Case Study—Part 2

12

Outcomes:

By the end of this chapter you should be able to:

- *design an attractive graphical user interface*;
- *use pseudocode to design event handling routines*;
- *implement the design in Java using a variety of JavaFX components*;
- *devise a testing strategy for a complete application and carry out the necessary steps to implement that strategy*.

12.1 Introduction

In the previous chapter we designed and developed the core classes required to implement the functionality of our *Student Hostel System*. We now go on develop a graphical user interface for this application.

12.2 Keeping Permanent Records

In practice, an application such as the *Student Hostel System* would not be much use if we had no way of keeping permanent records—in other words, of saving a file to disk. However, reading and writing files is something that you will not learn until your second semester. So, in the meantime, in order to make it possible to keep a permanent record of your data, we have created a special class for you to use; we have called this class `TenantFileHandler`. This class (along with the rest of the files from this case study) can be found on the accompanying website.

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_12

The `TenantFileHandler` class has two **`static`** methods: the first, `saveRecords`, needs to be sent two parameters, an integer value indicating the number of rooms in the hostel, and a `TenantList`, which is a reference to the list to be saved; the second, `readRecords`, requires only a reference to a `TenantList` so that it knows where to store the information that is read from the file.

The `readRecords` method will be called when the application starts (so this method call will therefore be coded into the `start` method of the JavaFX application), and the `saveRecords` method will be called when we finish the application (and will therefore be coded into the event-handler of a "Save and Quit" button). The user can of course exit without saving by clicking the cross-hairs, just in case, for any reason, the user should want to abandon any changes.

12.3 Design of the *Hostel* Class

In Fig. 11.1 of the previous chapter we presented the `Hostel` class as part of the Student Hostel application design, but did not give any design details for this class. Let's consider this `Hostel` class now. Figure 12.1 is a reminder of the important classes in the UML diagram and also includes some key attributes and methods.

As you can see, the `Hostel` class requires a single instance of the `TenantList` collection class we developed in Chap. 11. In Fig. 12.1 this instance is recorded as the private `list` attribute in the `Hostel` class.

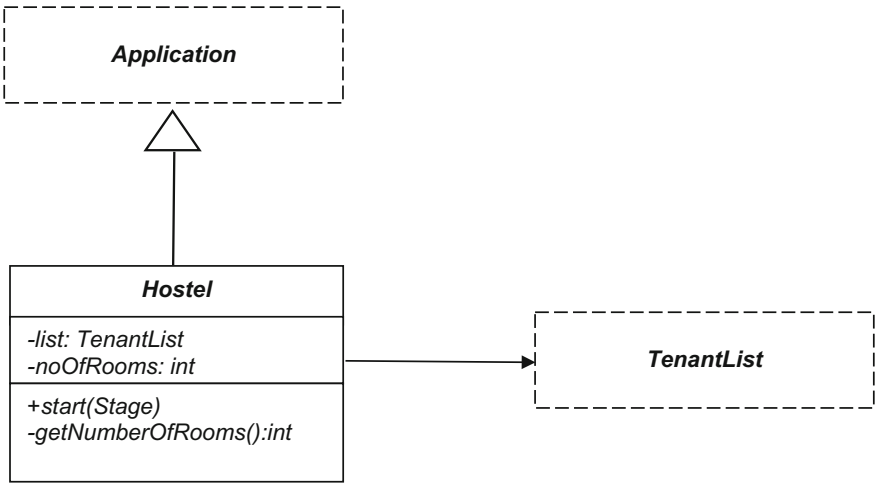

Fig. 12.1 The initial design of the *Hostel* class

Our Student Hostel application will have a JavaFX interface, so the `Hostel` class also needs to inherit from the JavaFX `Application` class. Consequently, we need to provide a `start` method to add components to the `Stage` and process the event-handling routines. This method will also initialise the tenant list and read any data into this list from a file (using the aforementioned `TenantFileHandler` class that we have provided).

In order to initialise the tenant list we have added a `noOfRooms` integer attribute to record how many rooms to limit the hostel to. We have also added a private method, `getNumberOfRooms`, to request this room limit from the user. We have included a `main` method to launch the application (although, as we explained in Chap. 10, this is not always necessary for JavaFX applications).

