters as **environmental** variables. In our example, for creating a label, we choose to store these in the view, and this has some consequences for the behaviour of the system.

Consider a document creation system that has Times-Roman as the default font. When the system starts up, the font parameter stores the value 'Times-Roman'. If the user selects a different font, say Helvetica, then this parameter is changed and any following text input is displayed in Helvetica. The font parameter could be stored in the model or in the view. If we store this in the model, then the font information does not have to be sent by the view to the controller, along with the text. In addition, this would result in storing the font parameter when the figure is saved to a file.

Now consider what happens when the file is retrieved at some later time. The font parameter would be set to Helvetica, and this font would apply to all the text input. On the other hand, if the font type is stored in the view, storing and retrieving would set the font back to Times-Roman (the default). Clearly, this is a choice that has to be made when the behaviour of the system is being decided.

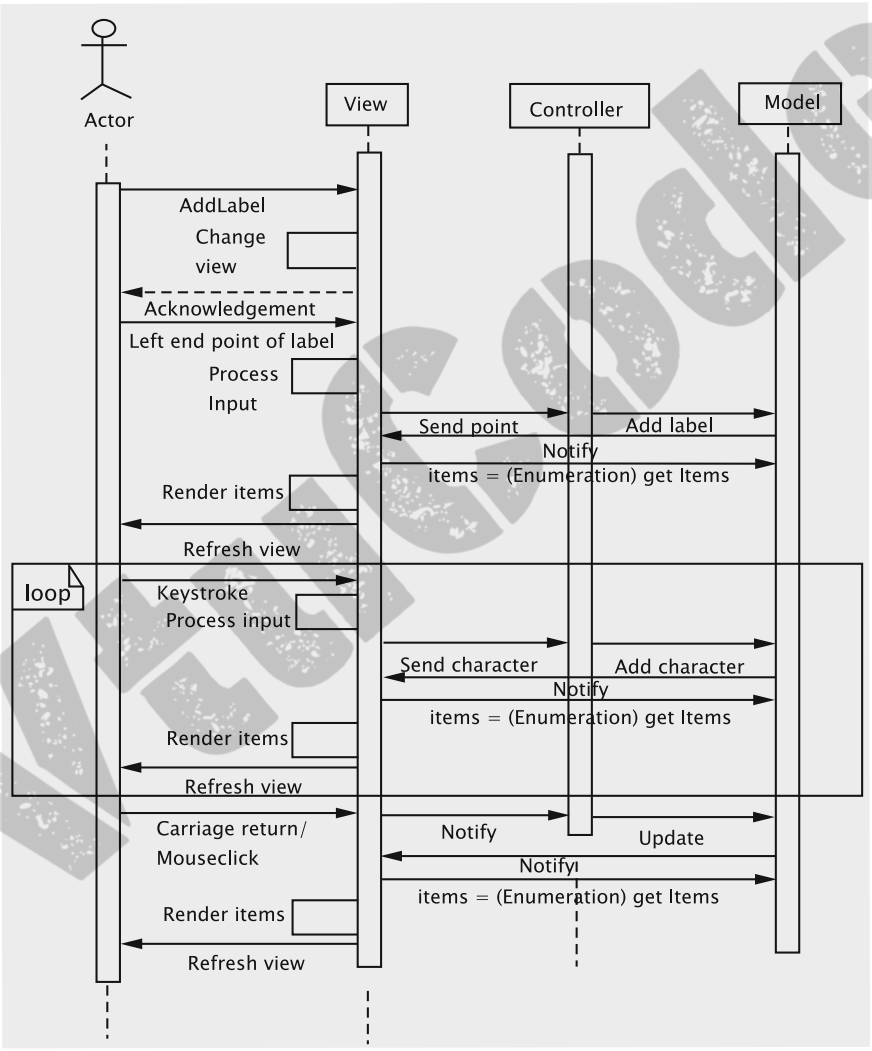


Fig. 11.5 Sequence of operations for adding a label

Sharing Responsibilities Between the View and the Controller

When we employ the MVC architecture, there is often a gray area between the responsibilities of the controller and those of the view, particularly for the kind of software discussed in this case-study. Issues that fall in this area can be confusing to the beginner, particularly since widely varying opinions have been expressed. Some of these issues have come up in this section and need clarification.

Accepting user input In our approach above, all user input is received by the view. Indeed, the view is the only mechanism through which the user can interact and the view parses all the input that comes in. The idea here is that the system as a whole be 'UI agnostic', i.e., the design of the system does not depend on how the UI has been implemented.

Consider the situation where the user gives a command. This is done by a button click. It is tempting to let the controller, or one of its components, listen to the click and take action. However, this creates problems if the UI is changed so that the same commands can instead be given by keystrokes. In such a situation, a change in the UI, or even in the look and feel, can force changes in the controller. In addition, there could be situations where the same operation can be initiated in multiple ways. If the controller has to accommodate all of these, it adds to the complexity of the controller and causes tight coupling.

Once an operation has been initiated, we have the issue of accepting the data. Once again, while some designers have argued that the data be received in the controller, this approach is fraught with problems. The data could be in one of several formats. For instance, a UI designer might want to accommodate for users to type in coordinate locations instead of clicking with the mouse. (This could be important for drawing precise geometric figures.) Having the controller deal with multiple formats is not desirable. A second, more serious issue is that when the data needs some 'correction' to adjust for the display. For instance, consider a situation where the figure is being drawn with *Cartesian coordinates* due to the nature of the application. The mouse-click specifies the value in coordinates with reference to the object that is being used for the display (in Java, this would be the `JPanel`, or a `JScrollPane`), which will have to be mapped to the Cartesian values. Doing this mapping in the controller would mean exposing the controller to all the details of the components used by the view. The important thing to keep in mind is that the view is providing the user with several input mechanisms, and therefore should be responsible for receiving and interpreting the data. *The task of accepting and standardising user input is therefore the responsibility of the view.*

Processing and Storing the input Once the standardised data is available, it is incorporated into the model. All data is received in conjunction with some operation, and hence the details of how the data is to be used to change the model are part of the operation. This activity is independent of the UI, implying that this would be the responsibility of the controller.

11.4.3 Selection and Deletion

The software allows us to delete lines, circles, or labels by selecting the item and then invoking the `delete` operation. These shall be treated as independent operations since selection can also serve other purposes. Also, we can invoke selection repeatedly so that multiple items can be selected at any given time.

When an item is selected, it is displayed in red, as opposed to black. The selection is done by clicking with the arrow (default) cursor. Lines are selected by clicking on one end point, circles are selected by clicking on the center, and labels are selected by clicking on the label.

The steps involved in implementing this are as follows:

1. The user gives the command through a button click. This is followed by a mouse click to specify the item. Both of these are detected in the view and communicated to the controller.
2. In order to decide what action the controller must take, we need to figure out how the system will keep track of the selected items. Since the view is responsible for how these will be displayed (in red, for instance) the view must be able to recognise these as selected when updating the display. Since the view gets the items from the model, it would seem appropriate that the model have a mechanism to flag the selected items. This can be done by having a tag field for each item, or simply by moving the selected items to a separate container. We shall use the latter.
3. The next step is to iterate through the (unselected) items in the model to find the item (if any) that contains the point. Since the model is to be used strictly as a repository for the data, the task of iterating through the items is done in the controller, which then invokes the methods of the model to mark the item as selected.
4. Model notifies view, which renders the unselected items in the default colour (black) and the selected items in red. View gets an enumeration of the two lists separately and uses the appropriate colour for each. Note that model only stores a separate list of the selected items. It is the view that decides how the two lists are to be rendered.

Deletion is a simpler operation. The button click is heard in the view and passed on to the controller, which simply requests the model to delete all selected items.

11.4.4 Saving and Retrieving the Drawing

The use cases for the processes of saving and retrieving are simply described: *the user requests a save/retrieve operation, the system asks for a file name which the user provides and the system completes the task.* This activity can be partitioned between our subsystems as follows:

1. The view receives the initial request from the user and then prompts the user to input a file name.
2. The view then invokes the appropriate method of the controller, passing the file name as a parameter.
3. The controller first takes care of any clean-up operation that may be required. For instance, if our specifications require that all items be unselected before the drawing is saved, or some default values of environment variables be restored, this must be done at the stage. The controller then invokes the appropriate method in the model, passing the file name as a parameter.
4. The model serializes the relevant objects to the specified file.

This completes the first step of distributing the responsibilities across the three subsystems. Note that unlike the earlier case studies, we did not look for classes and methods and try to create a class interaction diagram right away. This would be fairly typical when we are designing a larger software system with some advance notice about the kind of architecture being employed. As we progress through the details, we might also realise that our partitioning of responsibilities across the subsystems may have to shift a little due to other considerations. This is not unusual, since the architecture only gives us broad guidelines, and not a detailed design.

11.5 Design of the Subsystems

The next step of the process is to design the individual subsystems. In this stage, the classes and their responsibilities are identified and we get a more detailed picture of how the required functionality is to be achieved. Since the model should remain independent of the 'look-and-feel' of the system and should remain stable, it is appropriate that we design it first.

11.5.1 Design of the Model Subsystem

Consider the basic structure of the model and the items stored therein. From Sect. 11.3, we know that the model should have methods for supporting the following operations:

1. Adding an item
2. Removing an item
3. Marking an item as selected
4. Unselecting an item
5. Getting an enumeration of selected items
6. Getting an enumeration of unselected items
7. Deleting selected items
8. Saving the drawing
9. Retrieving the drawing

Based on the above list, it is straightforward to identify the methods. The class diagram is shown in Fig. 11.6. The class `Item` represents a shape such as line or label and enables uniform treatment of all shapes within a drawing.

Since the methods, `getItems()` and `getSelectedItems()` return an enumeration of a set of items, we need polymorphic containers in the model. The view uses these methods to get the objects from the model as an enumeration of the items

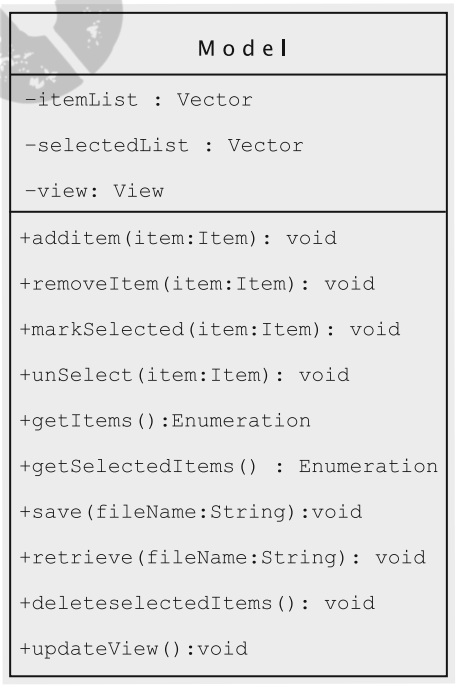


Fig. 11.6 Class diagram for model

stored and draws each one on the display panel. The model must also keep track of the view, so it needs a field for that purpose.

The method `updateView` is used by the controller to alert the model that the display must be refreshed. It is also invoked by methods within the model whenever the model realises that its data has changed. This method invokes a method in the view to refresh the display.

11.5.2 Design of Item and Its Subclasses

Clearly, `Item` will have several subclasses, one for each shape. Each subclass will store attributes that are relevant to the corresponding shape.

