Chapter 6

Implementing Graphical User Interfaces

6.1 Introduction

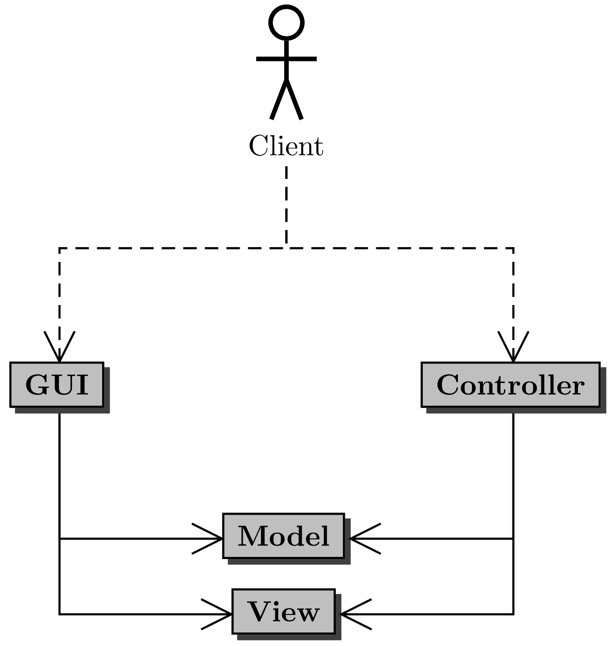
To see aggregation and inheritance in action, we implement a graphical user interface (GUI for short). This chapter is *not* about GUIs, but we do introduce some of the concepts that play a role in the implementation of GUIs. The focus of this chapter is aggregation and inheritance.

To implement a GUI, we exploit the model-view-controller (MVC) pattern. This pattern decouples the data, the graphical representation of the data, and the interactions of the client with the data. More precisely, the MVC pattern is based on the following three elements.

- The *model* represents the data and provides ways to manipulate the data.
- The *view* provides a graphical representation of the model on the screen.
- The *controller* translates the client's interactions with the view into actions that manipulate the view and the model. These interactions can be menu selections, button clicks, etc.

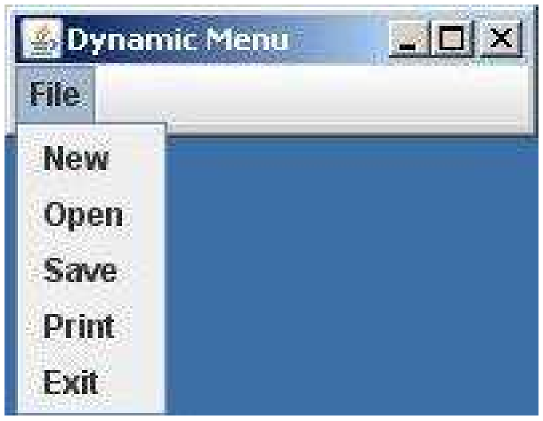
In our implementation, we introduce the classes `Model`, `View` and `Controller`. Furthermore, we develop an app, named `GUI`, which is launched by the client to run the GUI.

The client launches the app `GUI`. The app creates a `Model` and a `View`. Subsequently, the `View` creates a `Controller`. The interactions of the client with the GUI via the `View` are handled by the `Controller`. The `Controller` subsequently updates the `Model` and the `View`.

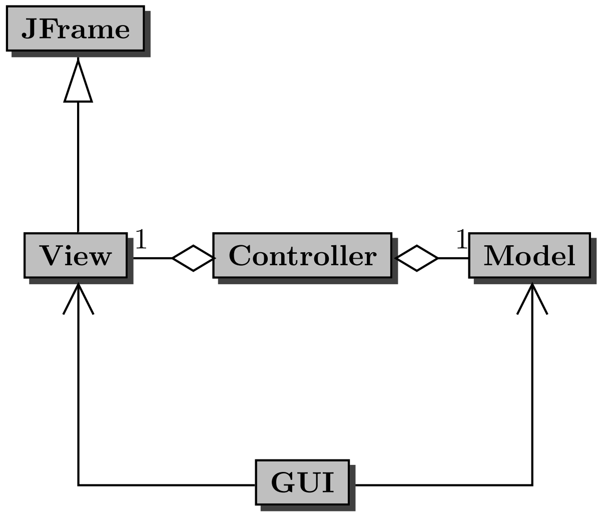


Since the `Controller` updates the `Model` and the `View`, it needs access to these objects. As a consequence, we will design the `Controller` class in such a way that it has a `Model` and it also has a `View`.

A GUI usually consists of a window with a title and a border. Furthermore, the window may contain components such as a menu bar, buttons, etcetera. A screenshot of the GUI we will develop can be found below.



By extending the `JFrame` class, which is part of the package `javax.swing`, our `View` class inherits more than three hundred methods from `JFrame` and its superclasses and more than one hundred attributes of `JFrame` and its superclasses are part of the state of a `View`.



We will see more examples of aggregation and inheritance when we implement the classes `Model`, `View` and `Controller`.