From this initial class design we have the following outline of our `Hostel` class:

```
import javafx.application.Application;
import javafx.stage.Stage;

public class Hostel extends Application
{
    // attributes
    private TenantList list;
    private int noOfRooms;

    // methods

    @Override
    public void start(Stage stage)
    {
        noOfRooms = getNumberOfRooms(); // call private method
        // initialise tenant list
        list  = new TenantList(noOfRooms);
        TenantFileHandler.readRecords(list);
        // code to layout components, process event handling routines and initialise the list here
    }

    /**
     * Method to request number of hostel rooms from the user
     * @return number of rooms
     */
    private int getNumberOfRooms( )
    {
        System.out.print("How many rooms?: ");
        int num = EasyScanner.nextInt();
        return num;
    }


    public static void main(String[] args)
    {
        launch(args);
    }
}
```

You can see, in the `start` method, we have called the **`private`** `getNumberOfRooms` method to request and return the number of rooms to limit the tenant list to. The `getNumberOfRooms` method uses a standard text window for output and the keyboard for input. The `readRecords` method of the `TenantFileHandler` class is then used in the `start` method to load into the tenant list any pre-existing `Tenant` records saved to file.

The remaining code within this class will relate to the GUI for this application, so let's consider the design of the GUI now.

12.4 Design of the GUI

There will be two aspects to the design of the graphical interface. Firstly, we need to design the visual side of things; then we need to design the algorithms for our event-handling routines so that the buttons do the jobs we want them to, like adding or displaying tenants.

Let's start with the visual design. We need to choose which graphics components we are going to use and how to lay them out. One way to do this is to make a preliminary sketch such as the one shown in Fig. 12.2.

It should be clear which `JavaFX` components we will be using here. We have a selection of buttons on our GUI (Fig. 12.3).

The remaining components are `Labels` ("Hostel Application", "Room" "Name", "Month" and "Amount") and `TextFields` (for the five single-row boxes) or `TextAreas` (for the two multiple-row boxes).

In the examples you saw in Chap. 10, you saw that we created our visual components within our `start` method. This made sense as the algorithms for our button event handlers (which needed access to these components) were also contained within the `start` method. In this application, however, we would expect to have much more complicated algorithms for our button event handlers, so we will structure things a little differently. In this `Hostel` class we will implement our event handlers in a series of **`private`** methods (one for each button).