Rendering the items A tricky issue regarding the design is how the items should be rendered. Rendering 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 UI* For instance, we are using the Java's Swing package, which means that our drawing panel is a `JPanel` and the drawing methods will have to be invoked on the associated `Graphics` object.
- *The item that is stored* 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 view, whereas the structure of the item is known to the author of the items. Since the needed information is in two different 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 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 item were responsible, each item would have a `render` method that accesses the fields and draws the item. The problem with this is that the way an object is to be rendered often depends on the tools that we have at our disposal. For instance, consider the problem of rendering a circle: 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. In addition to the set of tools, there are other specific features that the technology has. Using the Swing package in Java, for instance, implies that all the drawing is done by invoking the methods on the `Graphics` object associated with the drawing panel.

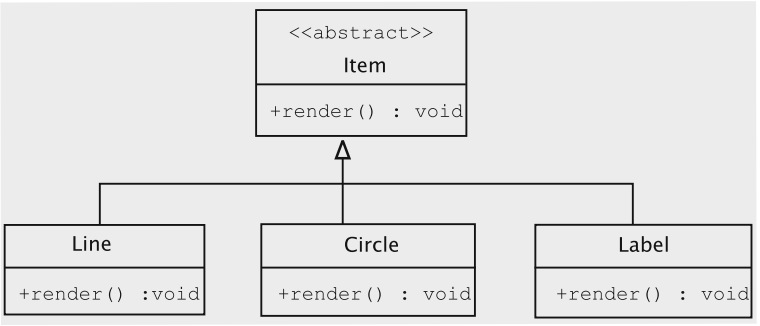


Fig. 11.7 The item class and its subclasses

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: *we are querying each object in the collection to apply the right methods.* This is very much at odds with the object-oriented philosophy, i.e., *the methods should be packed with the data that is being queried.* This really means that the `render` method for each item should be stored in the item itself, which is in fact the approach of the second option. This simplifies our task somewhat, so we can focus on the task of fixing the shortcomings of the second option.

The structure of the abstract `Item` class and its subclasses are shown in Fig. 11.7.

Catering to Multiple UI Technologies

Swing is just one package for drawing. Before it was developed, there was (and still is) the AWT (Abstract Windowing Toolkit) package available to Java programmers, and it is conceivable that there may appear some other drawing toolkits. Let us assume that we have available two new toolkits, which are called, for want of better names, `HardUI` and `EasyUI`. Essentially, what we want is that each item has to be customised for each kind of UI, which boils down to the task of having a different `render` method for each UI. One way to accomplish this is to use inheritance.

To adapt the design to take care of the new situation, we have the `Circle` class implement most of the functionality for circle, except those that depend on the UI technology. We extend `Circle` to implement the `SwingCircle` class. Similar extensions are now needed for handling the new technologies, `HardUI` and `EasyUI`. Each of the three classes has code to draw a circle using the appropriate UI technology. The idea is shown in Fig. 11.8.

In each case, the `render` method will decompose the circle into smaller components as needed, and invoke the methods available in the UI to render each component. In addition, each method would have to get any other contextual information. For instance, with the Swing package, the `render` method would get the graphics object from the view and invoke the `drawOval` method. The code for this could look something like this:

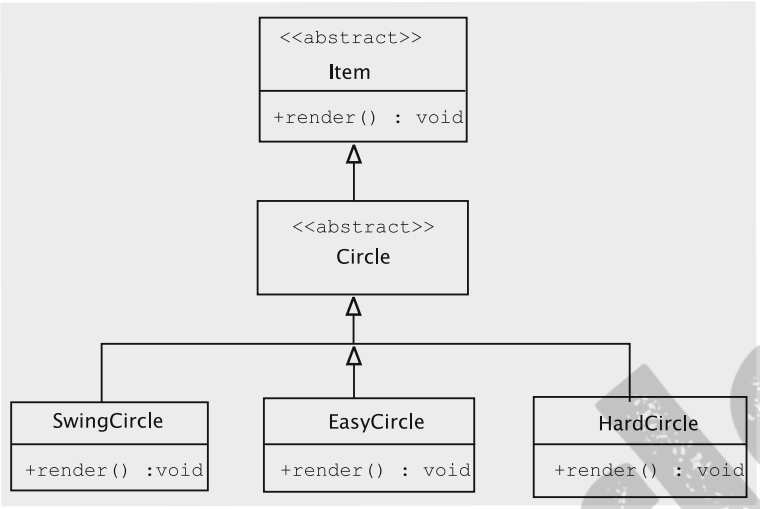


Fig. 11.8 Catering to multiple UI technologies

```
public class SwingCircle extends Circle {
  // circle class for SwingUI
  public void render() {
    Graphics g = (View.getInstance()).getGraphics();
    g.drawOval(/* parameters */);
  }
}
```

The actual parameters for `drawOval` would depend on any mapping needed, but would be computed using quantities stored in the `Circle` object. In addition to the `Graphics` object, we may need several other pieces of information from the context, such as the size of the drawing area, etc. The model could potentially employ several types of items, each of which has a corresponding abstract class.

Clearly, we need abstract classes for implementing the technology-independent parts of lines (`Line`) and labels (`Label`). They are extended by classes such as `SwingLabel`, `SwingLine`, `EasyLabel`, etc. This extension adds another six classes. Each abstract class ends up with as many subclasses as the number of UIs that we have to accommodate.

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

$$\textit{Number of types of items} \times \textit{Number of UI packages}$$

As is evident from the pictorial view of the resulting hierarchy (see Fig. 11.9), this causes an unacceptable explosion in the number of classes.

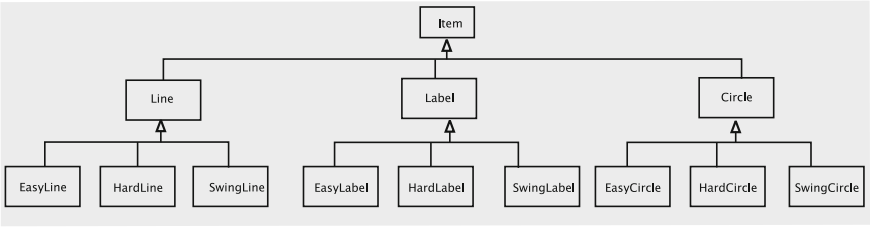


Fig. 11.9 Class explosion due to multiple UI implementations

This causes an unacceptable explosion in the number of classes.

Next, consider the situation where items 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 UI package that we are using.

A third and more subtle point is that of software upgrades. Suppose we create a version of our drawing program that supports the HardUI package and we use that to create a figure. All the items created in the model will belong to the HardUI subclasses, and can be used only with a system where the HardUI package is available. If a later version of the software does not support HardUI (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 UI, this problem could be avoided.

Can all these problems be circumvented? What we have here are two subsystems viz., the model and the view, each of which has its own classification viz., the types of items and the types of UIs. We are creating objects that account for both of these variations. Since the `Item` subclasses are being created in the model, the types of items are an *internal variation*. On the other hand, the subclasses of `Circle`, `Line`, and `Label` (such as `HardCircle`) are an *external variation*. The standard approach for this is to factor out the external variations and keep them as a separate hierarchy, and then set up a *bridge* between the two hierarchies. This standard approach is therefore called the **bridge pattern.**

We already have the hierarchy that captures the variation in the items. We need a second hierarchy to capture the variation in the drawing methods, due to the variation in the UIs. The hierarchy of the UIs has an interface `UIContext` and as many concrete implementations as the number of different UIs we need. Figure 11.10 describes the interaction diagram between the classes and visually represents the bridge between the two hierarchies.

Since the only variation introduced in the items due to the different UIs is the manner in which the items were drawn, this behaviour is captured in the `UIContext` interface as shown in Fig. 11.11.

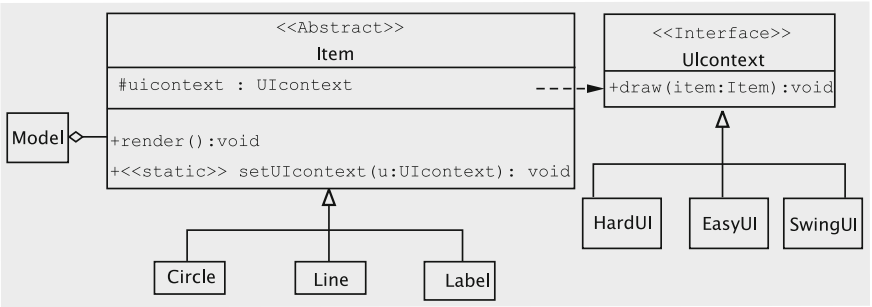
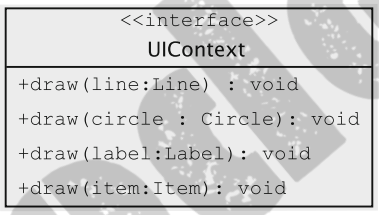


Fig. 11.10 Interaction diagram for the bridge pattern

Fig. 11.11 UIContext interface



Using the Bridge Pattern

The intent of the bridge pattern is as follows: *Decouple an abstraction from its implementation so that two can vary independently.* In our example, the abstraction is the abstract class `Item`. The `render` method of this abstraction has different implementations for different UIs. Using inheritance to allow for the different implementations has the following drawbacks:

- The abstractions and implementations cannot be modified and reused independently.
- If the variations in the implementation are introduced from two independent sources, keeping them in the same hierarchy could have a multiplicative effect on the number of concrete classes.

The bridge pattern takes care of these problems avoiding a permanent binding between the two. This gives our design the following desirable properties:

- Both abstraction and implementation are independently extensible (UIContext and Items change independently).
- Changes in the implementation do not affect the clients (if the SwingUI is changed, no other class is affected).

- Allows the implementation to be completely hidden from clients (in our case-study, the controller does not know anything about how the variations in the rendering come into play).
- Reduces the number of classes.
- Multiple classes can share the same representation. (Recall our discussion of how going to a new versions can make old documents unusable.)

One of the guiding principles of object-oriented design[1] states:
Favour object composition over class inheritance.

This principle is usually applied in the context that object composition allows us to achieve reuse by assembling existing components and get the needed functionality. The effectiveness of the bridge pattern can also be related back to this idea. If several aspects of the implementation (each of which is represented by some abstract method) of an abstraction have to be varied independently, the abstraction itself can be viewed as a composition of all the aspects of the implementation. The bridge pattern says that in such a case, having the abstract class as a composition of hierarchies that represent each of these aspects of the implementation is a definite improvement over relying entirely on inheritance. The flip side of this pattern, as often happens with applying object oriented principles, is that we lose some performance due to the indirection.

Note that the total number of classes is now reduced to

$$\textit{Number of types of items} + \textit{Number of UI packages}$$

Since we have only one concrete class for each item, the creation process is simple. Finally, by factoring out the `render` method, we are no longer concerned with what kind of UI is being used to create the figure, or what UI 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 the 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 invoke the `render` method, we have to get the model and the UIContext, in addition to invoking the drawing method.

Reflecting on the design The `UIContext` interface has a separate method for drawing each of the shapes, thereby establishing a one-to-one mapping with the shapes (circle, line, label). In general, such a one-to-one mapping is neither necessary nor realistic. Assume that we want to start supporting a new shape, say `Triangle`, with the obvious semantics, in our drawing program. This is clearly an example of a change that one should expect in a drawing program and, within reason, it should

[1] GoF, p. 127

impact as few interfaces and classes as possible. The class `Triangle` can then be written as below.

```
public class Triangle extends Item {
  private Line line1;
  private Line line2;
  private Line line3;
  // Fields, constructor, and other methods
  public  void render() {
    uiContext.draw(line1);
    uiContext.draw(line2);
    uiContext.draw(line3);
  }
}
```

Similarly, we could support arbitrary polygons.

This demonstrates a couple of things. For one, it justifies the use of the bridge pattern in our design. We are varying the `Item` hierarchy while requiring no changes at all to the `UIContext` hierarchy. In addition, it shows that the methods of `UIContext` can be quite 'general purpose' and not tied exclusively to one specific shape.