6.2 The Constructor Sections

Before implementing the classes `Model`, `View` and `Controller`, we first focus on their constructors. As we already mentioned, in the `GUI` app we create instances of the `Model` and `View` classes.

```
Model model = new Model(...);
View view = new View(...);
```

As we also mentioned, an instance of the `Controller` class is created in (the constructor of) the `View` class.

```
public View(...)
{
   Controller controller = new Controller(...);
}
```

Since the `Controller` has a `Model` and has a `View`, the `Controller` class has attributes of type `Model` and `View`.

```
private Model model;
private View view;
```

These attributes are initialized in the constructor of the `Controller` class.

```
public Controller(Model model, View view)
{
   this.setModel(model);
   this.setView(view);
}
```

Note that we have to provide two arguments to the constructor of the `Controller` class in line 3 of the constructor of the `View` class, one of type `Model` and one of type `View`. The former can be passed as an argument to the constructor of the `View` class, whereas the latter is the object on which the constructor is invoked. This leads to the following constructor of the `View` class.

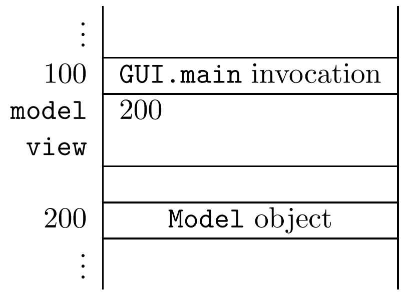
```
1 public View(Model model)
2 {
3    Controller controller = new Controller(model, this);
4 }
```

Now, we can return to the `GUI` app and fill in the arguments of the constructors.

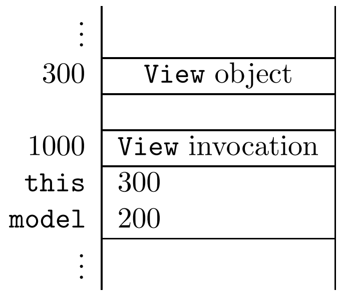
```
1 Model model = new Model();
2 View view = new View(model);
```

In summary, the `GUI` app first creates a `Model` and passes a reference to that `Model` to the constructor of the `View` class. In the constructor of the `View` class, that reference to the `Model` is passed on to the constructor of the `Controller` class together with a reference to the created `View`. Hence, the constructor of the `Controller` class receives references to the created `Model` and `View`.

Once the execution reaches the end of line 1 of the `GUI` app, memory can be depicted as follows.



When executing line 2 of the `GUI` app, a memory block for the `View` object is allocated and the constructor of the `View` class is invoked.



When executing the constructor of the `View` class, a memory block for the `Controller` object is allocated and the constructor of the `Controller` class is invoked.

⋮	
400	`Controller` object
`model`	
`view`	
1100	`Controller` invocation
`this`	400
`model`	200
`view`	300
⋮	

Once we reach the end of the constructor of the `Controller` class, the `Controller` object can be depicted as follows.

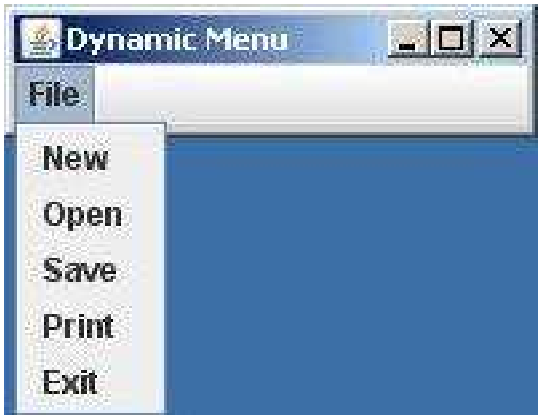
⋮	
400	`Controller` object
`model`	200
`view`	300
⋮	

Once we reach the end of line 2 of the `GUI` app, its invocation block can be depicted as follows.

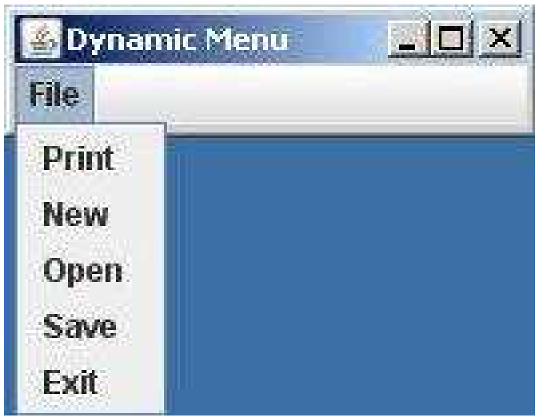
⋮	
100	`GUI.main` invocation
`model`	200
`view`	300
⋮	

6.3 Dynamic Menu

We start with a rather simple GUI. The view consists of a menu bar. The menu bar has a single menu. The items of the menu are ordered in such a way that the most recently selected item is at the top of the menu. As a consequence, the order of the items may change as items are selected by the client. Assume that initially, when no items have been selected yet, the items are ordered as follows.



After the client has selected the Print item, the items are ordered as follows.



Below, we discuss the `Model`, `View`, and `Controller` classes and the `GUI` app.

6.3.1 The `Model` Class

The model contains the data of the GUI. In this case, we keep track of the order of the items of the menu. Furthermore, we provide the `Controller` with the method

```
public void select(String title)
```

to update the model when the item with the given `title` has been selected by the client. The method

```
public List<String> getTitles()
```

provides the `Controller` with the list of titles of the items of the menu. The complete API of the `Model` class can be found following this link.

The Attributes Section

To represent the order of the items of the menu, we use a list of the titles of the items. Hence, we introduce the following attribute.

```
private List<String> titles;
```

As class invariant, we introduce the following.

```
this.titles != null && this.titles contains no duplicates
```

The Constructors Section

The `Model` class only contains a one parameter constructor. In this constructor, we initialize the attribute `titles`.

```
public Model(List<String> titles)
{
   this.setTitles(titles);
}
```

The Methods Section

The accessor and mutator are implemented in the usual way. Note that only the accessor is public so that the `Controller` can invoke this method.