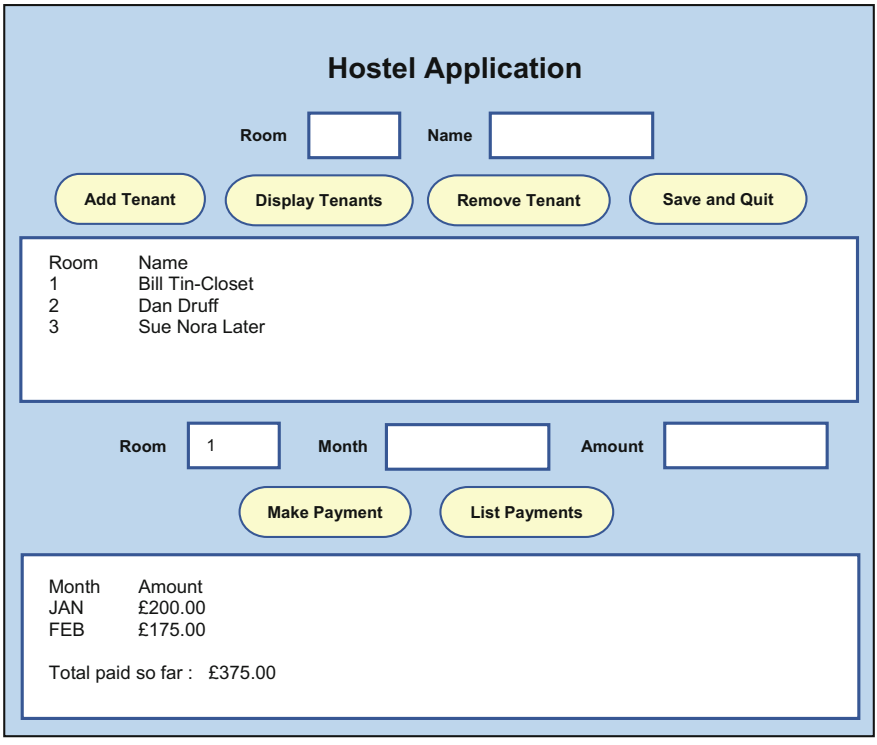

Fig. 12.2 Preliminary design of the *Hostel* GUI

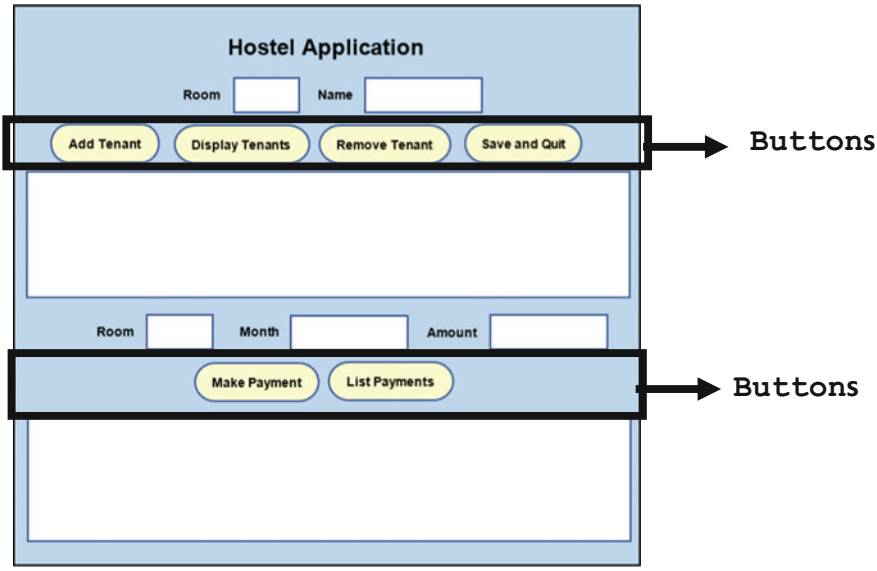

Fig. 12.3 Some graphical components in the *Hostel* GUI

One implication of implementing the event handling code in separate methods is that it makes more sense now to make our visual components accessible *throughout* the class rather than just in the `start` method. To do that we make these visual components *attributes* of the `Hostel` class. In the code snippet below we create these component attributes in the order they appear in our GUI design of Fig. 12.2 (from top to bottom and left to right):

```
public class Hostel extends Application
{
    // previous attributes here

    // visual components declared as attributes of the class
    private Label headingLabel = new Label("Hostel Application");
    private Label roomLabel1 = new Label("Room");
    private TextField roomField1 =  new TextField();
    private Label nameLabel = new Label("Name");
    private TextField nameField =  new TextField();
    private Button addButton = new Button("Add Tenant");
    private Button displayButton =  new  Button("Display Tenants");
    private Button removeButton  = new Button("Remove Tenant");
    private Button saveAndQuitButton  = new Button("Save and Quit");
    private TextArea displayArea1  = new TextArea();
    private Label roomLabel2 = new Label("Room");
    private TextField roomField2  = new TextField();
    private Label monthLabel = new Label("Month");
    private TextField monthField  = new TextField();
    private Label amountLabel = new Label("Amount");
    private TextField amountField =  new TextField();
    private Button paymentButton  = new Button("Make Payment");
    private Button listButton  = new Button("List Payments");
    private TextArea displayArea2 =  new TextArea();

    // code for methods goes here
}
```

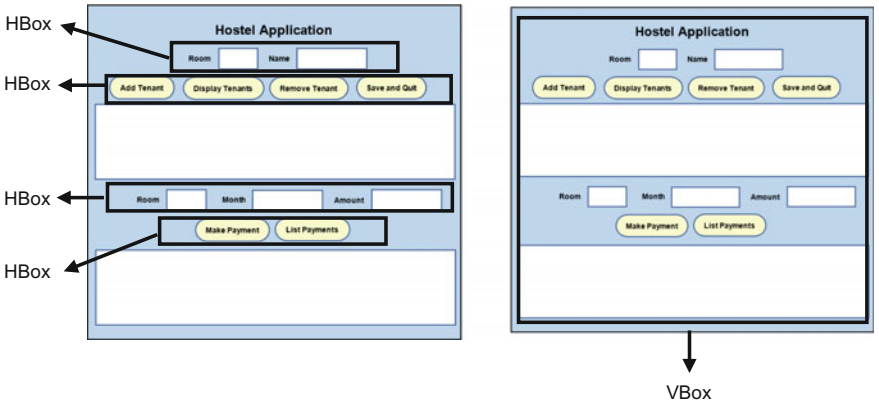

Fig. 12.4 Organizing the *Hostel* GUI with HBoxes and a VBox

Now let's turn to the layout of these components. We will use a mixture of `HBox`es and a `VBox` to organise our layout (see Fig. 12.4).

