MODULE-4

Interactive systems and the MVC architecture:

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,
- 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.

- 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.**

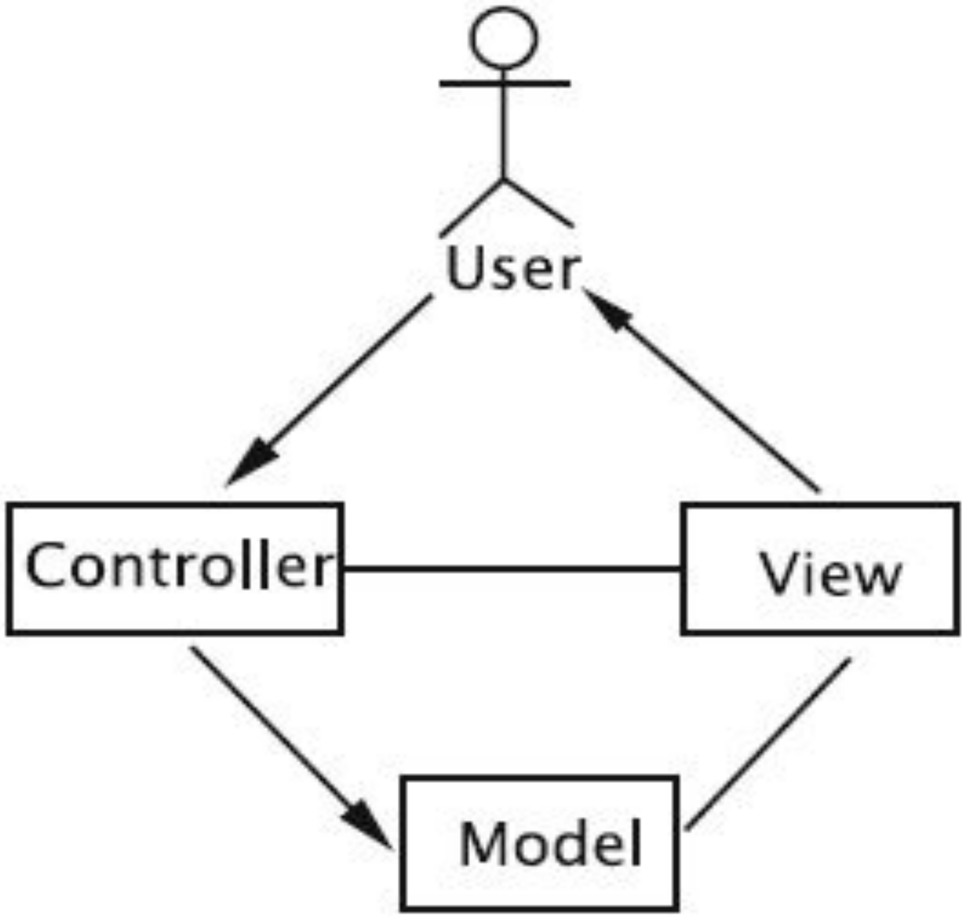
- In this chapter, we start by describing a well-known software architecture **(sometimes referred to as an architectural pattern**) called the In this chapter, we start by describing a well-known software architecture (sometimes referred to as an **architectural pattern) called the Model–View–Controller**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

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 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–view–controller architecture



- 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 application is housed

- 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

An alternate view of the the MVC architectur

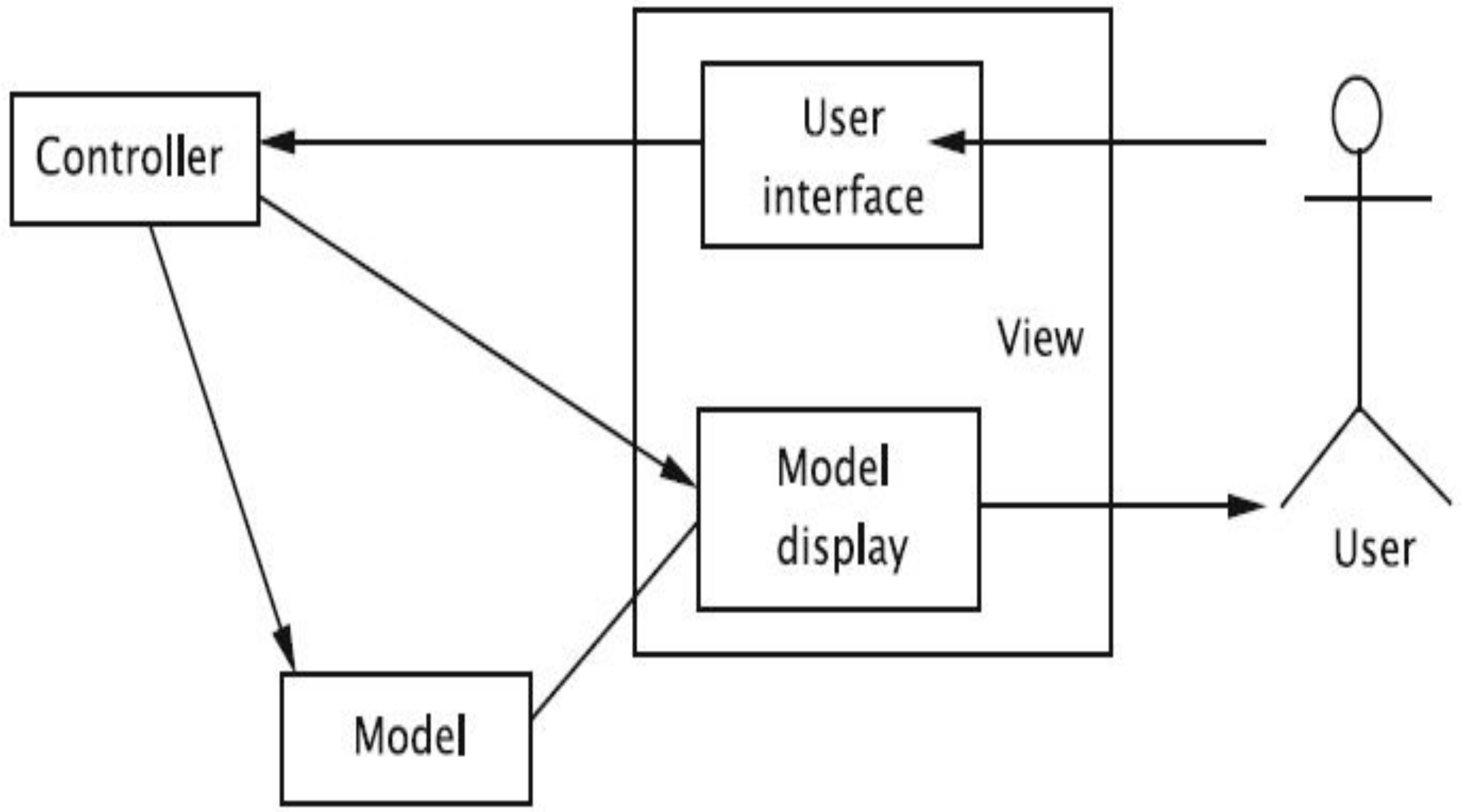


Fig. 11.2 An alternate view of the the MVC architecture

- 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.
- 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.

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.

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.

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.

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.

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
- 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.

- 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 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 amouse-click.