We have left to implement the `select` method. We have to move the given `title` to the beginning of the list. This can be accomplished by first removing the given `title` and by next inserting the given `title` at the beginning of the list.

```
public void select(String title)
{
   this.getTitles().remove(title);
   this.getTitles().add(0, title);
}
```

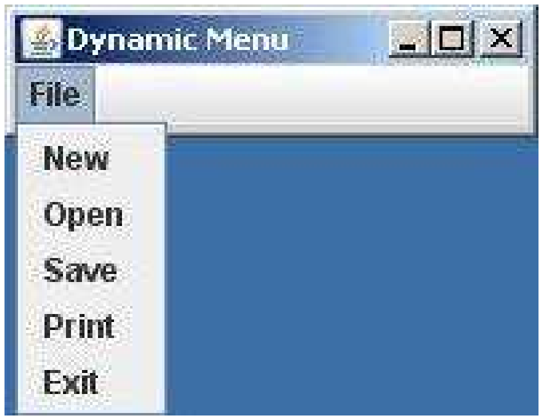
In the following snippet of client code, we create a model for our GUI.

```
List<String> titles = new LinkedList<String>();
titles.add("New");
titles.add("Open");
titles.add("Save");
titles.add("Print");
titles.add("Exit");
Model model = new Model(titles);
```

Once we have implemented the `Model` class, we can test its constructor and methods in the usual way. The code of the `Model` class can be found by following this link.

6.3.2 The `View` Class

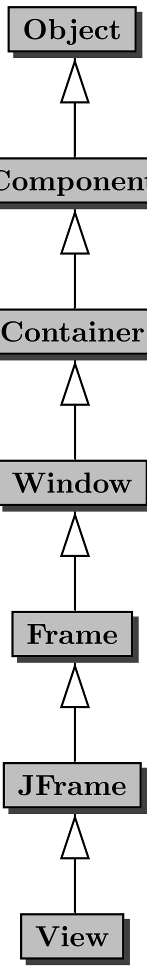
The view provides the graphical representation of the GUI. In this case, the GUI consists of a window with the title "Dynamic Menu" and a menu bar. The menu bar has a single menu, entitled "File". This menu has five items as shown below.



As we already mentioned in the introductory section of this chapter, the `View` extends `JFrame`. This is reflected in the class header as follows.

```
public class View extends JFrame
```

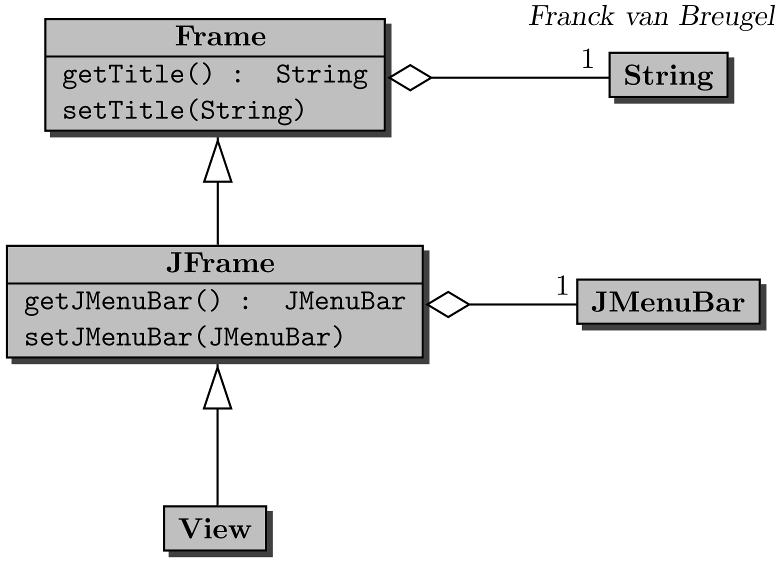
The corresponding inheritance hierarchy can be depicted as follows.



The Attributes Section

Our GUI has a title and a menu bar. However, the superclass `Frame` contains the attribute `title` of type `String` and the superclass `JFrame` contains an attribute `menuBar` of type `JMenuBar`, the accessor `getJMenuBar` and the mutator `setJMenuBar`.[1] Hence, the attributes `title` and `menuBar` are already part of the state of the `View` and the methods `getJMenuBar` and `setJMenuBar` are inherited by `View`. Therefore, we do not need to introduce any additional attributes.

[1]To be precise, the `JFrame` class contains the attribute `rootPane` of type `JRootPane` and the `JRootPane` class contains the attribute `menuBar` of type `JMenuBar`.

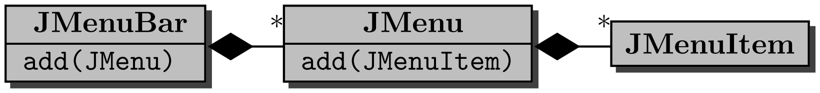


The Constructors Section

As we have already seen in Section 6.2, the constructor of the `View` class has the following skeleton.

```
public View(Model model)
{
   Controller controller = new Controller(this, model);
}
```

Since our `View` class extends the `JFrame` class, we start the constructor by delegating to a constructor of the `JFrame` class. We use `super("Dynamic Menu")` to set the title of the frame to "Dynamic Menu." We have left to create the menu bar. A menu bar has a collection of menus and each menu has a collection of items.