As always, we use the `start` method to organise our layout:

```
public void start(Stage stage)
{
    // previous code here

    // create four HBoxes
    HBox roomDetails = new HBox (10);
    HBox tenantButtons = new HBox(10);
    HBox paymentDetails = new HBox(10);
    HBox paymentButtons = new HBox(10);
    // add components to HBoxes
    roomDetails.getChildren().addAll(roomLabel1, roomField1, nameLabel, nameField);
    tenantButtons.getChildren().addAll(addButton, displayButton, removeButton, saveAndQuitButton);
    paymentDetails.getChildren().addAll(  roomLabel2, roomField2, monthLabel, monthField,
                                          amountLabel, amountField);
    paymentButtons.getChildren().addAll(paymentButton, listButton);
    // create VBox
    VBox root = new VBox(10);
    // add components to VBox
    root.getChildren().addAll( headingLabel, roomDetails, tenantButtons, displayArea1,
                               paymentDetails, paymentButtons, displayArea2);
    // add the VBox to the Scene
    Scene scene = new Scene(root, Color.LIGHTBLUE);


    // rest of start method here
}
```

We have created the four `HBox`es given in Fig. 12.4 and then added the relevant components, we then do the same for the `VBox`. Finally, we add the `VBox` to the `Scene`.

Eventually we will also customise the look of our visual components (by setting fonts and borders for example) when we present the complete code for this class. But now let's turn our attention to designing the event-handlers for the buttons on our GUI.

12.5 Designing the Event-Handlers

As you saw in Fig. 12.2, there are six buttons that need to be coded so that they respond in the correct way when pressed:

- the "Add Tenant" button;
- the "Display Tenants" button;
- the "Remove Tenant" button;
- the "Save and Quit" button;
- the "Make Payment" button;
- the "List Payments" button.

As always, we will use the `setOnAction` method of each button to process these button clicks, but (as we said in the previous section) we will place the code for the event-handlers in separate **`private`** methods and call these methods from our lambda expressions:

```
public void start(Stage stage)
{
    // previous code here

    // call private methods for button event handlers
    addButton.setOnAction(e -> addHandler());
    displayButton.setOnAction(e -> displayHandler() );
    removeButton.setOnAction( e -> removeHandler());
    paymentButton.setOnAction( e -> paymentHandler());
    listButton.setOnAction( e -> listHandler());
    saveAndQuitButton.setOnAction( e -> saveAndQuitHandler());

    // rest of start method here
}

// private event handler methods here
```

We have summarized below the task that each button's event-handler method must perform, and then gone on to design our algorithms using pseudocode.

The Add Tenant Button

The purpose of this button is to add a new `Tenant` to the list. The values entered in `roomField1` and `nameField` must be validated; first of all, they must not be blank; second, the room number must not be greater than the number of rooms available (or less than 1!); finally, the room must not be occupied. If all this is okay, then the new tenant is added (we will make use of the `addTenant` method of `TenantList` to do this) and a message should be displayed in `displayArea1`. We can express this in pseudocode as follows:

```
read roomField1
read nameField
IF roomField1 blank OR nameField blank
    display blank field error in displayArea1
ELSE IF roomField1 value < 1 OR roomField1 value > noOfRooms
    display invalid room number error in displayArea1
ELSE IF tenant found in room
    display room occupied error in displayArea1
ELSE
BEGIN
    add tenant
    blank roomField
    blank nameField
    display message to confirm success in displayArea1
END
```

The Display Tenants Button

Pressing this button will display the full list of tenants (room number and name) in `displayArea1`.

If all the rooms are vacant a suitable message should be displayed; otherwise the list of tenants' rooms and names should appear under appropriate headings as can be seen in Fig. 12.2. This can be expressed in pseudocode as follows:

```
IF list is empty
    display rooms empty error in displayArea1
ELSE
BEGIN
    display header in displayArea1
    LOOP FROM first item TO last item in list
    BEGIN
        append tenant room and name to displayArea1
    END
END
```

The Remove Tenant Button

Clicking on this button will remove the tenant whose room number has been entered in `roomField1`.

As with the *Add Tenant* button, the room number entered must be validated; if the number is a valid one then the tenant is removed from the list (we will make use of the `remove` method of `TenantList` to do this) and a confirmation message is displayed. The pseudocode for this event-handler is given as follows:

```
read roomField1
IF roomField1 blank
    display blank field error in displayArea1
ELSE IF roomField1 value < 1 OR roomField1 value > noOfRooms
    display invalid room number error in displayArea1
ELSE IF no tenant found in room
    display room empty error in displayArea1
ELSE
BEGIN
    remove tenant from list
    display message to confirm success in displayArea1
END
```

The List Payment Button

This button records payments made by an individual tenant whose room number is entered in `roomField2`. The values entered in `roomField2`, `monthField` and `amountField` must be validated to ensure that none of the fields are blank, that the room number is a valid one and, if so, that it is currently occupied.