Suppose we restrict `UIContext` to the following:

```
public interface UIContext {
  public void draw(Point point1, Point point2); // for Line
  public void draw(String string, RenderInformation information);
                                                  // for Label
}
```

As the reader might guess, `draw` with the two `Point` parameters renders a line connecting the given points. The other `draw` method draws a sequence of characters with information such as the font and font size specified in an as yet unimplemented class named `RenderInformation`. Clearly, the `Line` class's `render` method can call the first `draw` method of `UIContext` and the label can be drawn by calling the second `draw` method. We do not require any additional functionality, since any shape can be drawn by decomposing it into a large number of lines.[2] Since there is no method to draw a circle, the `Circle` class must repeatedly invoke the first `draw` method to render the circle.

Employing option 1 Assume that rather than assigning the responsibility of drawing an `Item` object to the object itself, we have the view draw all the items. This could be accomplished by having methods such as `draw(Line line)` and `draw(Circle circle)` in the view subsystem. Every view will potentially have a different implementation of these methods. To render the items, a reference to the current view is obtained and the appropriate `draw` method is then called on that object.

While the methods that result from employing Option 1 are essentially the same as we get using the bridge pattern, there is a difference in that the bridge pattern employs a different class for each UI technology whereas Option 1 employs a set of `draw` methods for each view.

[2]This is in fact exactly how most curve drawing algorithms are implemented.

11.5.3 Design of the Controller Subsystem

Unlike the view, which by definition could be implemented in multiple ways, we structure the controller so that it is not tied to a specific view and is unique to the drawing program.

The view receives details of a shape (type, location, content, etc.) via mouse clicks and key strokes. As it receives the input, the view communicates that to the controller through method calls. This is accomplished by having the fields for the following purposes.

1. For remembering the model;
2. To store the current line, label, or circle being created. Since we have three shapes, this would mean having three fields.[3]

When the view receives a button click to create a line, it calls the controller method `makeLine`. To reduce coupling between the controller and the view, we should allow the view to invoke this method at any time: before receiving any points, after receiving the first point, or after receiving both points. For this, the controller has three versions of the `makeLine` method and keeps track of the number of points independently of the view.

The execution of `makeLine` causes the line to be part of the model. The view can set the endpoints of the line via the `setLinePoint` method.

The approach to add a label is similar to the one for adding a line. For a label, remember that by pressing the backspace the user can delete a character, so we provide a method `removeCharacter` for this purpose.

The controller also supplies a method (`selectItem`) that the view can call when it receives the command to select an item. The controller searches through the entire list of unselected items and determines if one of them is selected, and if so, it moves the item from the list of unselected items to the list of selected items.

The rest of the methods are for deleting selected items and for storing and retrieving the drawing and are fairly obvious. The class diagram is shown in Fig. 11.12.

To implement the saving and retrieval of files, the only objects to be serialized are the list(s) of the `Item` objects, which is a straightforward process. However, one of our stated goals is that of allowing a file to be retrievable even if the software has been modified so that we have a different version of the view, or if new features are added. This means that in the new version of the software the concrete `UIContext` may be different from the one that was used to create the items in the serialized list. One solution to this could be to set `uiContext` to `null` in all the objects being stored to disk and then reset these when the objects are read from disc. This solution is inelegant and some what worrisome in that the objects are being modified when saved and retrieved.

This is a reason why we have made `Item` an abstract class (instead of an interface). This enables us to store `UIContext` as a `static` field in this class, along with the

[3] We leave the circle implementation as an exercise, so we end up having only two fields in our design.

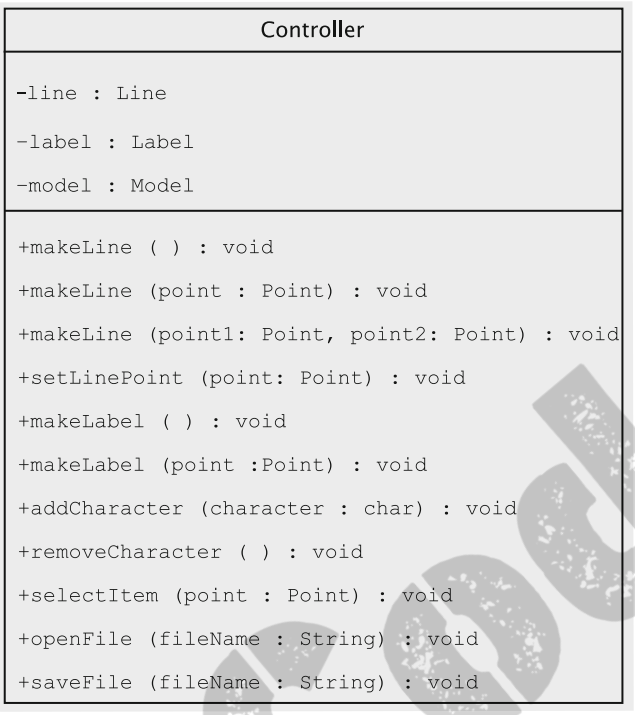


Fig. 11.12 Controller class diagram

static method `setUIContext` to modify it. The `UIContext` object is thus not a part of the object that is saved. This is consistent with the basic idea of the Bridge pattern, which calls for separation between the items and the manner in which they are rendered.

11.5.4 Design of the View Subsystem

The separation of concerns inherent in the MVC pattern makes the view largely independent of the other subsystems. Nonetheless, its design is affected by the controller and the model in two important ways:

1. Whenever the model changes, the view must refresh the display, for which the view must provide a mechanism.
2. The view employs a specific technology for constructing the UI. The corresponding implementation of `UIContext` must be made available to `Item`.

The first requirement is easily met by making the view implement the `Observer` interface; the `update` method in the `View` class, shown in the class diagram in Fig. 11.13, can be invoked for this purpose.

The issue regarding `UIContext` needs more consideration. The view consists of a drawing panel, which extends `JPanel` and needs to be updated using the appropriate instance of `UIContext`. A major question that arises is as to how and when this variable is to be set in `Item`. This can be achieved by having a public method, say `setUIContext`, in the model that in turn invokes the `setUIContext` on `Item`.

However, the time when we have to ensure that we are using the right instance of `UIContext` is just before a drawing is rendered by the view. Also, it is the view that knows which specific instance of `UIContext` is to be used in conjunction with itself. A logical way of doing this, therefore, would be to keep track of the appropriate `UIContext` in the view and invoke the `setUIContext` method in the model just before refreshing the panel that displays the drawing. In the Swing package, repainting is effected in the `paintComponent` method.

With multiple views, invoking the `setUIContext` method is problematic. Consider: more than one view might have scheduled repainting the screen, which would cause all of them to be executing `paintComponent` (or similar drawing method). If one of the views updates the `UIContext` field in the model while another is in the middle of painting the screen, chaos would result. This can be overcome by viewing the repainting code as a *critical section*. For details, please see Sect. 11.11.5.

Accepting input We have already decided that the user will issue commands by clicking on buttons. In the current implementation, we will assume that coordinate information (endpoints of lines, starting point of labels, etc.) will be specified by

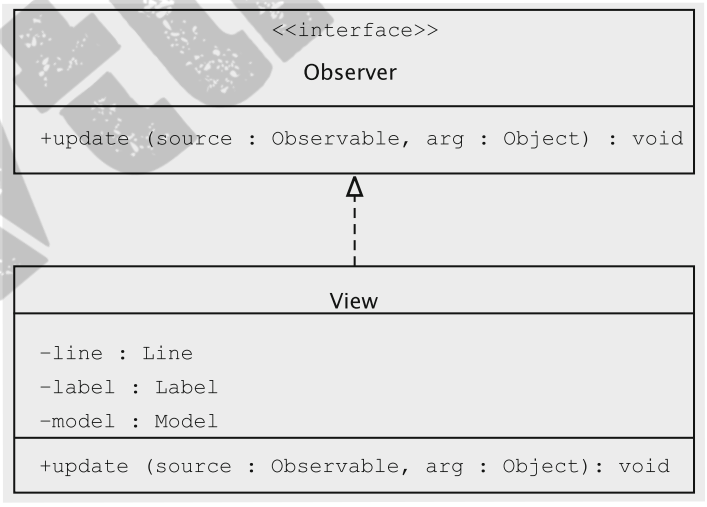


Fig. 11.13 Basic structure of the view class

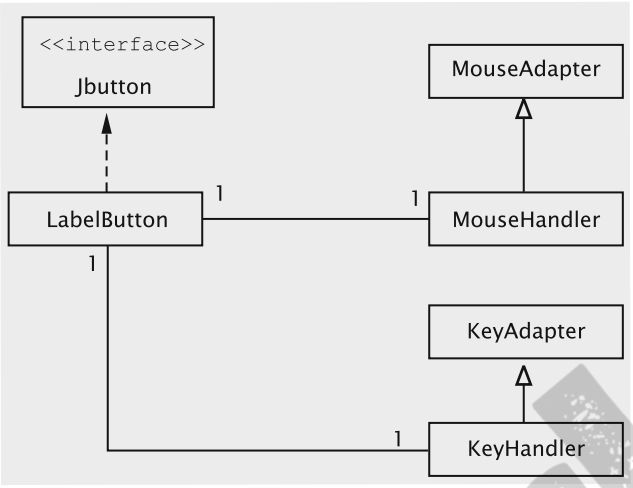


Fig. 11.14 Organisation of the classes to add labels

clicking on the panel. To catch these clicks, we need a class that acts as a mouse listener, which in Java demands the implementation of the `MouseListener`[4] interface.

Commands to create labels, circles, and lines all require mouse listeners. Since the behaviour of the mouse listener is dependent on the command, we know from previous examples in the book that a truly object-oriented design warrants a separate class for capturing the mouse clicks for each command. Since there is a one-to-one correspondence between the mouse listeners and the drawing commands, we have the following structure:

1. For each drawing command, we create a separate class that extends `JButton`. For creating labels, for instance, we have a class called `LabelButton`. Every button is its own listener.
2. For each class in (1) above, we create a mouse listener. These listeners invoke methods in the controller to initiate operations.
3. Each mouse listener (in (2) above) is declared as an inner class of the corresponding button class. This is because the different mouse listeners are independent and need not be known to each other.

The idea is captured in Fig. 11.14. The class `MouseHandler` extends the Java class `MouseAdapter` and is responsible for keeping track of mouse movements and clicks and invoking the appropriate controller methods to set up the label. In addition to capturing mouse clicks, the addition of labels requires the capturing of keystrokes. The class `KeyHandler` accomplishes this task by extending `KeyAdapter`.

[4]The reader is asked to study the documentation on this and other related interfaces and classes.

In another implementation, the view may choose to have other listeners that keep track of events like resising the window, zooming-in, etc. These do not affect the model and can be handled by redrawing the figure.

If the user abandons a particular drawing operation, we could be in a tricky situation where there is more than one `MouseHandler` object receiving mouse clicks and performing conflicting operations such as one object attempting to create a line and another trying to add a label. To prevent this, we have two mechanisms in place.

1. The `KeyAdapter` class also implements `FocusListener` to know when key strokes cease to be directed to this class.
2. The drawing panel ensures that there is at most one listener listening to mouse clicks, key strokes, etc. This is accomplished by overriding methods such as `addMouseListener` and `addKeyListener`.