Defining the Use Cases

We can now write the detailed use cases for each operation. The first one, for drawing a line,

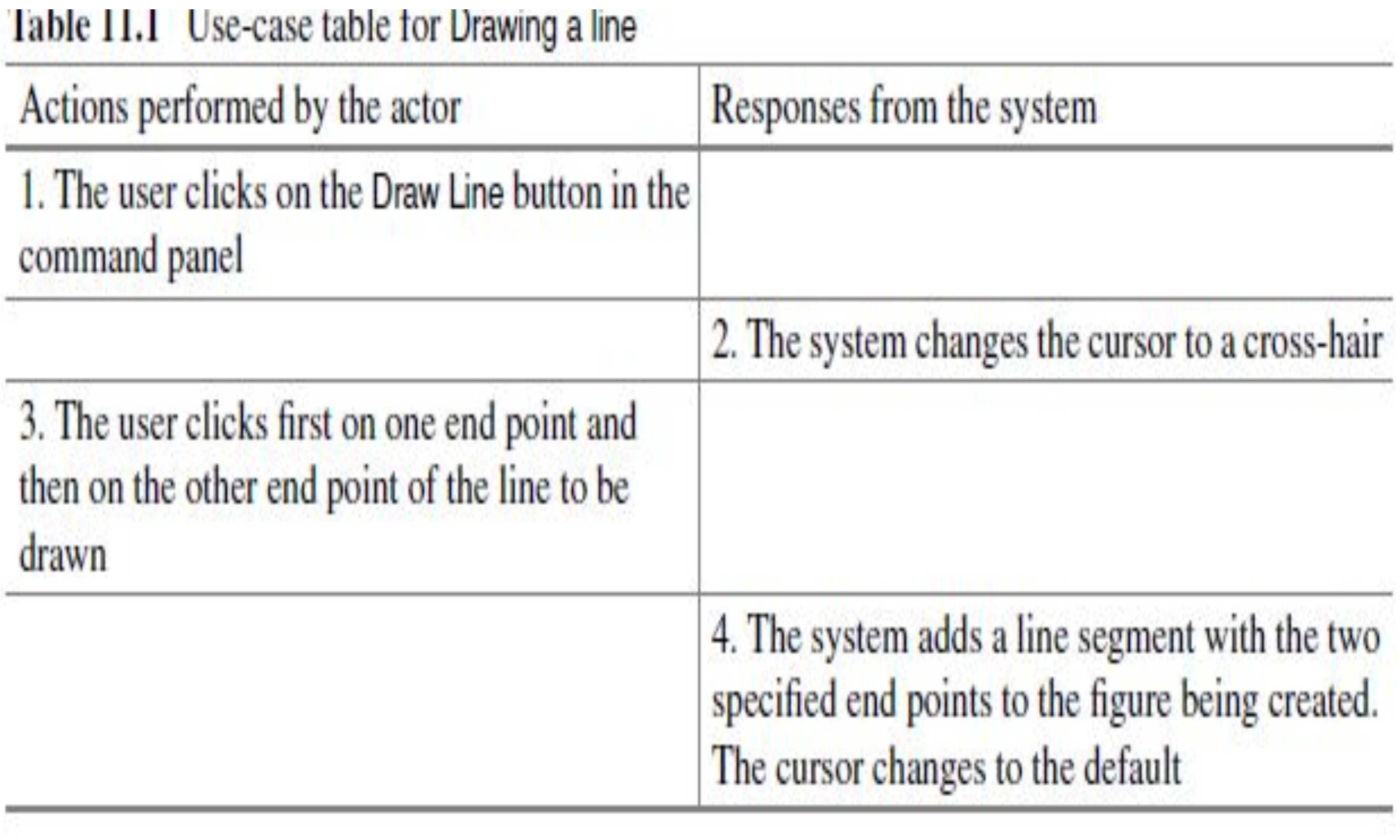


Table 11.1 Use-case table for Drawing a line

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

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

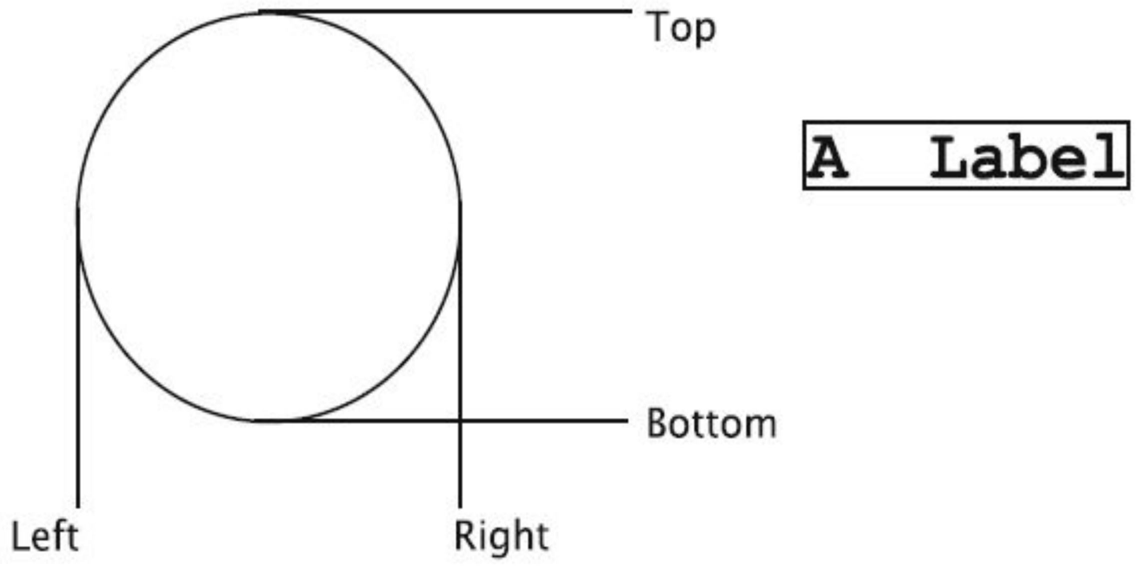
Actions performed by the actor	Responses from the system
1. The user clicks on the Select button in the command panel	
	2. The system changes the display to the *selection mode*
3. The user clicks the mouse on the drawing	
	4. If the click falls on an item, the system adds the item to its collection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

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.

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. E
- 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



. **11.3** Representing a circle and a label

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.