We create a menu bar as follows.

```
JMenuBar bar = new JMenuBar();
```

We use the mutator `setJMenuBar` to initialize the menu bar of the `View`. We create a menu entitled "File" as follows.

```
JMenu menu = new JMenu("File");
```

The menu can be added to the menu bar as follows.

```
bar.add(menu);
```

We create a menu item entitled "New" as follows.

```
JMenuItem item = new JMenuItem("New");
```

The item can be added to the menu as follows.

```
menu.add(item);
```

In our implementation of the constructor, we delegate to the method

```
public void setMenu(List<String> titles)
```

which ensures that the first menu of the menu bar has items with the given `titles` in the corresponding order. The titles of the items of the menu can be obtained using `model.getTitles()`. Combining the above, we arrive at the following implementation of the constructor.

```
1 public View(Model model)
2 {
3    super("Dynamic Menu");
4    Controller controller = new Controller(this, model);
5    JMenuBar bar = new JMenuBar();
6    this.setJMenuBar(bar);
7    bar.add(new JMenu("File"));
8    this.setMenu(model.getTitles());
9 }
```

The Methods Section

Recall that the `setMenu` method ensures that the first menu of the menu bar has items with the given list of titles in the corresponding order. This method can implemented as follows.

```
public void setMenu(List<String> titles)
{
   JMenu menu = this.getJMenuBar().getMenu(0);
   menu.removeAll();
   for (String title : titles)
   {
      JMenuItem item = new JMenuItem(title);
      menu.add(item);
   }
}
```

If we want to have a look at our `View`, we have to comment out line 4 of the constructor, since we have not yet implemented the `Controller`. The following snippet of client code creates a `Model` and a `View`.

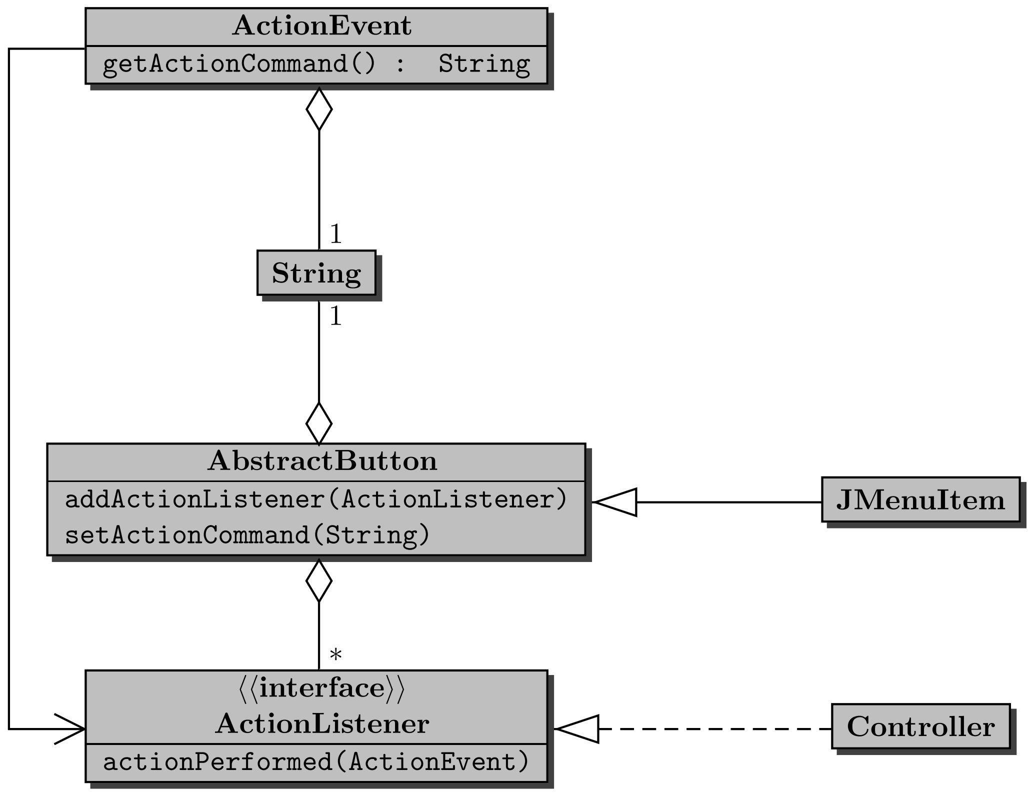
```
List<String> titles = new LinkedList<String>();
titles.add("New");
titles.add("Open");
titles.add("Save");
titles.add("Print");
```

```
6  titles.add("Exit");
7  Model model = new Model(titles);
8  View view = new View(model);
9  final int WIDTH = 200;
10 final int HEIGHT = 50;
11 view.setSize(WIDTH, HEIGHT);
12 view.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13 view.setVisible(true);
```

In line 11, the width and height (in the number of pixels) of the GUI is set using the method `setSize`, which is inherited from the `Window` class. In line 12, the behaviour of the GUI is defined when the box with the cross at the right upper corner is clicked. The method `setDefaultCloseOperation` is inherited from the `JFrame` class. The constant `JFrame.EXIT_ON_CLOSE` is used to specify that the GUI exits when the box with the cross at the right upper corner is clicked. The method `setVisible`, inherited from the `Window` class, makes the GUI visible.

6.3.3 Event-Driven Programming

Recall that the `Controller` translates the client's interactions, with the `View` into actions that manipulate the `View` and the `Model`. For our GUI, the client can interact with the `View` by selecting one of the items of the menu. The following class diagram provides an overview of the classes and methods that play a role.