11.6 Getting into the Implementation

11.6.1 Item and Its Subclasses

This class `Item` is abstract and its implementation is as follows:

```
import java.io.*;
import java.awt.*;
public abstract class Item implements Serializable {
  protected static UIContext uiContext;
  public static void setUIContext(UIContext uiContext) {
    Item.uiContext = uiContext;
  }
  public abstract boolean includes(Point point);

  protected double distance(Point point1, Point point2) {
    double xDifference = point1.getX() - point2.getX();
    double yDifference = point1.getY() - point2.getY();
    return ((double) (Math.sqrt(xDifference * xDifference +
                      yDifference * yDifference)));
  }
  public void render() {
    uiContext.draw(this);
  }
}
```

The `UIContext` and its significance were discussed earlier in the context of using the bridge pattern. The `includes` method is used to check if a given point selects the item.

The `Line` class looks something like this:

```
public class Line extends Item {
  private Point point1;
  private Point point2;
  public Line(Point point1, Point point2) {
```

```
    this.point1 = point1;
    this.point2 = point2;
  }
  public Line(Point point1) {
    this.point1 = point1;
  }
  public Line() {
  }
  public boolean includes(Point point) {
    return ((distance(point, point1 ) < 10.0) || (distance(point, point2)
              < 10.0));
  }
  public void render() {
    uiContext.draw(this);
  }
  // setters and getters for the two points
}
```

The class provides three constructors. A client may thus construct a `Line` object without knowing either endpoint, or by specifying one point, or after gathering both endpoints.

Unlike `HardUI` and `EasyUI`, which are 'imaginary' UI technologies, we can readily construct an implementation of `UIContext` for the Java Swing technology.

```
public class SwingUI implements UIContext {
  private Graphics g;
  // Any other fields to hold context variables
  public  void setGraphics(Graphics graphics) {
    g = graphics;
  }
  // any other methods to set context variables
  public void draw(Circle circle) {
    g.drawOval(/* parameters */);
  }
  public void draw(Line line) {
    g.drawLine(/* parameters */);
  }
  public void  draw(Label label){
    g.drawString(/* parameters */);
  }
  public void draw(Item item) {
    // error message
  }
}
```

As was the case earlier, `draw` needs information from both the UI and the item. The UI information is obtained within the context object and the item is passed in as a reference. The only difference is that instead of doing all this in the `render` method of `Item`, we invoke the appropriate draw method on the UI object with which the view has been configured.

11.6.2 Implementation of the Model Class

The class maintains `itemList` and `selectedList`, which respectively store the items created but not selected, and the items selected. The constructor initialises these containers.

```
public class Model extends Observable {
  private Vector itemList;
  private Vector selectedList;
  public Model() {
    itemList = new Vector();
    selectedList = new Vector();
  }
  // other methods
}
```

The `setUIContext` method in the model in turn invokes the `setUIContext` on `Item`.

```
public static void setUIContext(UIContext uiContext) {
  Model.uiContext = uiContext;
  Item.setUIContext(uiContext);
}
```

As an `Observable`, the model notifies all of the views when it needs to inform them of changes. We have seen that this approach allows us to change `UIContext` dynamically, and also supports the displaying of multiple views simultaneously, where each view is using a different `UIContext`.

At the moment, we handle the drawing of items (including a possibly 'incomplete' one), especially labels, by having a method `updateView` in the model, which is called by the controller at appropriate moments, for example after each character is read in from the keyboard. The method simply asks that the view be refreshed.

```
public void updateView() {
  setChanged();
  notifyObservers(null);
}
```

The `addItem` method is simple: it just stores the item in `itemList` and redraws the screen.

```
public void addItem(Item item) {
  itemList.add(item);
  updateView();
}
```

The class also provides a method to delete an item.

```
public void removeItem(Item item) {
  itemList.remove(item);
  updateView();
}
```

When an item is selected by the user, the model marks it as selected by transferring the item from `itemList` to `selectedList` as below.

```
public void markSelected(Item item) {
  if (itemList.contains(item)) {
    itemList.remove(item);
    selectedList.add(item);
    updateView();
  }
}
```

Selected items are deleted using the `deleteSelectedItems`.

```
public void deleteSelectedItems() {
  selectedList.removeAllElements();
  updateView();
}
```

The `getItems` method is used by the controller to determine which item is selected. The view uses the same method to render the items.

```
public Enumeration getItems() {
  return itemList.elements();
}
```

11.6.3 Implementation of the Controller Class

The class must keep track of the current shape being created, and this is accomplished by having the following fields within the class.

```
private Line line;
private Label label;
```

When the view receives a button click to create a line, it calls one of the following controller methods. The controller supplies three versions of the `makeLine` method and keeps track of the number of points independently of the view.

```
public void makeLine() {
  makeLine(null, null);
  pointCount = 0;
}
public void makeLine(Point point) {
  makeLine(point, null);
  pointCount = 1;
}
public void makeLine(Point point1, Point point2) {
  line = new Line(point1, point2);
  pointCount = 2;
  model.addItem(line);
}
```

The variables `pointCount` and `model` are both fields within the `Controller` class that respectively keep track of the number of points received and the instance of the `Model` class.

The execution of `makeLine` causes the line to be part of the model. The view can set the endpoints of the line via the following method.

```
public void setLinePoint(Point point) {
  if (++pointCount == 1) {
    line.setPoint1(point);
  } else if (pointCount == 2) {
    pointCount = 0;
    line.setPoint2(point);
  }
  model.updateView();
}
```

After it receives each end-point, the controller calls the model's `updateView` method to inform it that the view should be updated.

The approaches to draw a circle and add a label are similar. For a label, remember that by pressing the backspace the user can delete a character. So we provide a method `removeCharacter` for this purpose.

The following method is called by the view when it receives the command to select an item. The controller searches through the entire list of unselected items and determines if one of them is selected, and if so, it moves the item from the list of unselected items to the list of selected items.

```
public void selectItem(Point point) {
  Enumeration enumeration = model.getItems();
  while (enumeration.hasMoreElements()) {
    Item item = (Item)(enumeration.nextElement());
    if (item.includes(point)) {
      model.markSelected(item);
      break;
    }
  }
}
```

11.6.4 Implementation of the View Class

The view maintains two panels: one for the buttons and the other for drawing the items.

```
public class View extends JFrame implements Observer {
  private JPanel drawingPanel;
  private JPanel buttonPanel;
  // JButton references for buttons such as draw line, delete, etc.
  private class DrawingPanel extends JPanel {
    // code to redraw the drawing and manage the listeners
  }
  public View() {
    // code to create the buttons and panels and put them in the JFrame
  }
  public void update(Observable model, Object dummy) {
    drawingPanel.repaint();
  }
}
```

The code to set up the panels and buttons is quite straightforward, so we do not dwell upon that.

The DrawingPanel class overrides the paintComponent method, which is called by the system whenever the screen is to be updated. The method displays all unselected items by first obtaining an enumeration of unselected items from the model and calling the render method on each. Then it changes the colour to red and draws the selected items.

```
public void paintComponent(Graphics g) {
  model.setUI(NewSwingUI.getInstance());
  super.paintComponent(g);
  (NewSwingUI.getInstance()).setGraphics(g);
  g.setColor(Color.BLUE);
  Enumeration enumeration = model.getItems();
  while (enumeration.hasMoreElements()) {
    ((Item) enumeration.nextElement()).render();
  }
  g.setColor(Color.RED);
  enumeration = model.getSelectedItems();
  while (enumeration.hasMoreElements()) {
    ((Item) enumeration.nextElement()).render();
  }
}
```

The DrawingPanel class also overrides the addMouseListener, addKeyListener, and addFocusListener methods. This is to ensure that there is at most one listener for each type of event on the drawing panel.

```
private MouseListener currentMouseListener;
public void addMouseListener(MouseListener newListener) {
  removeMouseListener(currentMouseListener);
  currentMouseListener =  newListener;
  super.addMouseListener(newListener);
}
```

Similarly, we ensure that there is just one listener for events related to the keyboard.

Although the various button classes are alike in many respects, some are more complicated than others. One of the more complicated ones is LabelButton, which is responsible for handling label creation requests. Constructors of most button classes get a reference to the view, and the ones that need to access the drawing panel also get a reference to the panel.

```
public class LabelButton  extends JButton implements ActionListener {
  protected JPanel drawingPanel;
  protected View view;
  private KeyHandler keyHandler;
  private MouseHandler mouseHandler;
  private Controller controller;
  public LabelButton(Controller controller, View jFrame, JPanel jPanel) {
    super("Label");
    this.controller = controller;
    keyHandler = new KeyHandler();
    addActionListener(this);
    view = jFrame;
    drawingPanel = jPanel;
  }
  public void actionPerformed(ActionEvent event) {
    drawingPanel.addMouseListener(mouseHandler = new MouseHandler());
```

```
    }
    private class MouseHandler extends MouseAdapter {
      // details not shown
    }
    private class KeyHandler extends KeyAdapter implements FocusListener {
      // details not shown
    }
  }
```

When this button is clicked, an instance of `MouseHandler` is created, and it becomes the sole listener of mouse clicks. `MouseHandler` overrides the `mouseClicked` method to determine the starting point of the label. Besides asking the controller to set up a `Label` object with the given starting point, the code makes the drawing panel receive further button clicks and keyboard events. Also note that the `KeyHandler` is a `FocusListener` as well, which lets it know when it longer receives keyboard input.

```
public void mouseClicked(MouseEvent event) {
  view.setCursor(new Cursor(Cursor.TEXT_CURSOR));
  Controller.instance().makeLabel(event.getPoint());
  drawingPanel.requestFocusInWindow();
  drawingPanel.addKeyListener(keyHandler);
  drawingPanel.addFocusListener(keyHandler);
}
```

In its `keyTyped` method, `KeyHandler` transmits all printable characters to the `Label` object via the controller. The `keyPressed` method distinguishes between the enter and backspace keys. For the former, it stops listening to mouse clicks and keyboard events. If the backspace is pressed, the label is made to delete the last typed character.

```
public void keyTyped(KeyEvent event) {
  char character = event.getKeyChar();
  if (character >= 32 && character <= 126) {
    Controller.instance().addCharacter(event.getKeyChar());
  }
}
public void keyPressed(KeyEvent event) {
  if (event.getKeyCode() == KeyEvent.VK_ENTER) {
    view.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
    drawingPanel.removeMouseListener(mouseHandler);
    drawingPanel.removeKeyListener(keyHandler);
    drawingPanel.repaint();
  } else if (event.getKeyCode() == KeyEvent.VK_BACK_SPACE) {
    Controller.instance().removeCharacter();
  }
}
```

If the user terminates label creation by clicking on a button, as opposed to hitting the Enter key, the system executes the `focusLost` method of `KeyHandler`, which properly ends the command.

```
public void focusLost(FocusEvent event) {
  view.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
  drawingPanel.removeMouseListener(mouseHandler);
```

```
    drawingPanel.removeKeyListener(keyHandler);
    drawingPanel.repaint();
  }
```

Finally, just before it refreshes the screen, the view sets up `UIContext` within the model appropriately:

```
public void paintComponent(Graphics g) {
  model.setUI(NewSwingUI.getInstance());
  // rest of the code not shown
}
```

11.6.5 The Driver Program

The driver program sets up the model. In our implementation the controller is independent of the UI technology, so it can work with any view. The view itself uses the Swing package and is an observer of the model.

```
public class DrawingProgram {
  public static void main(String[] args){
    Model model = new Model();
    Controller.setModel(model);
    Controller controller = new Controller();
    View.setController(controller);
    View.setModel(model);
    View view = new View();
    model.addObserver(view);
    view.show();
  }
}
```

11.6.6 A Critique of Our Design

The partial design of the view and the model are quite robust. We have examined some of the issues to be taken care of earlier on, and the implementation takes them into consideration. The controller appears to be quite straightforward, and we simply need to add methods to handle all the operations.

Let us see how the design stands up to the task of adding a new operation, say, to draw a polygon.