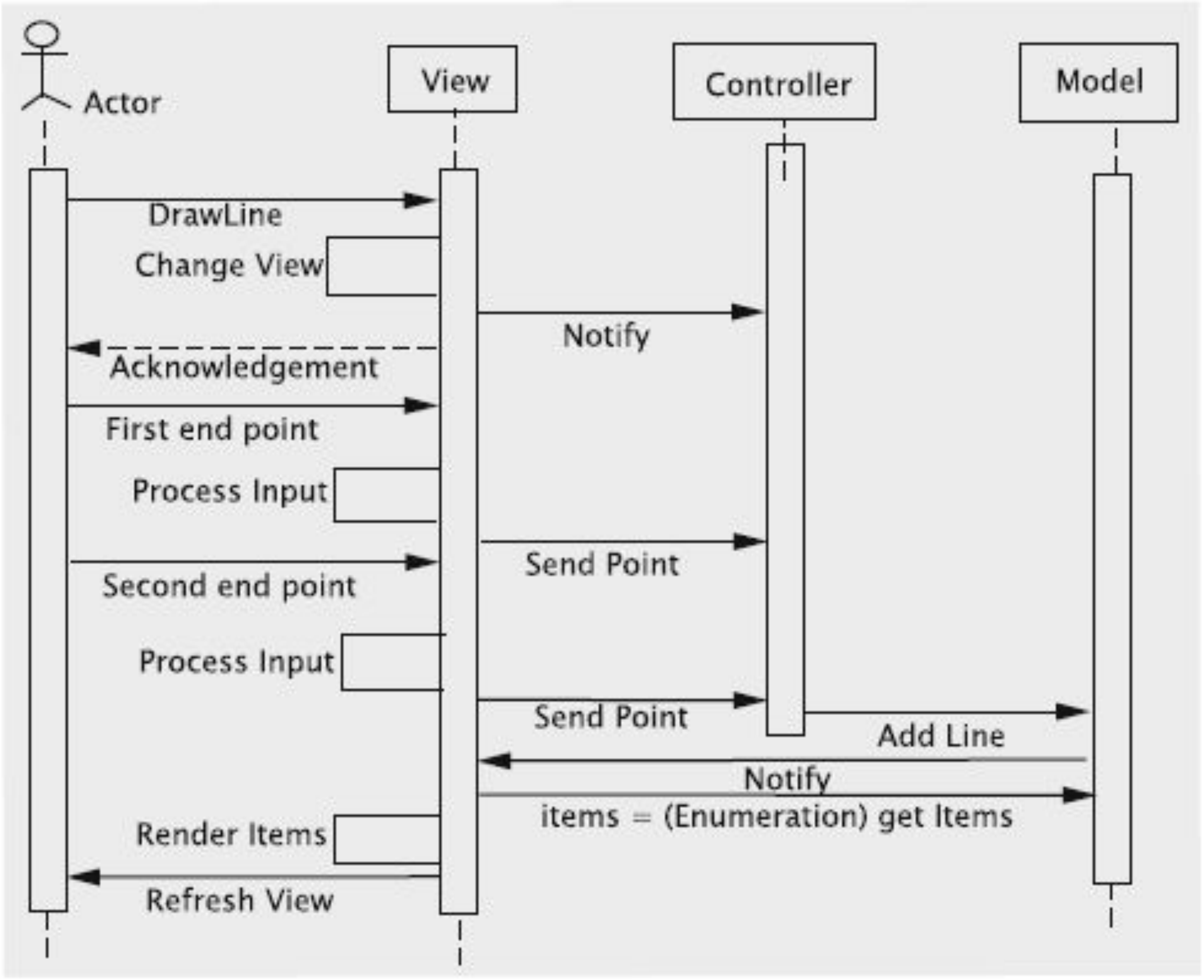


Fig. 11.4 Sequence of operations for drawing a line

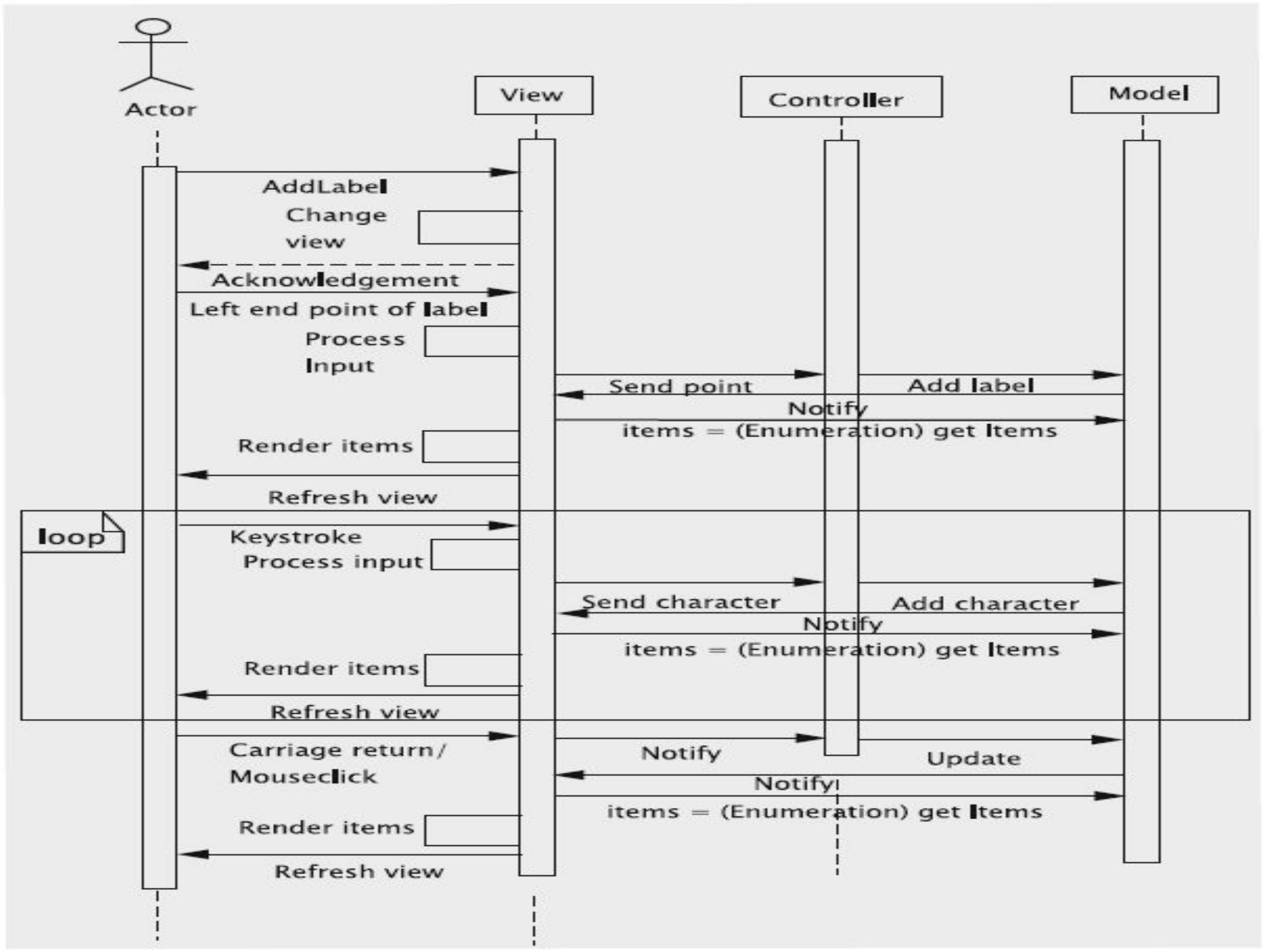


Fig. 11.5 Sequence of operations for adding a label

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.

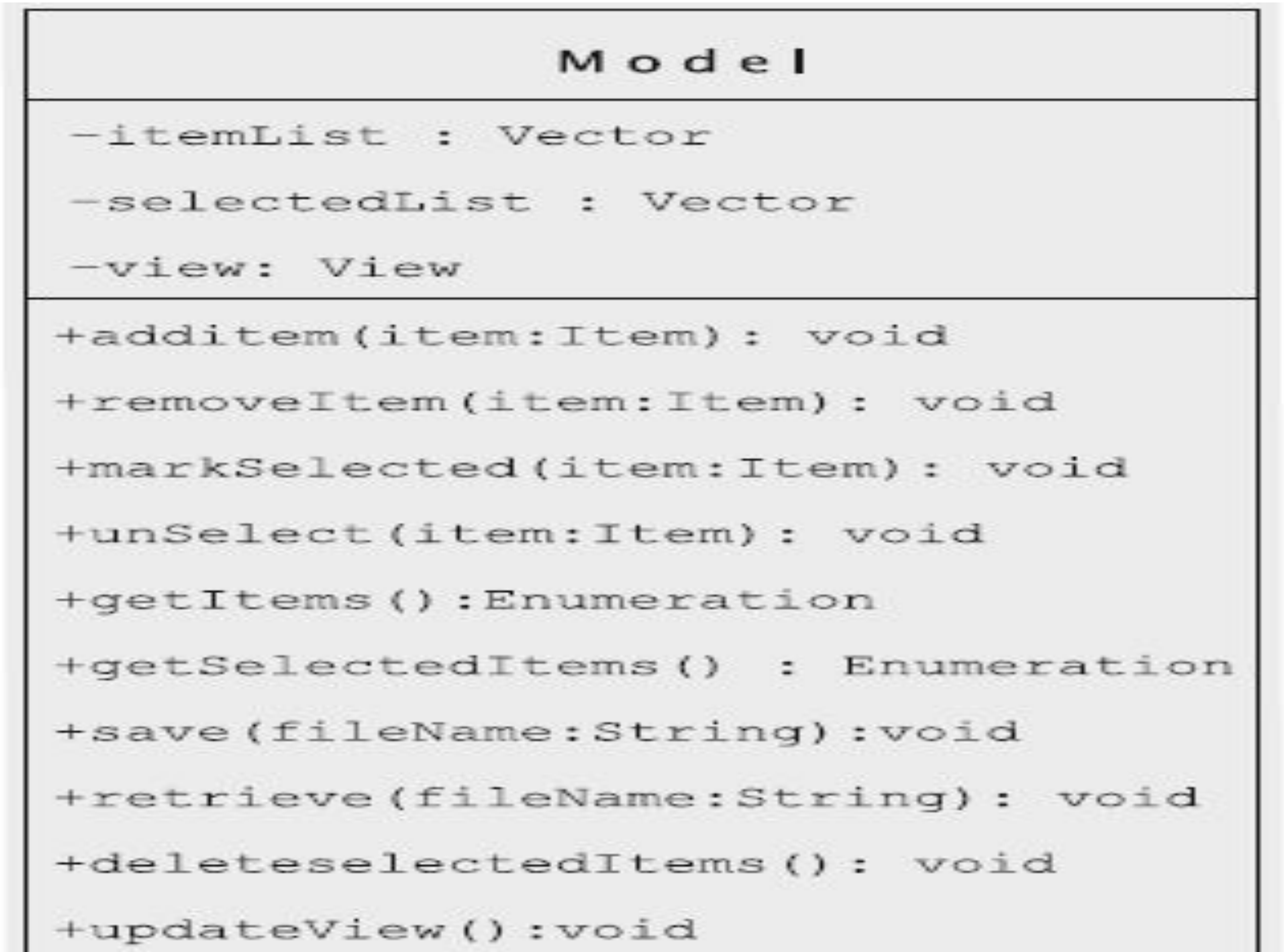
- **Design of the Subsystems**
- The next step of the process is to design the individual subsystems. In this stage, the classes and their responsibilities are identified and we get a more detailed picture of how the required functionality is to be achieved.
- Since the model should remain independent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