Of the above classes we only have to implement the `Controller` class. The other classes are part of the Java standard library. We discuss the implementation of the `Controller` class in the next section. Below we discuss the additions that have to be made to the `View` class to signal the interactions of the client with the `View` to the `Controller`

Whenever the client interacts with the view (by, for example, selecting a menu item), the operating system and the Java virtual machine ensure that the `actionPerformed` method of the action listeners of the components (for example, of the menu item) are invoked. In our GUI, whenever the client selects a menu item, the operating system and the Java virtual machine ensure that the `actionPerformed` method of the `Controller` is invoked. From the above diagram we can conclude that an `AbstractButton` has `ActionListener`s and that a `JMenuItem` is an `AbstractButton` and, hence, also has `ActionListener`s. Furthermore, we can also deduce from the above diagram that the `Controller` class implements the `ActionListener` interface and, hence, a `Controller` is an `ActionListener`.

According to the above diagram, an `AbstractButton` (and, hence, a `JMenuItem`) and an `ActionEvent` have an action command, which is of type `String`. As we will see, this action command provides a link between the `Controller` and the `View`.

As we already mentioned above, whenever the client interacts with the view (by, for example, selecting a menu item), the operating system and the Java virtual machine ensure that the `actionPerformed` method of the action listeners of the components (for example, of the selected menu item) are invoked. Each invocation of the `actionPerformed` method is provided with a single argument of type `ActionEvent`. The Java virtual machine ensures that the action command of this `ActionEvent` is the same as the action command of the component (for example, of the menu item). In our GUI, whenever the client selects a menu item, the operating system and the Java virtual machine ensure that the `actionPerformed` method of the `Controller` is invoked with an `ActionEvent`, whose action command is the action command of the menu item, as its argument.

Within the `View` class, we have to

- set the action command of each menu item, and
- add the `Controller` as an `ActionListener` of each menu item.

In our GUI, we use the title of each menu item as its action command. We set the action command of each menu item using the mutator `setActionCommand`. Hence, we add to the `setMenu` method the following.

```
item.setActionCommand(title);
```

In the `setMenu` method we also add an `ActionListener` to each menu item, by adding a parameter named `actionListener` of type `ActionListener` to the signature of the `setMenu` method and by adding the following to the body of this method.

```
item.addActionListener(actionListener);
```

Hence, we arrive at the following `setMenu` method.

```
public void setMenu(List<String> titles, ActionListener actionListener)
{
   JMenu menu = this.getJMenuBar().getMenu(0);
   menu.removeAll();
   for (String title : titles)
   {
      JMenuItem item = new JMenuItem(title);
      item.setActionCommand(title);
      item.addActionListener(actionListener);
      menu.add(item);
   }
}
```

Recall that we invoke the `setMenu` method in the constructor of the `View` class. Since we have changed the signature of the `setMenu` method, by adding a parameter of type `ActionListener`, we also have to modify the invocation of the `setMenu` method. We replace line 8 of the constructor with

```
this.setMenu(model.getTitles(), controller);
```

The API of the `View` class can be found by following this link and the code of the `View` class can be found by following this link.

6.3.4 The `Controller` Class

As we already saw in the above class diagram, the `Controller` class implements the `ActionListener` interface (so that we can use the `Controller` as the action listener of each menu item). This is reflected by the following class header.

```
public class Controller implements ActionListener
```

In Section 6.2 we already discussed the attributes and the constructor of the `Controller` class. We have left to introduce the methods of this class. Apart from the usual accessors and mutators, the `Controller` class contains a single method, namely the single method of the `ActionListener` interface: `actionPerformed`.

Whenever the client selects a menu item, the `Controller` has to modify the `Model` and `View`. In particular, the `Controller` has to invoke the `select` method on the `Model` with the title of the selected item as its argument. Furthermore, the `Controller` has to invoke the `setMenu` method on the `View`. The latter invocation takes two arguments: the list of titles of the menu items and the action listener to be associated with the menu items. The former information can be obtained from the `Model` using the `getTitles` accessor. The latter is the `Controller` itself.

As we already mentioned above, the `actionPerformed` method takes an `ActionEvent` object as its single argument. For our GUI, this `ActionEvent` contains the title of the menu item that has been selected as its action command. Hence, the `actionPerformed` method can be implemented as follows.

```
public void actionPerformed(ActionEvent event)
{
   this.getModel().select(event.getActionCommand());
   this.getView().setMenu(this.getModel().getTitles(), this);
}
```

The API of the `Controller` class can be found by following this link and the code of the `Controller` class can be found by following this link.

Next, we show how the `actionPerformed` method of the `Controller` class can be invoked and, hence, be tested in an app. As we have seen above, the `actionPerformed` method takes an `ActionEvent` as its single argument. To invoke the `actionPerformed` method, we need to create an `ActionEvent` object. According to the API of the `ActionEvent` class, its simplest constructor takes the following three arguments:

- `source` – the component that originated the event;
- `id` – an integer that identifies the event;
- `command` – a string that specifies the action command associated with the event.