If everything is okay then a new payment record is added to that tenant's list of payments (we will make use of the `makePayment` method of `PaymentList` to do this) and a confirmation message is displayed in `displayArea2`. This design is expressed in pseudocode as follows:

```
read roomField2
read monthField
read amountField
IF roomField2 blank OR monthField blank OR amountField blank
    display fields empty error in displayArea2
ELSE IF roomField2 value < 1 OR roomField2 value > noOfRooms
    display invalid room number error in displayArea2
ELSE IF no tenant found in room
    display room empty error in displayArea2
ELSE
BEGIN
    create payment from amountField value and monthField value
    add payment into list
    display message to confirm success in displayArea2
END
```

The List Payments Button

Pressing this button causes a list of payments (month and amount) made by the tenant whose room number is entered in `roomField2` to be displayed in `displayArea2`.

After validating the values entered, each record in the tenant's payment list is displayed. Finally, the total amount paid by that tenant is displayed (we will make use of the `calculateTotalPaid` method of `PaymentList` to do this). The pseudocode is given as follows:

```
read roomField2
IF roomField2 blank
display room field empty error in displayArea2
ELSE IF roomField2 value < 1 OR roomField2 value > noOfRooms
    display invalid room number error in displayArea2
ELSE IF no tenant found in room
    display room empty error in displayArea2
ELSE
BEGIN
    find tenant in given room
    get payments of tenant
    IF payments = 0
        display no payments error in displayArea2
    ELSE
    BEGIN
        display header in displayArea2
        LOOP FROM first payment TO last payment
        BEGIN
            append amount and month to displayArea2
        END
        display total paid in displayArea2
        blank monthField
        blank amountField
    END
END
```

The Save and Quit Button

Pressing this button causes all the records to be saved to a file (here we make use of the `saveRecords` method of the `TenantFileHandler` class that we talked about in Sect. 12.2); it then closes the application, terminating the program.

It will only contain a few lines of code and we have therefore not written pseudocode for it.

12.6 Implementing the *Hostel* Class

The complete code for the `Hostel` class now appears below. When you see the code, you should notice that we have utilized the `NumberFormat` class (which is to be found in the `java.text` package) to print the amounts in the local currency. Also note the use of two constants, `WIDTH` and `HEIGHT`, to help size our visual components and the `parseInt` method of the `Integer` class to convert the room values, entered as text, into integer values. We have also enhanced our visual components by making use of borders and backgrounds as discussed in Chap. 10.

Study the code and the comments carefully (in particular compare the event-handling code to the pseudocode we presented in the previous section) to make sure you understand it and we will explain the new concepts to you after that.