1. We need to provide a new button which informs the user that the new operation is available. We also should create a mouse handler to handle mouse clicks, etc. These changes are relatively obvious and clearly unavoidable. Even then, note that most of the classes in the view are left unchanged.
2. The model is not affected by adding new types of items, operations or new UIs.
3. The `UIContext` interface does not have to be necessarily extended when new kinds of items are added. We refer the reader to the discussion in Sect. 11.5.2.

4. The controller should have new methods such as `makePolygon` and `addPointToPolygon`. It is not clear that this change is not a consequence of some basic flaw in our design. For instance, it might be possible to replace the methods `makeLine`, `makeCircle`, etc. by a single method, say `makeShape`.

Thus one drawback to our approach is that we need to change the controller class every time new operations are added or even if we change the way things are implemented. In addition, the controller has all the implementation in one class, which makes things complicated.

A more tricky problem is that of implementing *undo*. Clearly 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 a pattern that can help us improve the design of the controller.

11.7 Implementing the Undo Operation

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

- *Single-level undo versus 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.
- *Not all things are undoable* This can happen for two reasons. Some operations like 'print file' are irreversible, and hence undoable. Other operations like 'save to disk' may not be worth the trouble to undo, due to the overheads involved.
- *Blocking further undo/redo operations* It is easy to see that uncontrolled undo and redo can result in meaningless requests. In general, it is safer to block redo whenever a new command is executed. Consider a situation where we have the sequence: *Select(a), undo, Select(a), redo*. The redo tries to mark *a* as selected, and this could result in an exception depending on how things are implemented. A more severe problem arises with *Create Rectangle(r), Colour Rectangle(r, blue), undo, Delete(r), redo.* Here, the redo will attempt to colour a rectangle that does not exist any more.

- *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 object;
   public StackObject(String string, Object object) {
     name = string;
     this.object = object;
   }
   public String getName() {
     return name;
   }
   public Object getObject() {
     return object;
   }
}
```

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;
  }
}
```

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

```
public class Controller {
  private Stack history;
  public void makeLine(Point point1, Point point2) {
    Line line = new Line(point1, point2);
    model.addItem(line);
    history.push(new StackObject("line", new LineObject(line)));
```

```
    }
    // other fields and methods
  }
```

Decoding is simply a matter of popping the stack reading the `String`.

```
public void undo() {
  StackObject undoObject = history.pop();
  String name = undoObject.getName();
  Object obj = undoObject.getObject();
  if (name.equals("line")) {
    undoLine((LineObject)obj);
  } else if (name.equals("delete")) {
    undoDelete((DeleteObject)obj);
  } else if (name.equals("select")) {
    undoSelect((SelectObject)obj);
  }
  // one else if for each command
}
```

Finally, undoing is simply a matter of retrieving the reference to and removing the line form the model.

```
public class Controller {
  public void undoLine(LineObject object){
    Line line = object.getLine();
    model.removeItem(line);
  }
}
```

There are two obvious drawbacks with this approach:

1. *The long conditional statement in the undo method of the controller.*
2. *The need to rewrite the controller whenever we make changes such as adding or modifying the implementation of an operation.*

The object-oriented approach for dealing with the first drawback is to subclass the behaviour 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 `undo` in the controller is mostly working off the data stored in `StackObject` and our first order of business is to extract and move this method.

```
public class Controller {
  private Stack history;
  public  void undo() {
    StackObject undoObject = history.pop();
    undoObject.undo(this);
  }
  // other fields and methods
}

public class StackObject {
  public  void undo(Controller controller) {
```

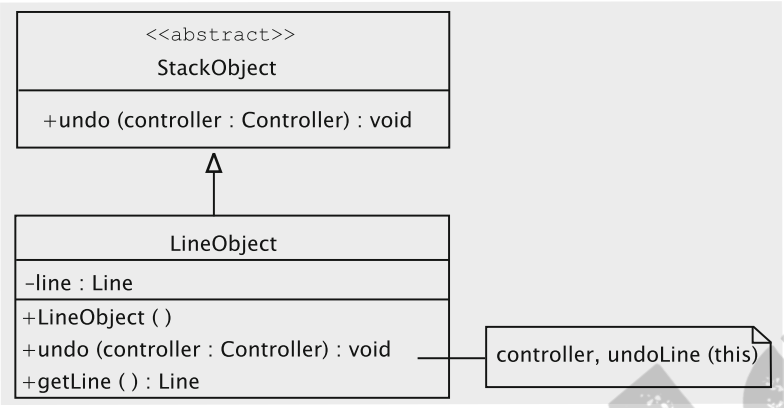


Fig. 11.15 Representing the drawing of a line

```
      String name = getName();
      Object object = getObject();
      if (name.equals("line")) {
        controller.undoLine((LineObject)object);
      } else if (name.equals("delete")) {
        controller.undoDelete((DeleteObject)object);
      } else if (name.equals("select")) {
        controller.undoSelect((SelectObject)object);
      }
    }
    // other fields and methods
}
```

Now our conditional is in `StackObject` and we are ready to subclass this behaviour. Since each kind of data object is associated with an operation, our hierarchy will have a subclass corresponding to each operation. For example, to represent the drawing of a line, we have the class `LineObject` as a subclass of `StackObject` (Fig. 11.15).

This 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 `makeLine` method is simplified, so it just creates a `LineObject` and pushes it onto the stack.

```
public void makeLine(Point point1, Point point2) {
  Line line = new Line(point1, point2);
  model.addItem(line);
  history.push(new LineObject(line));
}
```

In the next subsection, we look into creating a fully reusable controller.

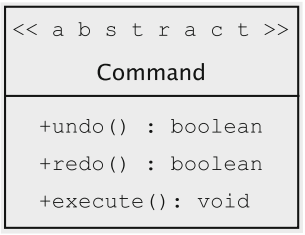


Fig. 11.16 The command class

11.7.1 Employing the Command Pattern

The reader may have noticed a familiar pattern in the above code. In its `undo` method, the controller 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 is an implementation of *double dispatch* that we used when employing the *visitor* pattern and was wholly appropriate when introducing new functionality into an existing hierarchy. In this context, however, we find that 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 (The Law of Inversion), which we discussed in Chap. 8, which we can utilise in this problem by using the **command pattern.**

The intent of the command pattern is as follows (see foonote 1):

> Encapsulate a request as an object, thereby letting you parametrise clients with different requests, queue or log requests, and support undoable 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 recognise so far is that this object need not merely be a repository of associated data but can also encapsulate the routines that need access to 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`. See Fig. 11.16.

The default `undo` and `redo` methods in `Command` return `false`, and these need to be overridden as needed by the concrete command classes.

The mechanism is best explained via an example, for which we develop a somewhat simplified sequence diagram for the command to add a line (Fig. 11.17).[5]

Adding a line Since every command is represented by a `Command` object, the first order of task when the Draw Line command is issued is to instantiate a `LineCommand` object. We assume that we do this after the user clicks the first endpoint although there is no reason why it could not have been created immediately after receiving the

[5]The sequence diagram abstracts out the complexity of the multiple classes associated with the UI into a single class called View.

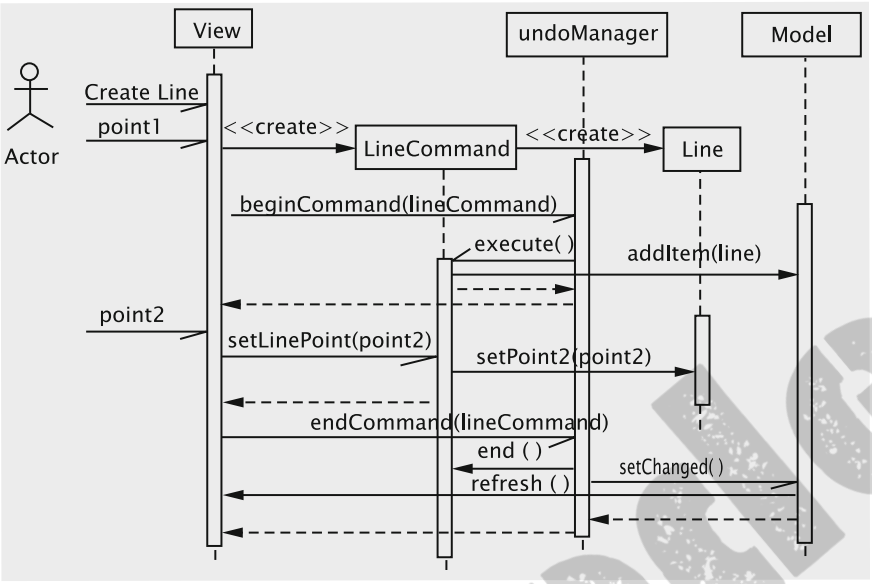


Fig. 11.17 Sequence diagram for adding a line

command. In its constructor, `LineCommand` creates a `Line` object with one of its endpoints specified.

The central idea behind the command pattern is to employ two stacks: one for storing the commands that can be undone (history stack) and the other for maintaining the commands that may be redone (redo stack). The class `UndoManager` maintains these stacks. (We refer to the corresponding object by the term **undo manager**.) The undo manager plays the role of the controller, but we have given it a new name to highlight its main function. We take the approach that as soon after the command object is created, the view informs the undo manager, which is then expected to initiate its bookkeeping operations. Similarly, when the view has received all of the data needed to complete the command, it notifies the `UndoManager` once more. The two methods `beginCommand` and `endCommand` are for these two purposes.

In the course of execution of the `beginCommand` method, the undo manager ensures the the `Line` object gets added to the model. This way, should the view be refreshed, the partial line will be shown on the screen.

When the command is completed and the `endCommand` method is executed, the undo manager pushes the command onto the history stack. This way the latest command is always at the top of this stack. To prevent inconsistencies of the kind we described at the very beginning of this section, we clear the redo stack whenever a new command is issued.

Assume that the user issues the sequence of commands:

```
Add Label (Label 1)
Draw Circle (Circle 1)
```

Add Label (Label 2)
Draw Line (Line 1)

At this time, there are four `Command` objects, one for each of the above commands, and they are on the history stack as in Fig. 11.18. The redo stack is empty: since no commands have been undone, there is nothing to redo. The picture also shows the collection object in the model storing the two `Label` objects, the `Circle` object, and the `Line` object.

Undoing an operation Continuing with the above example, we now look at the sequence of actions when the undo request is issued immediately after the line (Line 1) has been completely drawn in the above sequence of commands. Obviously, the user views the command as undone if the line disappears from the screen: for this, the `Line` object must be removed from the collection. To be consistent with this action and to allow redoing the operation, the `LineCommand` object must be popped from the history stack and pushed onto the redo stack. The resulting configuration is shown in Fig. 11.19.

Not every command is undoable. So the general rule is that when the undo operation is requested, if the top of the undo stack is a command that can be undone, the command is undone and transferred to the redo stack.