Since we create the `ActionEvent` in the app, it has no originating component (such as a menu item). However, we cannot use null for the `source`, because the constructor throws an exception in that case. Hence, we provide a generic `Object` as the first argument. The API provides a number of constants that can be used for the second argument. We use the constant `ActionEvent.ACTION_PERFORMED` which indicates that a meaningful action occured. The third and final argument is the action command. For our GUI, the action command is the title of one of the menu items. Hence, we can create an `ActionEvent` object as follows.

```
new ActionEvent(new Object(), ActionEvent.ACTION_PERFORMED, title)
```

In the app below, we create a `Model`, `View` and `Controller` in the usual way. Then we print the list of titles, randomly select a title, and print the selected title. Now we can invoke the `actionPerformed` method, mimicking that the item with the given title has been selected. Finally, we print the list of titles so that we can check if the invocation of the `actionPerformed` method has the desired effect.

```
List<String> titles = new LinkedList<String>();
titles.add("New");
titles.add("Open");
titles.add("Save");
titles.add("Print");
titles.add("Exit");
Model model = new Model(titles);
View view = new View(model);
Controller controller = new Controller(view, model);
output.println(model.getTitles());
Random random = new Random();
String title = titles.get(random.nextInt(titles.size()));
output.println("Selected title: " + title);
```

```
controller.actionPerformed(new ActionEvent(new Object(), ActionEvent.
   ACTION_PERFORMED, title));
output.println(model.getTitles());
```

6.3.5 The `GUI` App

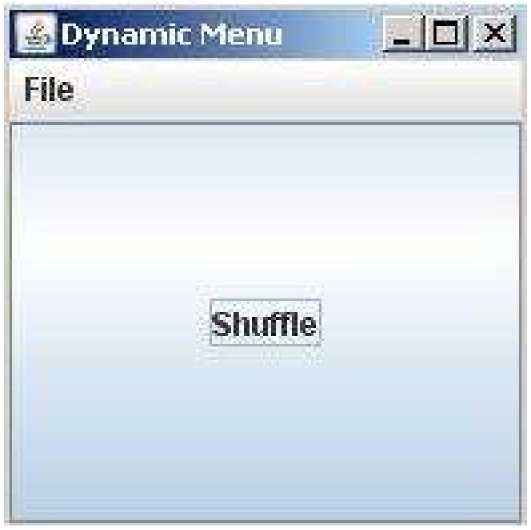
We have already discussed all the ingredients of the GUI app. Here, we simply show the body of its `main` method.

```
List<String> titles = new LinkedList<String>();
titles.add("New");
titles.add("Open");
titles.add("Save");
titles.add("Print");
titles.add("Exit");
Model model = new Model(titles);
View view = new View(model);
final int WIDTH = 200;
final int HEIGHT = 50;
view.setSize(WIDTH, HEIGHT);
view.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
view.setVisible(true);
```

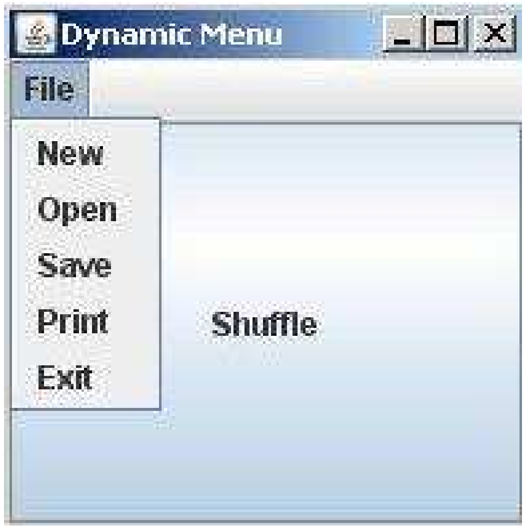
The code of the `GUI` app can be found by following this link.

6.4 Shuffling the Items

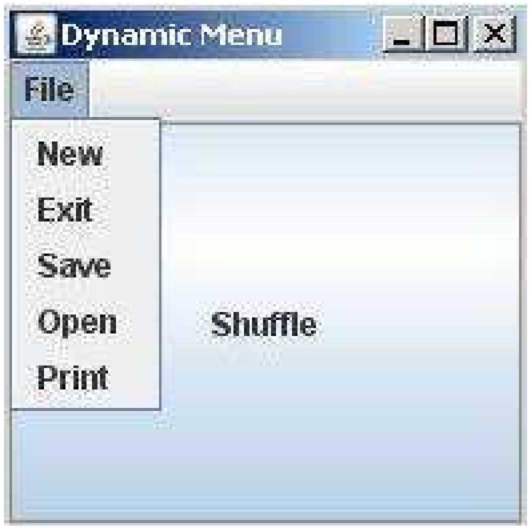
We add a button to our GUI. The button is emtitled “Shuffle.”



When the client presses the button, the items of the menu are shuffled. For example, assume that the menu items are ordered as follows.



After the client has pressed the button, the order of the items may have changed as follows.



Next, we discuss the modifications to the classes `Model`, `View` and `Controller`.