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

Class diagram for model



```
Model
-itemList : Vector
-selectedList : Vector
-view: View
+additem(item:Item): void
+removeItem(item:Item): void
+markSelected(item:Item): void
+unSelect(item:Item): void
+getItems():Enumeration
+getSelectedItems() : Enumeration
+save(fileName:String):void
+retrieve(fileName:String): void
+deleteselectedItems(): void
+updateView():void
```

Design of item and subclass

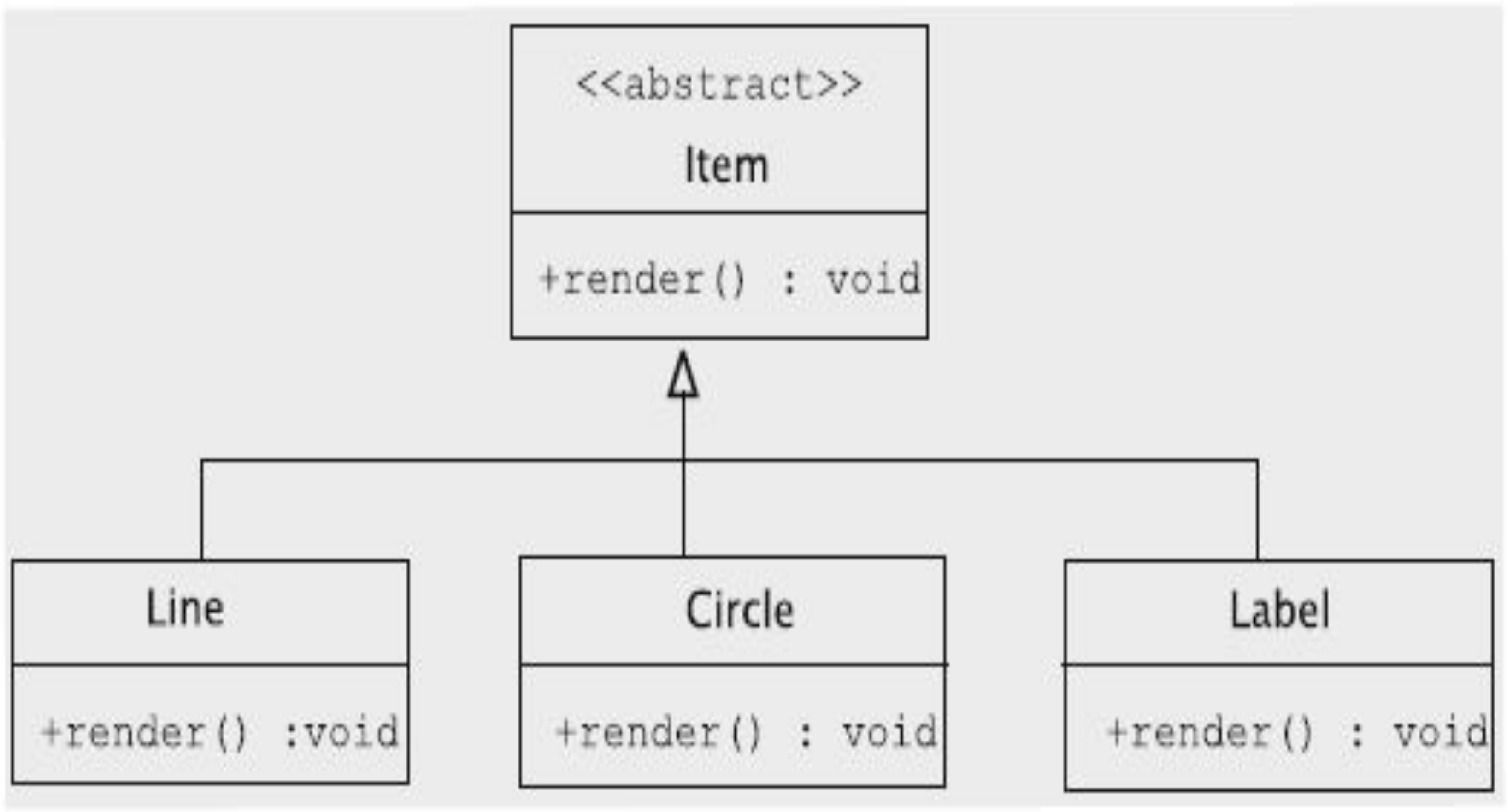


Fig. 11.7 The item class and its subclasses

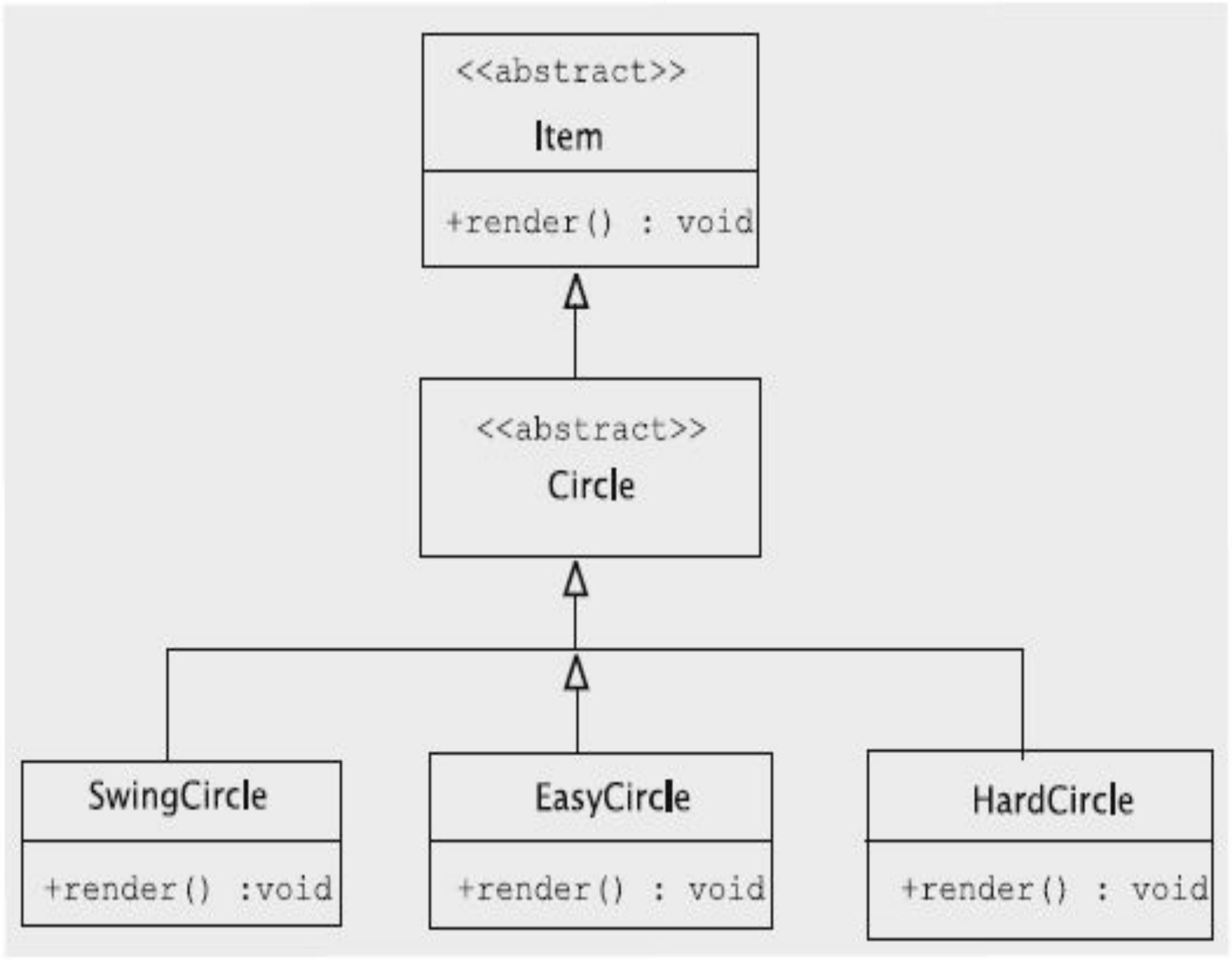


Fig. 11.8 Catering to multiple UI technologies