The redo operation is simple enough: if the redo stack is not empty, the command must be re-executed, and the top object in the redo stack must be transferred to

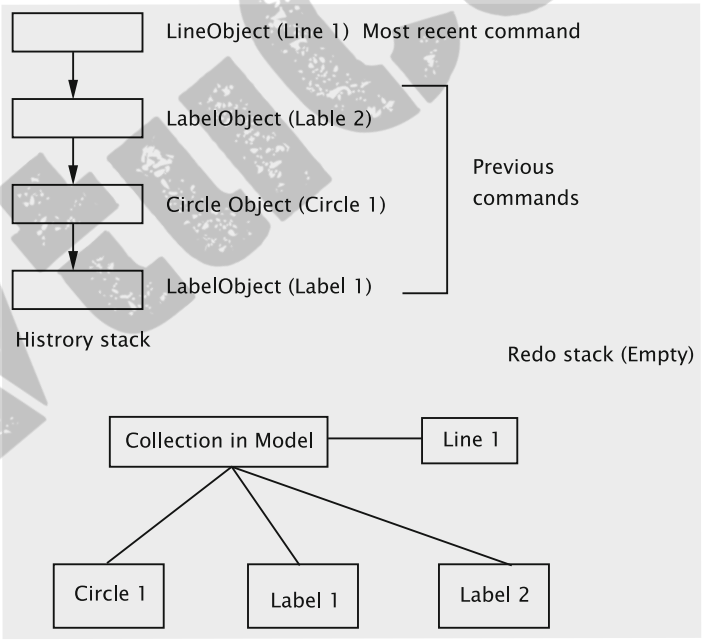


Fig. 11.18 Status of the stacks and the collection in the model

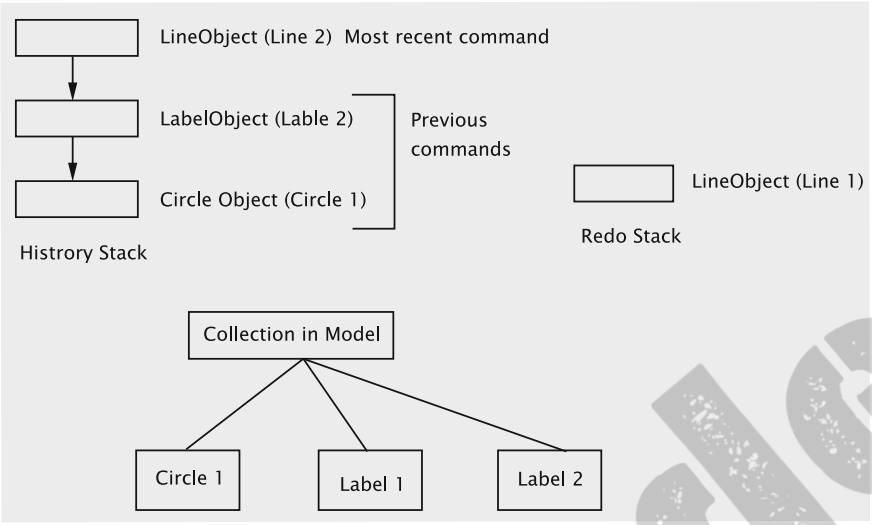


Fig. 11.19 Status of the stacks and the collection in the model after undo

the history stack. The redo involves updating the model: the `redo` method of the `Command` object calls the `execute` method to do the necessary actions. For the `LineCommand` object, this involves adding the line back to the model's collection object.

As we noted earlier, not every command may be undoable, or, at least, is not worth the trouble. If an undoable operation is on the undo stack, the undo cannot proceed beyond that operation although there might be undoable operations underneath it in the stack. To get around this problem, we might choose to not push undoable commands onto the stack. This can be accomplished by making the command itself assume the responsibility for pushing onto the history stack. This can conveniently be done in the class's constructor.

A related issue concerns unfinished commands. We use the term *incomplete command* to refer to a command that has not yet been properly terminated. An *incomplete item* is an item, such as a line or a label, that might not have proper values for every field. We use the term *complete item* to refer to an item for which the user has supplied (or the system has invented) all the input necessary for completely specifying the item. For example, suppose a user clicks the 'Create Line' button and clicks one point. Before clicking a second time to specify the second point, suppose the user clicks the 'Add Label' button. The Create Line command is incomplete. Moreover, the line is also incomplete at this stage, and it is already stored in the model, which now ends up containing incomplete data. One could argue that it was the user's fault, but the program must tolerate such errors and it would be nice if there was a way to fix this problem.

How should this be handled? We can suggest at least two ways:

1. We could prevent the possibility of users aborting commands in the middle. A popular approach is to disable all command buttons when a new command is finished and leave them disabled until the command is completed. When the command is completed, all of the buttons are enabled.
2. A second possibility is to handle this with an additional method in both the undo manager and the command class.

The difficulty with the first approach is that the UI is responsible for ensuring data consistency. The responsibility for ensuring that items are complete must rest with the command classes and not with the user interface.

We proceed with the second choice, for which we will have the undo manager keep the current command away from the history stack until the command itself 'certifies' that it is complete. For this purpose, every command class has an additional method, `end`, which checks whether the item is complete and attempts to fill the missing values if necessary. If there is not enough data to make the item complete, the method returns a `false` value and the undo manager does not put the command on the stack.

The pseudo-code for the `end` method is as follows:

```
public boolean end() {
  if item is incomplete
     attempt to complete using data already received;
     if cannot be completed
        return false;
     end if
  end if
  return true
}
```

The undo manager does not push a new command onto the stack until it is clear that the item is complete.

We now explain the implementation of the above concepts.

11.7.2 Implementation

Subclasses of `Command` The concrete command classes (such as `LineCommand`) store the associated data needed to undo and redo these operations. Just as the `makeLine` method in the previous implementation had three versions, the `LineCommand` class has three constructors, allowing some flexibility in the design of the view.

The implementation of methods specific to the `Command` class are shown below. The `execute` method simply adds the command to the model so the line will be drawn. To undo the command, the `Line` object is removed from the model's collection. Finally, `redo` calls `execute`.

```
public void execute() {
  model.addItem(line);
}
public boolean undo() {
```

```
    model.removeItem(line);
    return true;
  }
  public boolean redo() {
    execute();
    return true;
  }
```

As explained earlier, the class has a method called `end`, which attempts to complete an unfinished command. The situation is considered hopeless if both endpoints are missing, so the object removes the line from the model (undoes the command) and returns a false value. Otherwise, if the line is incomplete (has at least one endpoint unspecified), the start and end points are considered the same. The implementation is:

```
public boolean end() {
  if (line.getPoint1() == null) {
    undo();
    return false;
  }
  if (line.getPoint2() == null) {
     line.setPoint2(line.getPoint1());
  }
  return true;
}
```

UndoManager It declares two stacks for keeping track of the undo and redo operations: (`history`) and (`redoStack`). The current command is stored in a field aptly named `currentCommand`.

```
public class UndoManager {
  private Stack history;
  private Stack redoStack;
  private Command currentCommand;
}
```

If the command was not properly terminated, we arrange matters such that `currentCommand` will not be `null` when a new command is issued. Recall that when a new command is issued, the `beginCommand` method of the undo manager is called. If `currentCommand` is not `null` at this time, the undo manager attempts to complete it by calling the command's `end` method. The `beginCommand` method is implemented as below.

```
public void beginCommand(Command command) {
  if (currentCommand != null) {
    if (currentCommand.end()) {
      history.push(currentCommand);
    }
  }
  currentCommand = command;
  redoStack.clear();
  command.execute();
}
```

The `undo` and `redo` are straightforward operations.

```
public void undo() {
  if (!(history.empty())) {
    Command command = (Command) (history.peek());
    if (command.undo()) {
      history.pop();
      redoStack.push(command);
    }
  }
}

public void redo() {
  if (!(redoStack.empty())) {
    Command command = (Command)(redoStack.peek());
    if (command.redo()) {
      redoStack.pop();
      history.push(command);
    }
  }
}
```

When a command is complete, the view calls the `endCommand` method of the undo manager, which pushes `currentCommand` onto the history stack and sets `currentCommand` to `null`.

```
public void endCommand(Command command) {
  command.end();
  history.push(command);
  currentCommand = null;
  model.updateView();
}
```

Handling the input The view declares one button class for each command (add label, draw line, etc.). The class for handling line drawing is implemented as below.

```
public class LineButton  extends JButton implements ActionListener {
  // fields for view, drawing panel, handlers, etc.
  public LineButton(UndoManager undoManager, View jFrame, JPanel jPanel) {
    // store the parameters and create the mouse listener
  }
  public void actionPerformed(ActionEvent event) {
    // change the cursor
    drawingPanel.addMouseListener(mouseHandler);
  }
  private class MouseHandler extends MouseAdapter {
    public void mouseClicked(MouseEvent event) {
      if (first point) {
        lineCommand = new LineCommand(event.getPoint());
        UndoManager.instance().beginCommand(lineCommand);
        } else if (second point) {
        lineCommand.setLinePoint(event.getPoint());
        drawingPanel.removeMouseListener(this);
        view.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
        UndoManager.instance().endCommand(lineCommand);
      }
    }
  }
}
```

The above class thus directly creates the appropriate command object when a request comes from a user.

11.8 Drawing Incomplete Items

Recall the terms incomplete item and complete item we introduced in the previous section. There are a couple of reasons why in the drawing program we might wish to distinguish between these two types of items.

1. Incomplete items might be rendered differently from complete items. For instance, for a line, after the first click, the UI could track the mouse movement and draw a line between the first click point and the current mouse location; this line keeps shifting as the user moves the mouse. Likewise, if we were to extend the program to include triangles, which need three clicks, one side may be displayed after two clicks. Labels in construction must show the insertion point for the next character.
2. Some fields in an incomplete item might not have 'proper' values. Consequently, rendering an incomplete item could be more tricky. An incomplete line, for instance, might have one of the endpoints null. In such cases, it is inefficient to use the same render method for both incomplete items and complete items because that method will need to check whether the fields are valid and take appropriate actions to handle these special cases. Since we ensure that there is at most one incomplete item, this is not a sound approach.

We can easily distinguish between incomplete items and complete items by having a field that identifies the type. The render method will behave differently based on this field. The approach would be along the following lines.

```
public class Line {
  private boolean incomplete = true;
  public boolean isIncomplete() {
    return incomplete;
  }
  // other fields and methods
}

public class NewSwingUI implements UIContext {
  // fields and methods
  public void draw(Line line) {
    if (line.isIncomplete()) {
       draw incomplete line;
    } else {
       draw complete line;
    }
  }
}
```

In circumstances such as the above, where we have variant behaviour based on field values, the object-oriented philosophy dictates subclassing, i.e., we treat the incomplete item as a different class of object with its own rendering method. We create classes for incomplete items (such as `IncompleteLabel`) that are subclasses of items (such as `Label`). Since the class `IncompleteLabel` is a subclass of `Label`, the model is unaware of its existence. Once the object is created, the incomplete object can be removed from the model.

The details are as follows.

```
import java.awt.*;
public class IncompleteLabel extends Label {
  public IncompleteLabel(Point point) {
    super(point);
  }
  public void render() {
    // code for rendering IncompleteLabel
  }
  public boolean includes(Point point) {
    return false;
  }
}
```

One problem we face with the above approach is that `UIContext` must include the method(s) for drawing the incomplete items (`draw(IncompleteLabel label)`, in our example). This suggests that `UIContext` needs to be modified. However, the manner in which incomplete items are rendered is an issue that largely relates to the look and feel of the system. For instance, `UIContext` might not have a method `draw(IncompleteLine line)` and creator of some view (`NewSwingUI`, for instance) might wish to include that. In general, we would like a solution that allows for a *customised presentation* which may require subclassing the behaviour of some concrete items. This can be accomplished through RTTI. In particular, the situation where the `NewSwingUI` wants its own method for drawing an incomplete line is implemented as follows:

```
public class NewSwingUI implements UIContext {
  // fields and methods
  public void draw(Line line) {
    if (line instanceof IncompleteLine) {
       this.draw((IncompleteLine) line);
    } else {
       //code to draw Line
    }
  }
}
```

Where Should We Employ RTTI?