Hostel

```
import java.text.NumberFormat;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.Border;
import javafx.scene.layout.BorderStroke;
import javafx.scene.layout.BorderStrokeStyle;
import javafx.scene.layout.BorderWidths;
import javafx.scene.layout.CornerRadii;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.stage.Stage;
import javafx.scene.control.TextInputDialog;

/**GUI for the Hostel application
  *@author Charatan and Kans
  *@version 7th April 2018
  */
public class Hostel extends Application
{
    // the attributes

    private int noOfRooms;
    private TenantList list;
    // WIDTH and HEIGHT of GUI stored as constants
    private final int WIDTH = 800;
    private final int HEIGHT = 500;
    // visual components
    private Label headingLabel = new Label("Hostel Application");
    private Label roomLabel1 = new Label("Room");
    private TextField roomField1 =  new TextField();
    private Label nameLabel = new Label("Name");
    private TextField nameField =  new TextField();
    private Button addButton = new Button("Add Tenant");
```

```
private Button displayButton =  new  Button("Display Tenants");
private Button removeButton  = new Button("Remove Tenant");
private Button saveAndQuitButton  = new Button("Save and Quit");
private TextArea displayArea1  = new TextArea();
private Label roomLabel2 = new Label("Room");
private TextField roomField2  = new TextField();
private Label monthLabel = new Label("Month");
private TextField monthField  = new TextField();
private Label amountLabel = new Label("Amount");
private TextField amountField =  new TextField();
private Button paymentButton  = new Button("Make Payment");
private Button listButton  = new Button("List Payments");
private TextArea displayArea2 =  new TextArea();



@Override
/** Initialises the screen
 *  @param stage:   The scene's stage
 */
public void start(Stage stage)
{
   noOfRooms = getNumberOfRooms(); // call private method
   // initialise tenant list
   list  = new TenantList(noOfRooms);
   TenantFileHandler.readRecords(list);

   // create four HBoxes
   HBox roomDetails = new HBox (10);
   HBox tenantButtons = new HBox(10);
   HBox paymentDetails = new HBox(10);
   HBox paymentButtons = new HBox(10);
   // add components to HBoxes
   roomDetails.getChildren().addAll(roomLabel1, roomField1, nameLabel, nameField);
   tenantButtons.getChildren().addAll(   addButton, displayButton, removeButton,
                                         saveAndQuitButton);

   paymentDetails.getChildren().addAll(  roomLabel2, roomField2, monthLabel, monthField,
                                         amountLabel, amountField);
   paymentButtons.getChildren().addAll(paymentButton, listButton);
   // create VBox
   VBox root = new VBox(10);
   // add all components to VBox
   root.getChildren().addAll( headingLabel, roomDetails, tenantButtons, displayArea1,
                              paymentDetails, paymentButtons, displayArea2);
   // create the scene
   Scene scene = new Scene(root, Color.LIGHTBLUE);

   // set font of heading
   Font font = new Font("Calibri", 40);
   headingLabel.setFont(font);

   // set alignment of HBoxes
   roomDetails.setAlignment(Pos.CENTER);
   tenantButtons.setAlignment(Pos.CENTER);
   paymentDetails.setAlignment(Pos.CENTER);
   paymentButtons.setAlignment(Pos.CENTER);
   // set alignment of VBox
   root.setAlignment(Pos.CENTER);

   // set minimum and maximum width of components
   roomField1.setMaxWidth(50);
   roomField2.setMaxWidth(50);

   roomDetails.setMinWidth(WIDTH);
   roomDetails.setMaxWidth(WIDTH);

   tenantButtons.setMinWidth(WIDTH);
   tenantButtons.setMaxWidth(WIDTH);

   paymentDetails.setMinWidth(WIDTH);
   paymentDetails.setMaxWidth(WIDTH);

   paymentButtons.setMinWidth(WIDTH);
   paymentButtons.setMaxWidth(WIDTH);

   root.setMinSize(WIDTH, HEIGHT);
   root.setMaxSize(WIDTH, HEIGHT);

   displayArea1.setMaxSize(WIDTH - 80, HEIGHT/5);
   displayArea2.setMaxSize(WIDTH - 80, HEIGHT/5);

   stage.setWidth(WIDTH);
   stage.setHeight(HEIGHT);
```

```
    // customise the visual components

    // customise the VBox border and background
    BorderStroke style = new BorderStroke(Color.BLACK, BorderStrokeStyle.SOLID,
                                          new CornerRadii(0), new BorderWidths(2) );
    root.setBorder(new Border (style));
    root.setBackground(Background.EMPTY);

    // customise buttons
    addButton.setBackground(new Background(new BackgroundFill(Color.LIGHTYELLOW,
                            new CornerRadii(10), Insets.EMPTY)));
    displayButton.setBackground(   new Background(new BackgroundFill(Color.LIGHTYELLOW,
                                   new CornerRadii(10), Insets.EMPTY)));
    removeButton.setBackground(new Background(new BackgroundFill(Color.LIGHTYELLOW,
                               new CornerRadii(10), Insets.EMPTY)));
    saveAndQuitButton.setBackground(  new Background(new BackgroundFill(Color.LIGHTYELLOW,
                                      new CornerRadii(10), Insets.EMPTY)));
    paymentButton.setBackground(   new Background(new BackgroundFill(Color.LIGHTYELLOW,
                                   new CornerRadii(10), Insets.EMPTY)));
    listButton.setBackground(  new Background(new BackgroundFill(Color.LIGHTYELLOW,
                               new CornerRadii(10), Insets.EMPTY)));

    // call private methods for button event handlers
    addButton.setOnAction(e -> addHandler());
    displayButton.setOnAction(e -> displayHandler() );
    removeButton.setOnAction( e -> removeHandler());
    paymentButton.setOnAction( e -> paymentHandler());
    listButton.setOnAction( e -> listHandler());
    saveAndQuitButton.setOnAction( e -> saveAndQuitHandler());

    // configure the stage and make the stage visible
    stage.setScene(scene);
    stage.setTitle("Hostel Applicaton");