Design of the controller subsystem

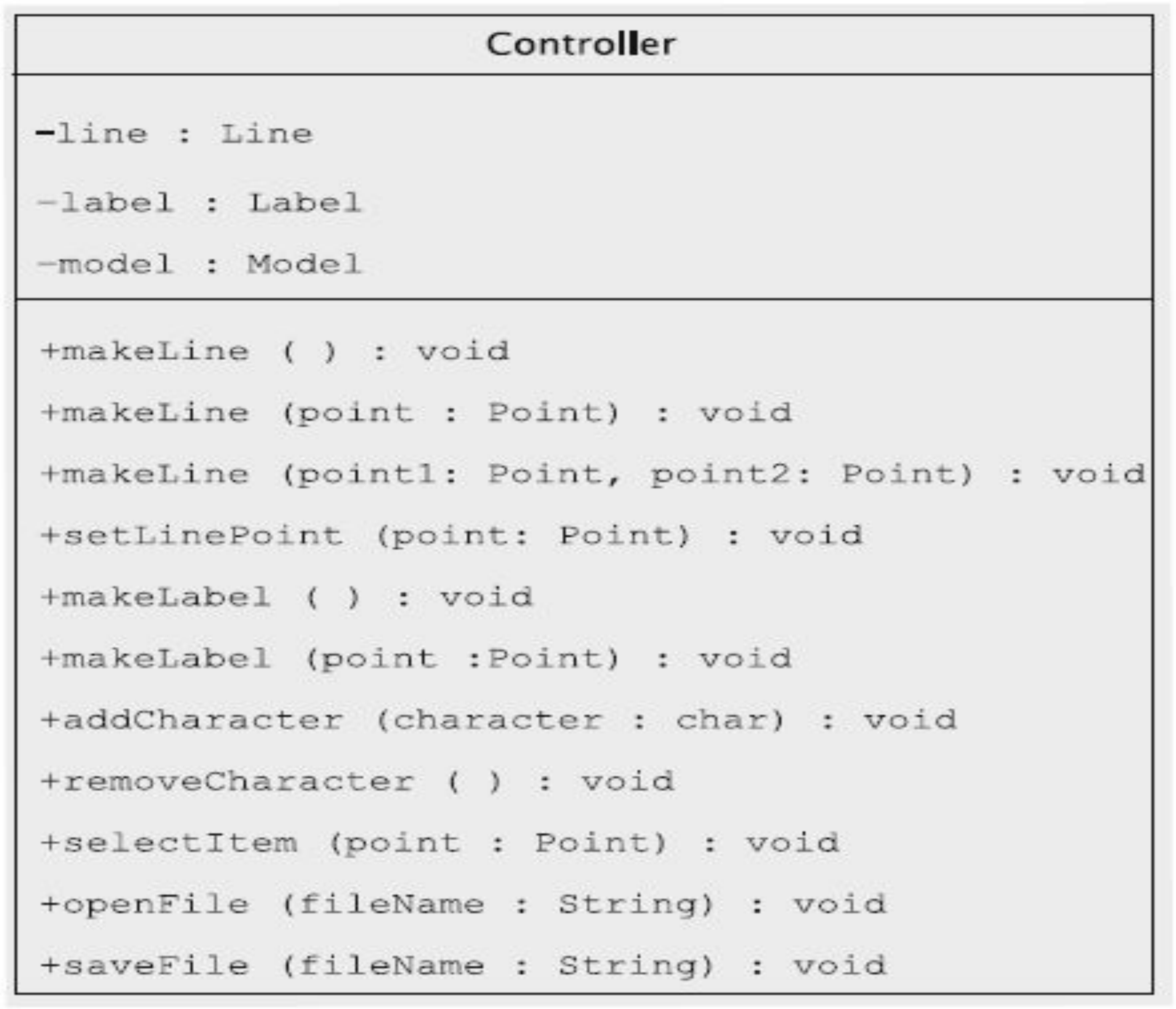

Controller
-line : Line
-label : Label
-model : Model
+makeLine ( ) : void
+makeLine (point : Point) : void
+makeLine (point1: Point, point2: Point) : void
+setLinePoint (point: Point) : void
+makeLabel ( ) : void
+makeLabel (point :Point) : void
+addCharacter (character : char) : void
+removeCharacter ( ) : void
+selectItem (point : Point) : void
+openFile (fileName : String) : void
+saveFile (fileName : String) : void

Fig. 11.12 Controller class diagram

Design of the View Subsystem

- The separation of concerns inherent in theMVC 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.