The use of RTTI can be puzzling to a beginner. On the one hand its application is actively discouraged; this attitude is fully justified since a novice developer can feel tempted to employ RTTI and resolve problems that really need a more thoughtful approach and a carefully designed hierarchy with appropriate design patterns. On the other hand, there are situations where it is necessary to check the type of an object at run time, as we have seen in Chaps. 5 and 10 and also in the case of the incomplete items in this chapter. In the examples in the earlier chapters, the development of the solution naturally led to the use of RTTI. In Chap. 5, the only way to know the exact type of the class that invoked the constructor was to invoke `getClass().getName()`. In Chap. 10, we had a situation where the expected behaviour was that the right

kind of listener would be passed as a parameter. If the expectation was met, the downcast would succeed; if not, throwing the exception was the right thing to do. In some situations, as with incomplete items in this chapter, it may not be so clear. A simple thumb rule for resolving this conundrum is to examine all the options that are available.

Consider what other choices we have for incorporating incomplete items. One approach would be to define `UIContext` to contain draw methods for all the incomplete items as well. This means that all concrete contexts must implement (dummy, perhaps) draw methods for incomplete items. Apart from the tedium of this and the fact that we are doubling the number of classes in the basic system, we have a solution that does not really allow for flexibility for the view to define the look and feel. We could conceivably have a system with different kinds of incomplete labels, each associated with different processes for label creation. With RTTI, we have a solution that allows for variability in a manner that does not affect other parts of the implementation.

The `LineCommand` object creates an `IncompleteLine` and adds this to the model. This new class is thus known only to the controller and `NewSwingUI`. When the label creation is complete, the `IncompleteLine` object is removed from the model and replaced with a `Line` object. This implementation therefore gives a solution where variability is contained.

Finally, we examine item creation in this new context. Assume that the user clicks on the 'Add Label' button. On the creation of the `LabelCommand` object, an `IncompleteLabel` object is created and stored within the command object. When label is completed, the `end` method of the command object is called, and in this method, a `Label` object is created and data from the incomplete version is copied to it. The `IncompleteLabel` object is deleted from the model and the `Label` object takes its place. The relevant code from `LabelCommand` is shown below.

```
public void end() {
  model.removeItem(label);
  String text = label.getText();
  label = new Label(label.getStartingPoint());
  for (int index = 0; index < text.length(); index++) {
    label.addCharacter(text.charAt(index));
  }
  execute();
}
```

This completes the basic implementation of our simple graphical system. Note that if any new operation has to be added, all we have to do is create new classes that extend `Command` and `Item`, and modify the view to allow the user to invoke the new operation. Modifying the view is simply a matter of defining a new class that extends `JButton` and adding an instance of this class to the button panel. The model, the view and the controller are essentially repositories for the items, buttons, and commands respectively, and thus provide a framework for creating the specified system.

11.9 Adding a New Feature

Most interactive systems that are used to create graphical objects, allow users to define new kinds of objects on the fly. A system for writing sheet music may allow a user to define a sequence of notes as a group. This would enable the user to manipulate these notes as a group, making copies of these as needed. In a system for drawing electrical circuits, a set of components interconnected in a particular way could be clustered together as a 'sub-circuit' that can then be treated as a single unit. In a drawing program like the one we have created, a complex figure may be created as a collection of lines and circles, which may have to be moved around a single unit. In all these cases, the user-friendliness of the system would be considerably improved if a feature is provided to enable such operations.

Let us examine how our system needs to be modified to accommodate this. The process for creating such a 'compound' object would be as follows: *The user would select the items that have to be combined by clicking on them. The system would then highlight the selected items. The user then requests the operation of combing the selected items into a compound object, and the system combines them into one.*

Which Subsystem 'Owns' a Class?

In our original approach to designing this system using the MVC architecture, we were partitioning the responsibilities between the three subsystems. As we looked into the finer details of the implementation, we encountered some problems and found some suitable patterns that could improve our design. The use of these patterns, however apparently 'blurs' some of the clear boundaries.

Consider for instance the bridge pattern. We created the `UIContext` interface within the model to house the `draw` methods of all the items. The model does not have the information, however, to create a concrete instance of `UIContext` and this task is left to the `View` class. `UIContext` and its implementing classes belong to the view subsystem.

The original controller was replaced by a collection of classes including `UndoManager` and the various subclasses of `Command`, so they could be considered belonging to the controller subsystem. The undo manager defines the interface for the command but does not have any information on how each individual command should receive and process input.

The reader should realise that the subsystems are only providing a context within which the details can be fleshed out. The controller is providing a format for the creation of commands and also a system that manages these commands. When a command has to be added, a class is defined and the view is modified to allow for its invocation. Likewise the model provides a template for rendering all the kinds of items, but a complete knowledge of the view is needed to provide a concrete implementation.

From a more practical view, it does not matter much whether we can label a class as belonging to any specific subsystem. What we need to worry about are properties such as modularity, proper assignment of responsibilities, cohesive classes, low coupling between classes, ease of meeting changing requirements, performance, and so on. The MVC paradigm provides the guidelines, and it is up to the designer to make decisions that ensure these properties.

Once a compound object has been created, it can be treated as a any other object. This process can be *iterated,* i.e., a compound object can be combined with other objects (which could themselves be compound or simple objects) to create another compound object. The system also allows the user to 'breakdown' a compound item into its constituent items by first selecting the item(s) to be broken down and then choosing the 'decompose' operation. Note that if a compound item is created by combining two compound items, then decomposing it will give us back the two original compound items. Finally, the system must have the ability to undo and redo these operations.

Since we have to store a collection of items, an obvious approach to implementing this would be to create a new kind of item that maintains a collection of the constituent items. This would be a concrete class and would look like this:

```
public class CompoundItem {
  List items;
  public CompoundItem(/* parameters */) {
    //instantiate lists
  }
  public Enumeration getItems() {
    //returns an enumeration of the objects in Items
  }
  // other fields and methods
}
```

Since `items` consists of both simple items and compound items, it seems logical that all entities stored in `items` are designated as belonging to the class `Object`. The model would also have to be modified so that the container classes would hold collections of type `Object`.

Consider now any class that examines at the collection of items in the model (i.e., a 'client' class). One of these would be the `SelectCommand`. When a `SelectCommand` object gets the coordinates of the mouse click, it iterates through the collection in the model to determine the selected item. If the object is a simple item, it would be cast as an `Item` and the `includes` method would be invoked; if the object is a compound item, it would be cast as a `CompoundItem` and the `getItems` method would be invoked to get an enumeration of the objects that make up the compound item. Clearly, this is not the most desirable state of affairs since the client method is querying the type of the object (which is akin to switching on the fields of the object) to determine what operation is to be performed. Our standard approach in such situations is to create an inheritance hierarchy and use dynamic

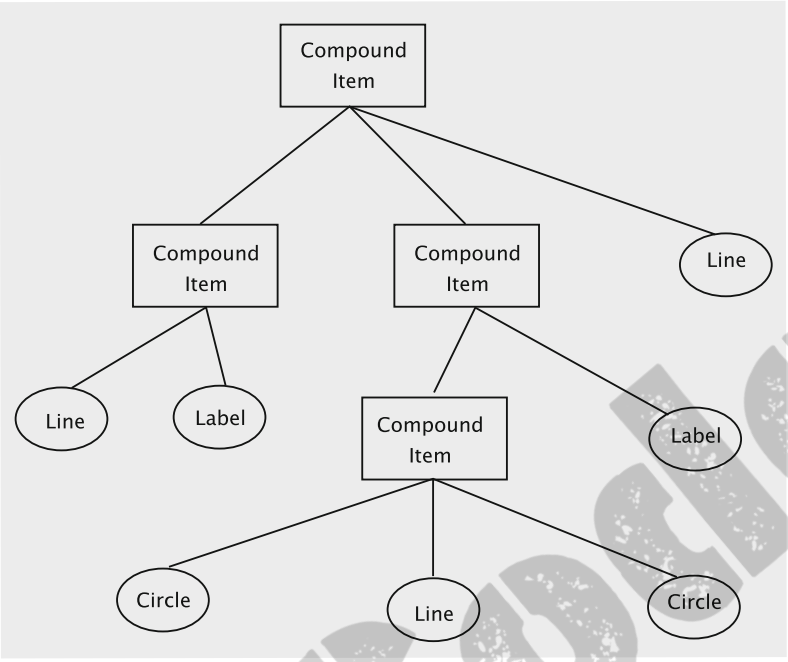


Fig. 11.20 Tree structure formed by compound items

binding. The dilemma here is that we have a two fundamentally different kinds of entities: *a simple item is a single item, whereas a compound item is a collection of items.* The **composite pattern** gives us an elegant solution to this problem.

The intent of the composite pattern is as follows (see footnote 1):

> *Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.*

A compound item is clearly a composition of simple items. Since each compound item could itself consist of other compound items, we have the requisite tree structure (see Fig. 11.20).

The class interaction diagram for the composite pattern is shown in Fig. 11.21. Note that the definition of the compound item is *recursive* and may remind readers of the recursive definition of a tree. Following this diagram, the class `CompoundItem` is redefined as follows:

```
public class CompoundItem extends Item {
  List items;
  public CompoundItem(/* parameters */){
    //instantiate lists
  }
  public void render(){
    // iterates through items and renders each one.
  }
```

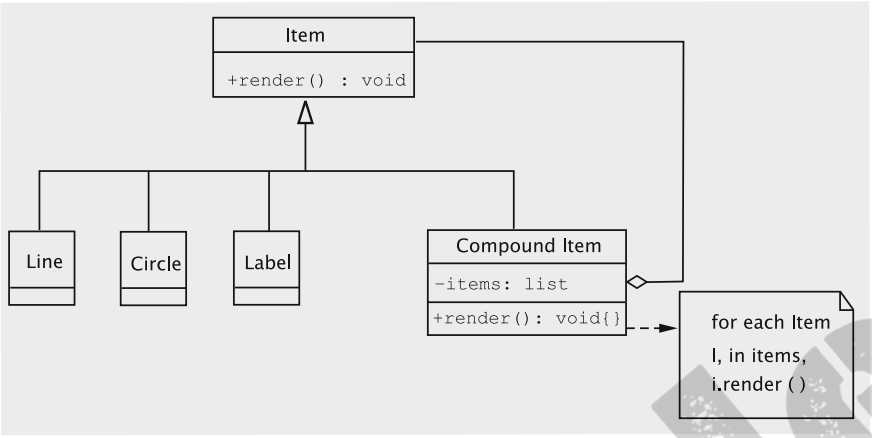


Fig. 11.21 Composite structure of the item hierarchy

```
    public boolean includes(Point point) {
      /* iterates through items and invokes includes  on each item.
         Returns true if any of the items returns true and false otherwise. */
    }
    public void addItem(Item item) {
      // Adds item to items
    }
      // other fields and methods
}
```

Modifying the system to allow for creating compound objects is just like any of the operations discussed earlier. The system already has an operation for selecting items. Once that is complete, user chooses the 'create composite' operation. This would require that a new class be defined (extending `JButton`) and that the view be modified to add this button to the button panel. A new class, `CompositeCommand` (extending `Command`) is defined. The `execute` method of this class removes all the selected items from the Model and adds them to a new `CompoundItem` object, which is then added to the Model. The view renders a `CompoundItem` exactly in the same way as it renders any other instance of `Item`. Note also that the select operation invokes the `includes` method on `CompoundItem` exactly as it would on simple items.