6.4.1 The `Model` Class

Recall that the `Model` has a list of titles. When the client presses the button, the `Controller` should shuffle this list. Hence, we add the method

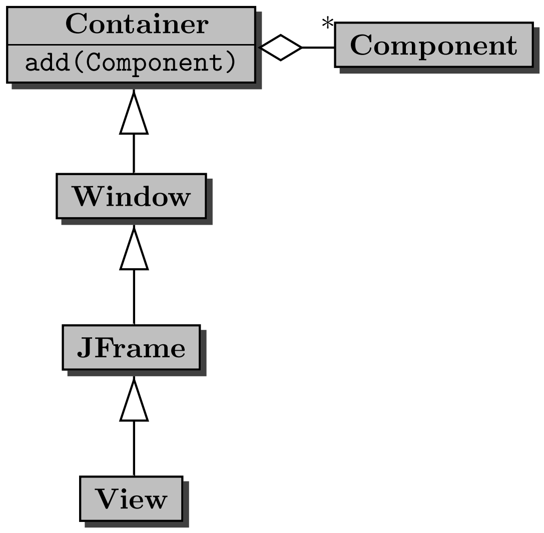
```
public void shuffle()
```

to shuffle the list of titles. To implement this method, we delegate to the utility `Collections`, which is part of the package `java.util`, as follows.

```java
public void shuffle()
{
   Collections.shuffle(this.getTitles());
}
```

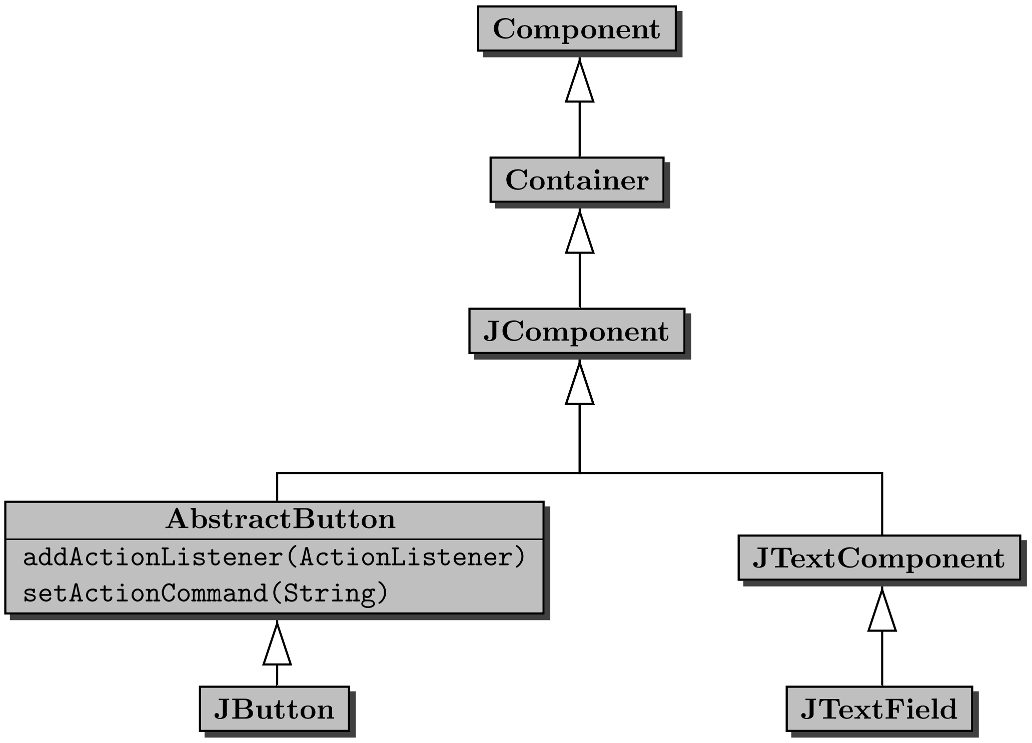
6.4.2 The `View` Class

The modified `View` has a button. This button can be represented by a `JButton` object. We may be tempted to introduce an attribute of type `JButton`. However, consider the following class diagram.



Note that a `Container` has a collection of `Components`. This collection contains the components (such as buttons and text fields) that are part of the container. Note that our `View` is a `Container`. As a consequence, the collection of `Components` is part of the state of the `View`.

Also consider the following class diagram.



Note that a `JButton` is a `Component` and, hence, can be added to the above mentioned collection of `Components`. Hence, rather than adding an attribute of type `JButton` to our `View`, we create a `JButton` object and add it to the collection of `Components`, which is part of the state of our `View`, using the `add` method inherited by our `View` class from the `Container` class.

As we did before for the menu items, we have to

- set the action command of the button, and
- add the `Controller` as an `ActionListener` of the button.

Recall that the action command is shared by the `Component` (the `JButton` in this case), which is part of the `View`, and the `ActionEvent`, which is provided as an argument to the `actionPerformed` method of the `Controller`. Hence, the action command is shared by the `View` and the `Controller`. Rather than arbitrarily placing it in either the `View` class or the `Controller` class, we introduce a new class `ActionCommands` that contains the action command.

```
public class ActionCommands
{
   public static final String SHUFFLE = "Shuffle";
}
```

To the constructor of the `View` class we add the following.

```
JButton button = new JButton("Shuffle");
this.add(button);
button.addActionListener(controller);
button.setActionCommand(ActionCommands.SHUFFLE);
```

6.4.3 The `Controller` Class

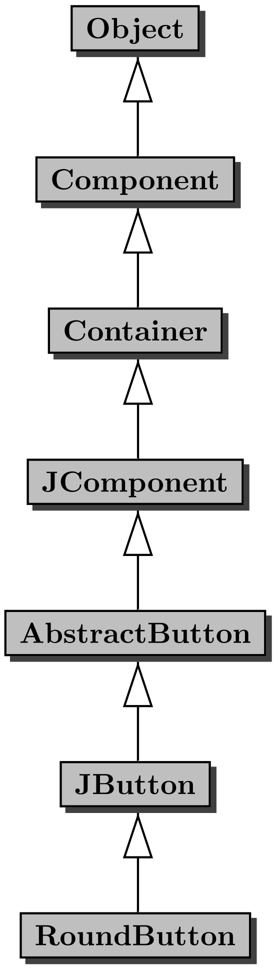
As we already mentioned above, whenever the client presses the button, the `Controller` should invoke the `shuffle` method on the `Model`. Hence, we modify the `actionPerformed` method to the following.

```
public void actionPerformed(ActionEvent event)
{
   String action = event.getActionCommand();
   if (action.equals(ActionCommands.SHUFFLE))
   {
      this.getModel().shuffle();
   }
   else
   {
      this.getModel().select(action);
   }
   this.getView().setMenu(this.getModel().getTitles(), this);
}
```

6.5 Beyond the Basics

6.5.1 A Round Button

We replace the rectangular button to shuffle the menu items with a round button. Rather than developing a `RoundButton` class from scratch, we extend the `JButton` class.