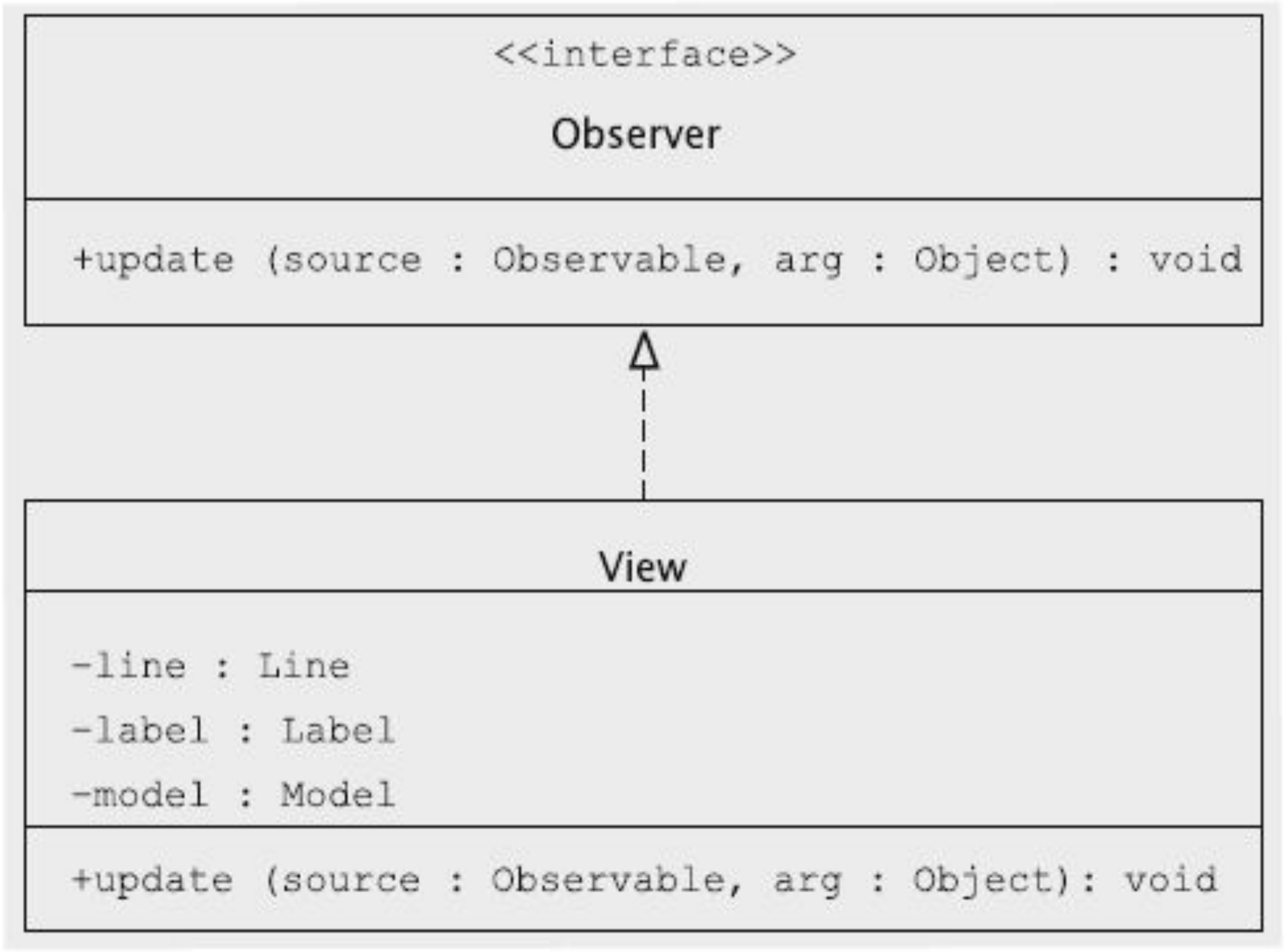

<<interface>>
Observer
+update (source : Observable, arg : Object) : void
View
-line : Line
-label : Label
-model : Model
+update (source : Observable, arg : Object): void

Fig. 11.13 Basic structure of the view class

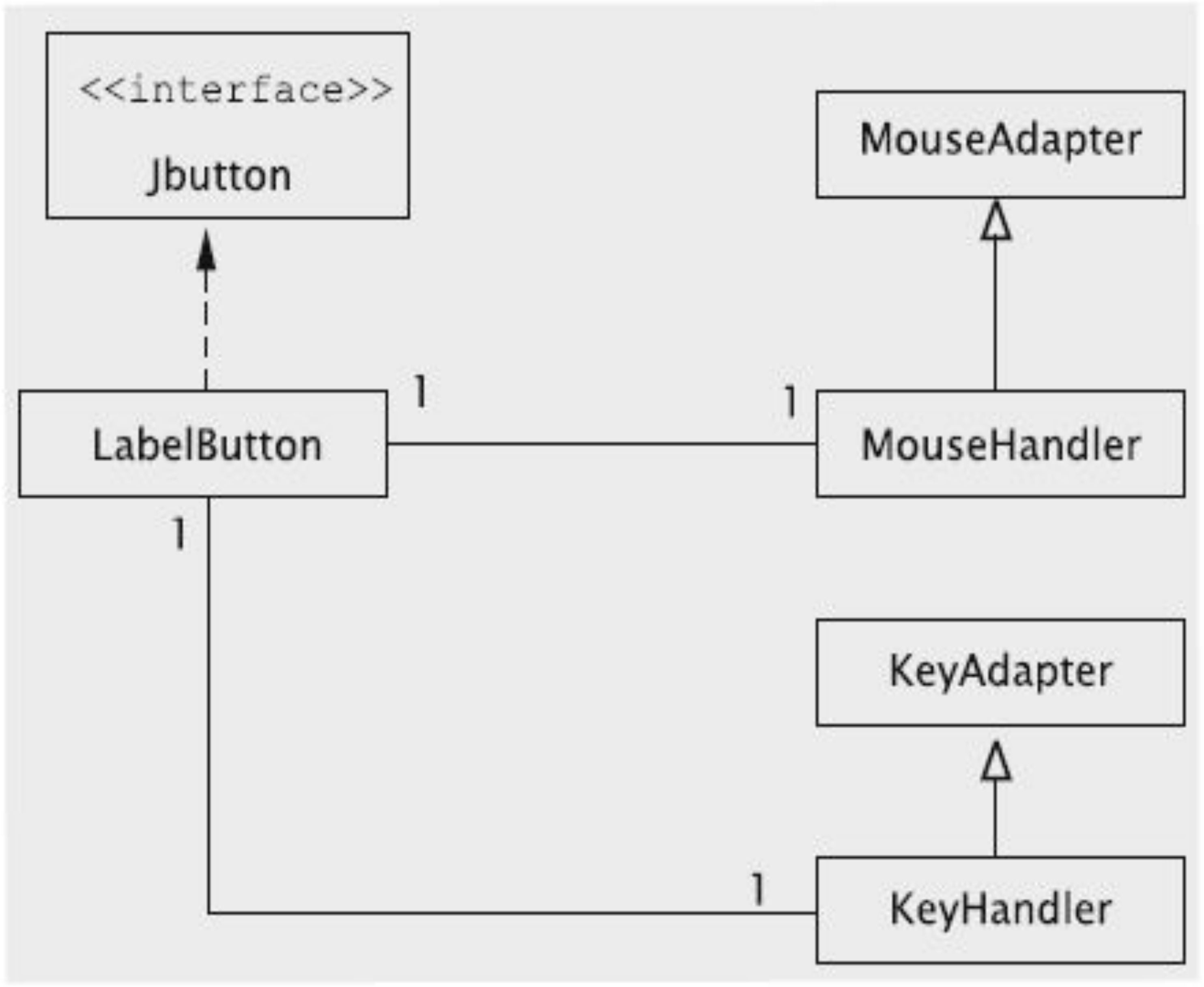


Fig. 11.14 Organisation of the classes to add labels

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);
  }
}
```

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);
}
```

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);
}
```

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.

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();
  }
}
```

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
- 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,
- ot all things are undoable This can happen for two reasons. Some operations like ‘print file’ are irreversible, and hence undoable.
- Blocking further undo/redo operations It is easy to see that uncontrolled undo and redo can result in meaningless requests.
- 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.