11.10 Pattern-Based Solutions

As explained earlier a pattern is a solution template that addresses a recurring problem in specific situations. In a general sense, these could apply to any domain. (A standard opening in chess, for instance, can be looked at as a 'chess pattern'.) In the context of creating software, three kinds of patterns have been identified. At the highest level, we have the **architectural patterns**. These typically partition a system into

subsystems and broadly define the role that each subsystem plays and how they all fit together. Architectural patterns have the following characteristics:

- *They have evolved over time* In the early years of software development, it was not very clear to the designers how systems should be laid out. Over time, some kind of categorisation emerged, of the kinds software systems that are needed. In due course, it became clearer as to how these systems and the demands on them change over their lifetime. This enabled practitioners to figure out what kind of layout could alleviate some of the commonly encountered problems.
- *A given pattern is usually applicable for a certain class of software system* The MVC pattern for instance, is well-suited for interactive systems, but might be a poor fit for designing a payroll program that prints paychecks.
- *The need for these is not obvious to the untrained eye* When a designer first encounters a new class of software, it is not very obvious what the architecture should be. One reason for this is that the designer is not aware of how the requirements might change over time, or what kind of modifications are likely to be needed. It is therefore prudent to follow the dictates of the wisdom of past practitioners. This is somewhat different from design patterns, which we are able to 'derive' by applying some of the well-established 'axioms' of object-oriented analysis and design. (In case of our MVC example, we did justify the choice of the architecture, but this was done by demonstrating that it would be easier to add new operations to the system. Such an understanding is usually something that is acquired over the lifetime of a system.)

At the next level, we have the *design patterns.* These solve problems that could appear in many kinds of software systems. Once the principles of object-oriented analysis and design have been established it is easier to derive these. Examples of these can be found throughout this text.

At the lowest level we have the patterns that are called **idioms**. Idioms are the patterns of programming and are usually associated with specific languages. They typically refer to the use of certain syntactic elements of the language. As programmers, we often find ourselves using the same code snippet every time we have to accomplish a certain task. Sometimes, we may save these as 'macros' to be copied and pasted as needed thus enabling us to be more productive in terms of code-generation. Idioms are something like these, but they are usually carefully designed to take the language features (and quirks!) into account to make sure that the code is safe and efficient. The following code, for instance, is commonly used to swap:

```
temp = a;
a = b;
b = temp;
```

In *Perl*,[6] the list assignment syntax allows us to employ a more succinct expression:

```
($a, $b) = ($b, $a);
```

[6] A commonly used scripting language.

This would be an example of an idiom for Perl. In addition to safety and efficiency, the familiarity of the code snippet makes the code more readable and reduces the need for comments. Typical Perl programmers might be more comfortable with the second whereas a Java programmer would prefer the first.

Not all idioms are without conflict. There are two possible idioms for an infinite loop:

```
for (;;) {
// some code
}
while (true) {
// some code
}
```

It has been argued that the first one should be preferred for efficiency, since no expression evaluation is involved at the end of each iteration. However, with the availability of optimising compilers and increasing hardware capacity nowadays, some programmers are making a case for the second one based on readability and elegance.

Familiarity with and acceptance of established patterns is clearly a must for success in any domain of activity. Most of our focus in our case studies has therefore been to convince the student of their usefulness by showing how they provide elegant solutions to naturally arising design problems. However, as mentioned earlier, it is much more difficult for a beginner to grasp the significance of architectural patterns in this manner.

11.10.1 Examples of Architectural Patterns

The Repository

This architecture is characterised by the presence of a single data structure called the *central repository*. Subsystems access and modify the data stored in this. An example of such a system could be software used for *managing an airline.* The subsystems in this case could be the ones for managing reservations, scheduling staff, and scheduling aircraft. All of these would access a central data repository that holds information about aircraft, staff, and passengers. These would be interrelated, since a choice of an aircraft could likely influence the choice of staff and be influenced by the volume of passenger traffic. In such systems, the control flow can be dictated by the central repository (changes in the data characteristics could trigger some operations), or from one of the subsystems. Another application of such a system could be for managing a large bank. The account information would have to be centrally located and could be accessed and modified from several peripheral locations. A software development system or a compiler could also employ such an architecture by having a centralised parse-tree or symbol table.

The Client-Server

In such a layout, there is a central subsystem known as a *server* and several smaller subsystems known as *clients* which are typically quite similar. There is a fair amount of independence in the control flow, and each subsystem may be using a different thread. Synchronisation techniques are often employed to manage requests and transmit results.

The world-wide-web is probably the best example of such an architecture. The browsers running on PCs are like clients and the sites they access play the role of servers. The server could also be housing a database and the clients could be processes that are querying and updating the database. A variant/generalisation of this is the **peer-to-peer** architecture where the client/server role of the subsystems are interchangeable. These variants are typically hard to design due to the possibility of deadlocks and a myriad of other problems that can complicate the flow of control.

The Pipe and Filter

The system in this case is made up of *filters*, i.e., subsystems that process data, and *pipes*, which can be used to interconnect the filters. The filters are completely mutually independent and are aware only of the input data that comes through a pipe, i.e., the filter knows the form and content of the data that came in, not how it was generated. This kind of architecture produces a system that is very flexible and can be dynamically reconfigured. In their simplest form, the pipes could all be identical, and each filter could be performing a fixed task on data input stream. An example of this would be that of processing incoming/outgoing data packets over a computer network. Each 'layer' would be like a filter that adds to, subtracts from or modifies the packet and sends it forward.

The Unix operating system is a more sophisticated version of such an architecture, and allows the user to create more complex operations by linking together simpler ones. In its most general form, one could have pipes that 'reformat' the data, so that any sequence of filters could be used.

11.11 Discussion and Further Reading

Software architectures and design patterns bear some similarity in that they both present efficient solutions to commonly occurring problems. The process of learning how to apply these are however very different. It is possible (and perhaps pedagogically preferable) to 'discover' design patterns by critically examining our designs and refactoring them. Such a process does not lend itself well to the task of learning about architectures due to the complexity of the problem we are encountering. The software designer's best bet is to learn about commonly used architectures in the given problem domain and adapt them to the current needs [1].

In this chapter and the previous one, we introduced design patterns by coming up some 'reasonable' design and then critically examining it using our knowledge of the

principles of object-oriented analysis and design. This process is not very different from that of refactoring to introduce patterns into existing code. The process for refactoring to introduce patterns has been well-studied and cataloged [2, 3].

11.11.1 Separating the View and the Controller

When studying the MVC architecture, we often hear the phrase 'model–view separation', which refers to the idea that we keep the reality and representation distinct from each other. In our case-study, we have done this by having the model manage a list of items, and leaving all other responsibilities to other subsystems.

The separation between view and controller is less clear. In our implementation, we have chosen to make the knowledge of concrete command classes available to the classes that receive the user request. This makes for a clean implementation, since the request is immediately packed into an object that can be managed by the controller. The literature does mention other of ways of implementing the command pattern, which are based on the notion that the command object must be created in the controller subsystem. One approach that has been suggested is to allow the requests for operations to be received in a class in the controller. This has the drawback that the controller must implement methods (like `ActionListener`) that are really dependent on the view implementation, thus causing tight coupling. Another approach is to capture the request as a string (see [4]), which is then parsed in a command factory to generate the command object. This results in an unnecessary loss of performance.

All of this underscores the fact that the view and controller are not easily and clearly separable in every context. One obvious question that arises is: *Why not move the controller operations into the view?* This has led to a variant of the MVC, called the 'document-view' architecture, where the document holds the Model and the view handles the functions of the controller as well.

11.11.2 The Space Overhead for the Command Pattern

One of the drawbacks of the command pattern is that it places a large demand on the memory resources, which in turn has a serious effect on runtime. Some systems restrict the number of levels of undo and redo to some manageable number to avoid this problem, but this solution may not always be acceptable.

Another approach that has been proposed is that each command be a singleton that keeps its own `history` and `redoStack` objects. No instances of command are created at the time of invocation, but the controller pushes a reference to the singleton command object into its own history stack. The invoked command creates the data object necessary to undo the operation and pushes it into its own `history` stack. This approach is particularly beneficial if we go with a Document–View architecture.

11.11.3 How to Store the Items

The manner in which items are stored in the model can affect the time it takes to render the items and thus affect performance. Consider the problem of rendering a curve that is specified by user as a collection of 'control points'. If the constructor decomposed the curve into a collection of line segments, then the process of rendering would be to simply draw each line segment. On the other hand, if the model stored only the control points, rendering (i.e., the corresponding `draw` method in the concrete `UIContext`) would have to compute all the line segments and then draw them. In the first case we are creating a large number of objects and storing these in the model. The rendering could be slowed down because of the large number of objects that have to be accessed. In the second, rendering may be delayed by the amount of computation. In general, memory access involves a much greater overhead than computation, and therefore one would expect the second approach to give better runtime performance.

11.11.4 Exercising Caution When Allowing Undo

Implementing the undo operation can be quite tricky. The process of executing a command could involve the methods of several classes and care must be taken to ensure that these are correctly reversible. A full treatment of this is beyond the scope of this text, but we can highlight a few of these issues.

What Should be Saved to Undo an Operation?

We must keep in mind that what we are undoing is the consequences of the operation on the entire system. Consider the process of undoing the creation of a line. The only input to the operation are the two end points, and elementary mathematics tells us that we do not need any other information to define a line. However, this information is not sufficient for us to undo the effects of this operation. The consequence to the system is that the a line object is added to the model, and what we need to store is a reference to this object. The model also must allow for a specified item to be removed; if this were not possible, the operation of removing a line would not be undoable.

Designing and Implementing with Undo in Mind

The manner in which responsibilities are divided between the model and the controller and the public methods that are implemented can affect the ease of undo operations. Since our `Line` object is created in the controller, it is easy to store this in the command object and then use the reference to remove the object when undoing. If the model took the end points and invoked the constructor, we would need some additional machinery to implement undo. Likewise, our model has a method to remove a specified item, which is effectively an 'undo' of the operation that adds

an item. If the methods invoked by the command object on other subsystems cannot be easily reversed, it may not be feasible to undo the operation.

11.11.5 Synchronising Updates

We have already alluded to the problem that could occur when multiple views concurrently update the `UIContext` field in the model. This is a well known problem in operating systems and the reader is referred to standard texts in the field [5] for a detailed description.

One possible solution is to use binary semaphores. For this, we first create the following class.

```
public class Synchroniser {
  private boolean drawing;
  public synchronised void beginDrawing() {
    try {
      while (drawing) {
        wait();
      }
    } catch (InterruptedException ie) {
    }
    drawing = true;
  }
  public synchronised void endDrawing() {
    drawing = false;
    notifyAll();
  }
}
```

Assume that the class is made a singleton. When the view is ready to start drawing, which would be at the very beginning of the `paintComponent` method in our example code, it invokes the `beginDrawing` method. After completing the drawing, that is, just before leaving the `paintComponent` method in our case, the view invokes `endDrawing`. The `beginDrawing` and `endDrawing` methods together ensures several desirable properties, including the following: at most one view is painting at any given time and every view gets a chance to paint, eventually.

Another solution employs monitors. Please see Silberschatz [5] for a description.

Projects

1. *Creating a simple spreadsheet.* The sheet will display a simple grid and allow for data and formulae to be entered into the boxes. The following features will be available:
   - Allow for a column to be widened. This will be done by user selecting a column and activating the operation from the menu
   - Automatic evaluation and re-evaluation of formulae
   - Drawing a graph using data from two columns

2. Implement the drawing program described in this chapter using the *Document–View* architecture. Implement each command as a singleton that keeps its own stack. What pros and cons do you see for this approach?
3. Create a simple graphical toy that consists of a circle, triangles and rectangles. All these shapes will be filled, and represent a 2-dimensional ball and 2-dimensional triangular and rectangular blocks. A menu will allow the user to create new blocks, change the colour of an existing shape, move the shapes, increase the size of the ball, rotate the blocks or drop the ball. When the ball is dropped, it will fall vertically and thereafter behave in accordance with the idealised laws of physics, with a coefficient of restitution of 0.5 (half the kinetic energy is lost whenever the ball collides with a block or a boundary). The blocks do not move when hit by the ball. There will be a designated threshold so that when the ball's velocity drops below this threshold, it will be assumed to have stopped.