Our `RoundButton` class inherits more than four hundred methods from `JButton` and its superclasses and more than one hundred attributes of `JButton` and its superclasses are part of the state of a `RoundButton`. To express that the `RoundButton` class is a subclass of the `JButton` class, we use the following class header.

```
public class RoundButton extends JButton
```

In our `RoundButton` class, we only need to provide a constructor (since constructors are not inherited) and override the methods `paintBorder` and `contains`. The former method paints the button's border. Since the border of a `RoundButton` is round, whereas the border of a `JButton` is rectangular, the method needs to be overridden. The latter method defines the shape of the button. Because a `RoundButton` is round, whereas a `JButton` is rectangular, also this method needs to be overridden.

The constructor of our `RoundButton` class takes the title of the button as its single argument. Since the `RoundButton` class extends the `JButton` class, we first delegate to a constructor of the superclass to initialize the state. Furthermore, we have to set the `contentAreaFilled` attribute to false so that the area of the `JButton`, which is rectangular, is not filled.

```
public RoundButton(String title)
```

```
{
   super(title);
   this.setContentAreaFilled(false);
}
```

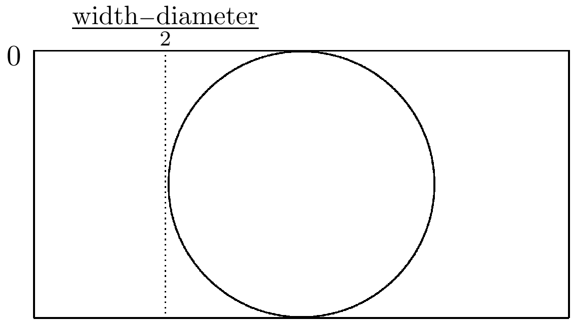
The method `paintBorder` paints the border of the button on the given `Graphics` object. This `Graphics` object is passed to the method by the Java virtual machine. Since our `RoundButton` is a `JButton` and a `JButton` has a width and a height, we take the diameter of the button to be the minimum of that width and height.

To draw a circle, we use the method

```
public void drawOval(int x, int y, int width, int height)
```

where `x` and `y` are the x- and y-coordinate of the left upper corner of the oval to be drawn and `width` and `height` are width and height of the oval to be drawn.

If the width is greater than the height, then the left upper corner has coordinates $(\frac{\text{width}-\text{diameter}}{2}, 0)$.



Otherwise, the left upper corner has coordinates $(0, \frac{\text{height}-\text{diameter}}{2})$. Both cases can be unified into one: the left upper corner has coordinates $(\frac{\text{width}-\text{diameter}}{2}, \frac{\text{height}-\text{diameter}}{2})$. Note that if the width is greater than the height, then the diameter is equal to the height and, hence, $\frac{\text{height}-\text{diameter}}{2}$ equals 0.

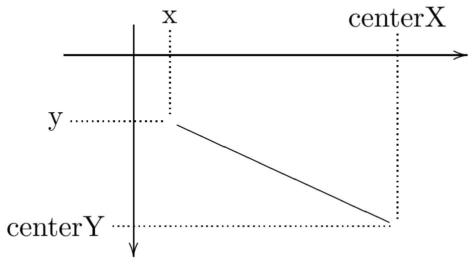
Since we want to draw a circle, we use the diameter of the circle as the width and height of the oval.

Hence, the `paintBorder` method can be implemented as follows.

```
public void paintBorder(Graphics canvas)
{
   int diameter = Math.min(this.getWidth(), this.getHeight());
   canvas.drawOval((this.getWidth() - diameter) / 2, (this.getHeight() -
      diameter) / 2, diameter, diameter);
}
```

The method `contains` checks whether the point specified by the given x- and y-coordinate falls within the circle that outlines the round button. The operating system and the Java virtual machine provide the x- and y-coordinate to the `contains` method.

The point (x, y) is within the circle if and only if the distance from (x, y) to the centre of the circle is smaller than or equals to the radius of the circle (the radius is half the diameter). Let (centerX, centerY) be the coordinates of the centre of the circle.



The distance from (x, y) to (centerX, centerY) is

$$\sqrt{(\mathrm{x} - \mathrm{centerX})^2 + (\mathrm{y} - \mathrm{centerY})^2}$$

Hence, the `contains` method can be implemented as follows.

```
public boolean contains(int x, int y)
{
   int diameter = Math.min(this.getWidth(), this.getHeight());
   double centerX = this.getWidth() / 2.0;
   double centerY = this.getHeight() / 2.0;
   return Math.sqrt(Math.pow(centerX - x, 2.0) + Math.pow(centerY - y, 2.0))
       <= diameter / 2.0;
}
```