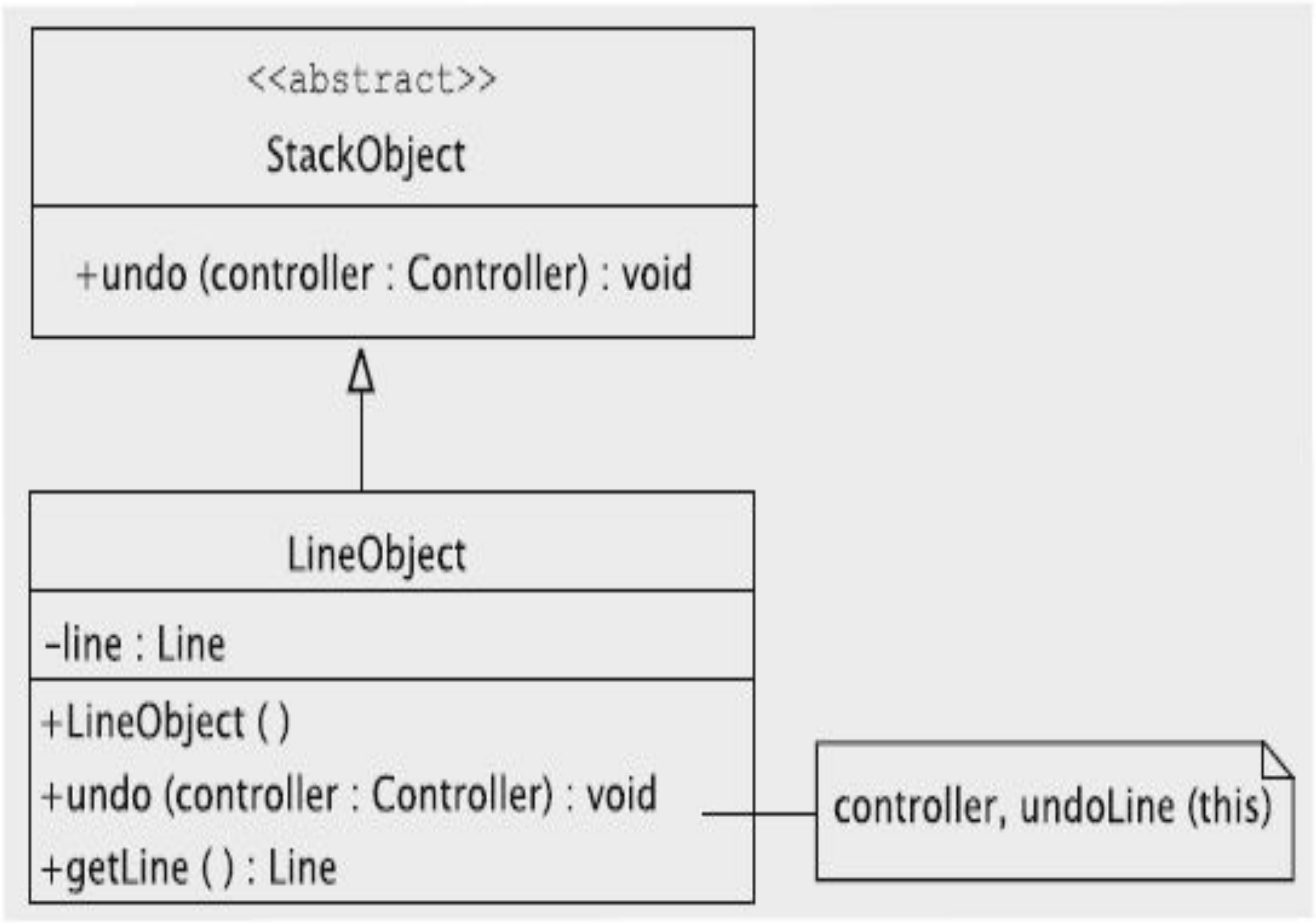
```
public class LineObject {
private Line line;
public Line getLine() {
return line;
}
public LineObject(Line line) {
this.line = line;
}
}
```

- When the operation for adding a line is completed, the appropriate Stack Object 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)));
- }

- 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);
}
}
```



<<abstract>>
StackObject
+undo (controller : Controller) : void
LineObject
-line : Line
+LineObject ( )
+undo (controller : Controller) : void
+getLine ( ) : Line
controller, undoLine (this)

Fig. 11.15 Representing the drawing of a line

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.

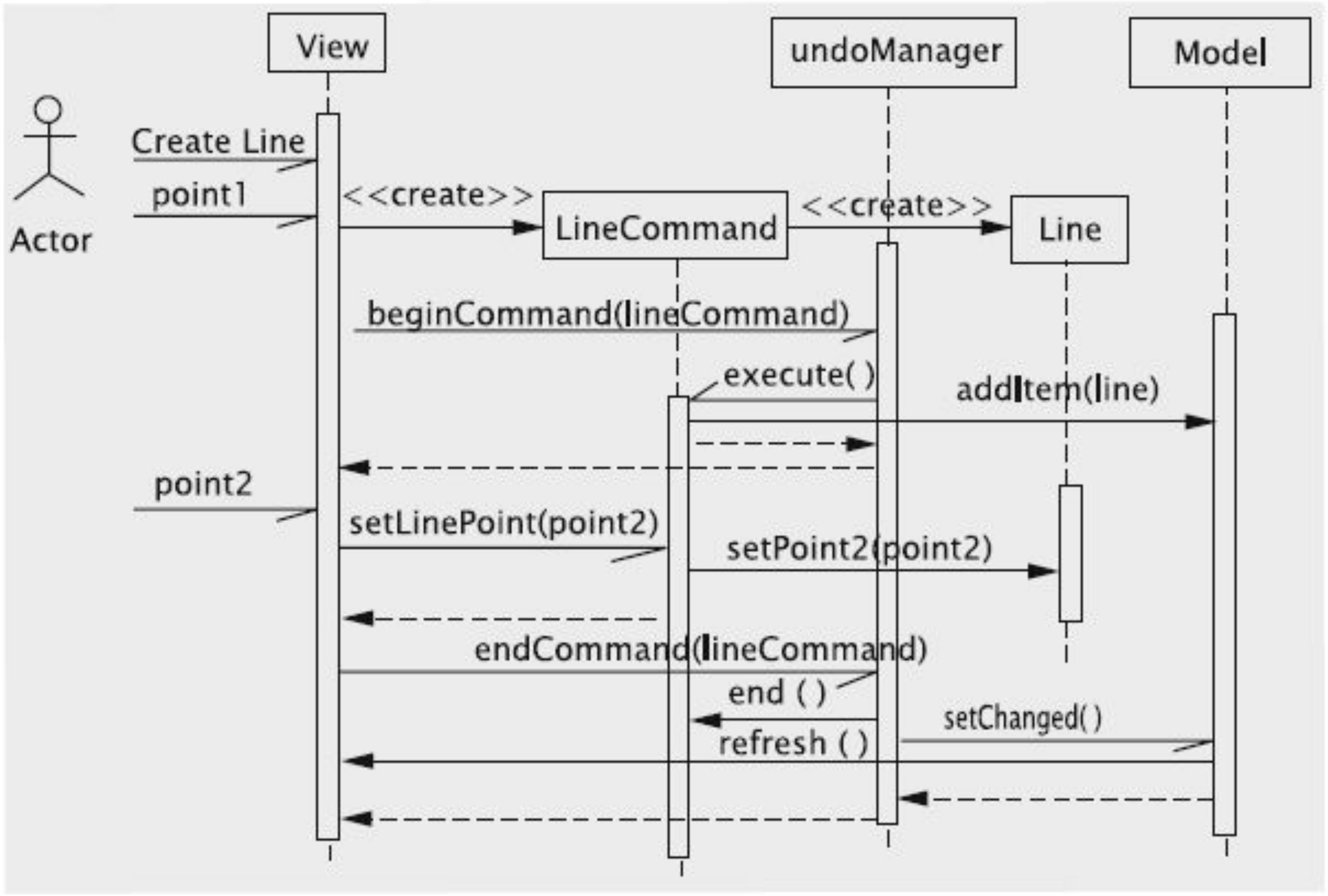


Fig. 11.17 Sequence diagram for adding a line

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.
- 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.

- 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;
} } }
```

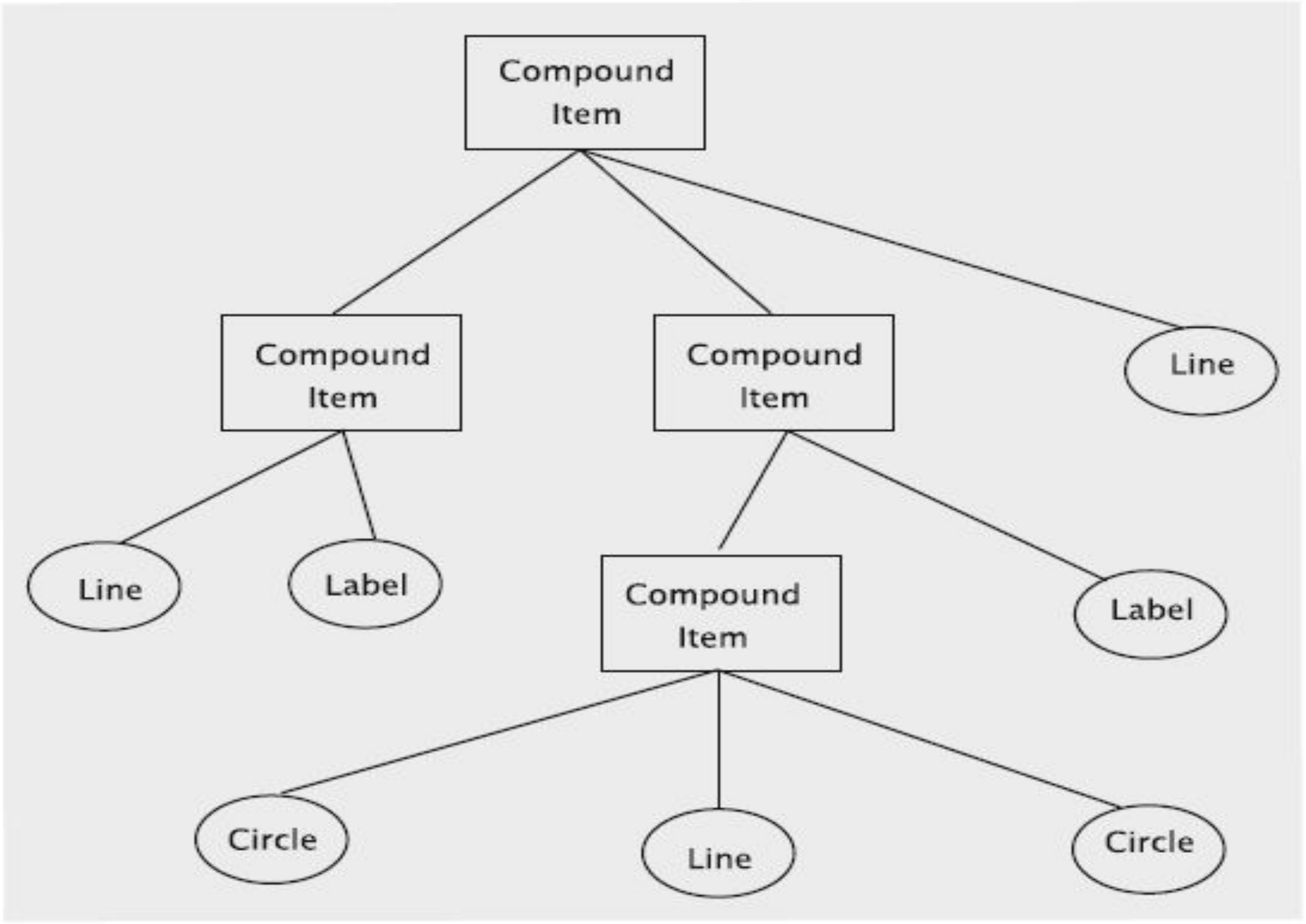
Adding a New Feature

- 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.
  I
- 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.*

Since we have to store a collection of items, an obvious approach to implementing thiswould be to create a newkind 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
}
```



ig. 11.20 Tree structure formed by compound items

- 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 Compound Item 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.
- }

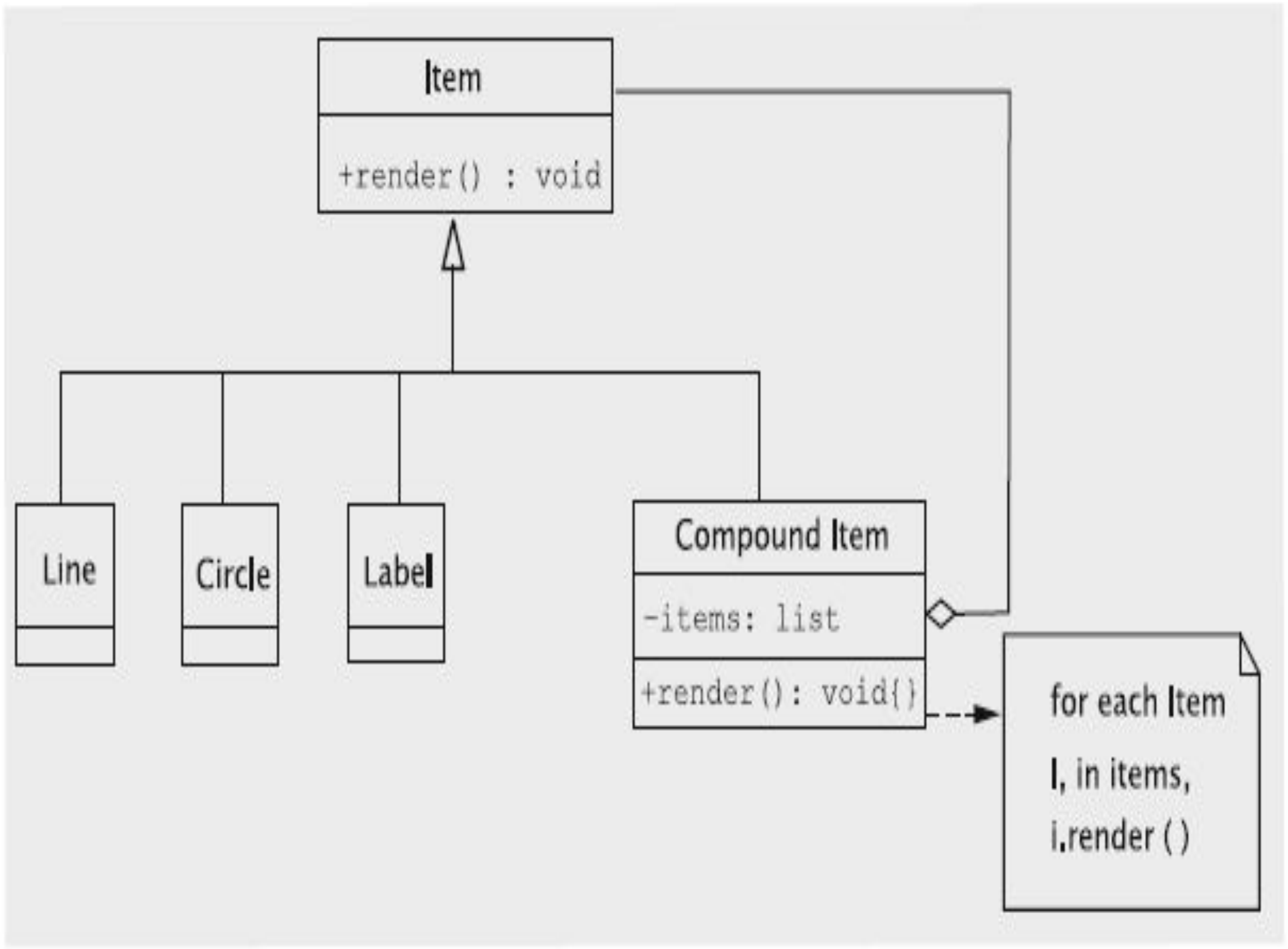


Fig. 11.21 Composite structure of the item hierarchy

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.)

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.

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.

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.

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

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 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.

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.