    stage.setResizable(false); // see discussion below
    stage.show();

}

/**
 * Method to request number of hostel rooms from the user
 * @return number of rooms
 */
private int getNumberOfRooms()
{
    TextInputDialog dialog = new TextInputDialog();
    dialog.setHeaderText("How many rooms?");
    dialog.setTitle("Room Information Request");

    String response = dialog.showAndWait().get();
    return Integer.parseInt(response);
}

// event handler methods

private void addHandler()
{
    String roomEntered =  roomField1.getText();
    String nameEntered =  nameField.getText();
    // check for errors
    if(roomEntered.length()== 0 || nameEntered.length()== 0)
    {
        displayArea1.setText ("Room number and name must be entered");
    }
    else if(Integer.parseInt(roomEntered)< 1 || Integer.parseInt(roomEntered)>noOfRooms)
    {
        displayArea1.setText ("There are only "  + noOfRooms  + " rooms");
    }
    else if(list.search(Integer.parseInt(roomEntered)) !=  null)
    {
        displayArea1.setText("Room number " +  Integer.parseInt(roomEntered)  + " is occupied");
    }
    else  // ok to add a Tenant
    {
        Tenant t =  new Tenant(nameEntered,Integer.parseInt(roomEntered));
        list.addTenant(t);
        roomField1.setText("");
        nameField.setText("");
        displayArea1.setText("New tenant in room "    +  roomEntered +  " successfully added");
    }
}
```

```
public void displayHandler()
{
    int i;
    if(list.isEmpty()) // no rooms to display
    {
        displayArea1.setText("All rooms are empty");
    }
    else // display rooms
    {
        displayArea1.setText("Room" +  "\t" +  "Name" +  "\n");
        for(i = 1; i <=  list.getTotal(); i++ )
        {
            displayArea1.appendText(list.getTenant(i).getRoom()
                                    + "\t\t"
                                    + list.getTenant(i).getName() + "\n");
        }
    }
}

private void removeHandler()
{
    String roomEntered =  roomField1.getText();
    // check for errors
    if(roomEntered.length()== 0)
    {
        displayArea1.setText("Room number must be entered");
    }
    else if(Integer.parseInt(roomEntered) < 1 || Integer.parseInt(roomEntered)>noOfRooms)
    {
        displayArea1.setText("Invalid room number");
    }
    else if(list.search(Integer.parseInt(roomEntered))== null)
    {
        displayArea1.setText("Room number " +  roomEntered +  " is empty");
    }
    else // ok to remove Tenant
    {
        list.removeTenant(Integer.parseInt(roomEntered));
        displayArea1.setText("Tenant removed from room " +  Integer.parseInt(roomEntered));
    }
}

private void paymentHandler()
{
    String roomEntered = roomField2.getText();
    String monthEntered = monthField.getText();
    String amountEntered = amountField.getText();
    // check for errors
    if(roomEntered.length()== 0 || monthEntered.length()== 0 || amountEntered.length()== 0)
    {
        displayArea2.setText("Room number, month and amount must all be entered");
    }
    else if(Integer.parseInt(roomEntered) < 1 || Integer.parseInt(roomEntered)>noOfRooms)
    {
        displayArea2.setText("Invalid room number");
    }
    else if(list.search(Integer.parseInt(roomEntered)) == null)
    {
        displayArea2.setText("Room number " +  roomEntered +  " is empty");
    }
    else // ok to process payment
    {
        Payment p =  new Payment(monthEntered,Double.parseDouble(amountEntered));
        list.search(Integer.parseInt(roomEntered)).makePayment(p);
        displayArea2.setText("Payment recorded");
    }
}

private void listHandler()
{
    int i;
    String roomEntered =  roomField2.getText();
    // check for errors
    if(roomEntered.length()== 0)
    {
        displayArea2.setText("Room number must be entered");
    }
    else if(Integer.parseInt(roomEntered) < 1 || Integer.parseInt(roomEntered) > noOfRooms)
    {
        displayArea2.setText("Invalid room number");
    }
    else if(list.search(Integer.parseInt(roomEntered)) == null)
    {
        displayArea2.setText("Room number " + Integer.parseInt(roomEntered) + " is empty");
    }
```

```
        else // ok to list payments
        {
            Tenant t =  list.search(Integer.parseInt(roomEntered));
            PaymentList p  = t.getPayments();
            if(t.getPayments().getTotal() == 0)
            {
                displayArea2.setText("No payments made for this tenant");
            }
            else
            {
                /* The NumberFormat class is similar to the DecimalFormat class that we used
                   previously.
                   The getCurrencyInstance method of this class reads the system values to find out
                which country we are in, then uses the correct currency symbol */
                NumberFormat nf =  NumberFormat.getCurrencyInstance();
                String s;
                displayArea2.setText("Month" +  "\t\t" +  "Amount" +  "\n");
                for(i =  1; i <=  p.getTotal(); i++  )
                {
                    s =  nf.format(p.getPayment(i).getAmount());
                    displayArea2.appendText("" + p.getPayment(i).getMonth() +  "\t\t\t" + s + "\n");
                }
                displayArea2.appendText("\n" + "Total paid so far :   " +
                                                nf.format(p.calculateTotalPaid()));
                monthField.setText("");
                amountField.setText("");
            }
        }
    }

    private void saveAndQuitHandler()
    {
       TenantFileHandler.saveRecords(noOfRooms,list);
       Platform.exit();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Before we complete our examination of the `Hostel` application we just draw your attention to a few new features.

Firstly, we ask the user for the number of rooms in the hostel by calling the helper method `getNumberOfRooms`:

```
    private int getNumberOfRooms()
    {
        TextInputDialog dialog = new TextInputDialog();
        dialog.setHeaderText("How many rooms?");
        dialog.setTitle("Room Information Request");

        String response = dialog.showAndWait().get();
        return Integer.parseInt(response);
    }
```

This makes use of a class that we will not actually come across until Chap. 17—the `TextInputDialog` class. This class provides a really useful way to get information from the user during the running of a JavaFX application. However, as it involves a few advanced concepts we won't deal with it in detail until semester two. For now, if you want to use it, you can just copy what we have done here. The result is shown in Fig. 12.5.

Next, as mentioned above, one new feature we made use of is the `NumberFormat` class. This class is similar to `DecimalFormat` that you met in Chap. 10, except that it is designed specifically to convert decimal numbers and convert them into local currency formats. The `getCurrencyInstance` picks up

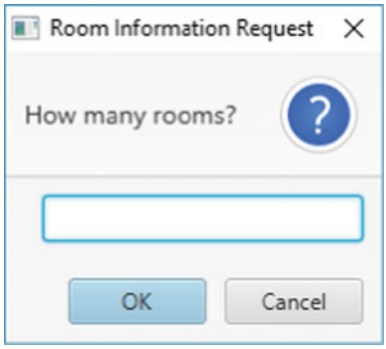

Fig. 12.5 Getting the number of rooms by using a text input dialog

the correct location by interrogating the system and it then returns an appropriate format object:

```
// generate a NumberFormat object
NumberFormat nf =  NumberFormat.getCurrencyInstance();
```

This object's `format` method can then be used to take decimal numbers and format them as local currency values. So the following expression:

```
nf.format(p.calculateTotalPaid())
```

would take a decimal number and, as we are in the United Kingdom, would format the number to two decimal places with a pound sterling symbol (£).

Finally, notice how we configure the stage before making it visible:

```
stage.setScene(scene);
stage.setTitle("Hostel Applicaton");
stage.setResizable(false); // stop the user resizing the stage window
stage.show();
```

You can see we have used a new `stage` method here called `setResizable`. This method takes a **`boolean`** parameter and giving it a value of **`false`** ensures the user cannot resize the window. We thought that would be a good idea as we have gone to such lengths in the code to size our window and the components within it (using our `WIDTH` and `HEIGHT` constants)! One implication of a non-resizable window is that the maximise icon of the window will be greyed out. The final running JavaFX GUI can be seen now in Fig. 12.6.

Before concluding this case study we shall consider how to test the application to ensure that it conforms to the original specification.

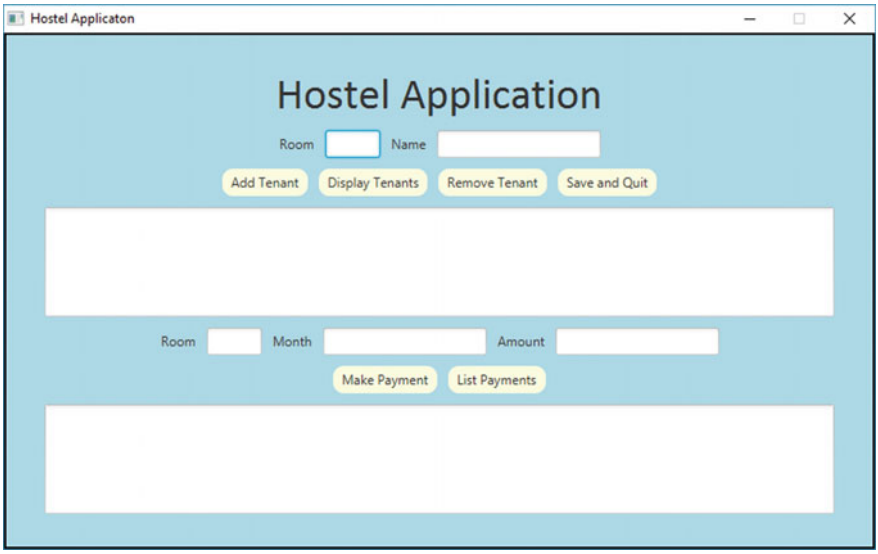

Fig. 12.6 The *Hostel* GUI running in a non-resizable window

12.7 Testing the System

If you look back at the `Hostel` class you can see that much of the event-handling code is related to the validation of data entered from the graphical interface. Much of the testing for such a system will, therefore, be geared around ensuring such validation is effective.

Amongst the types of validation we need to test is the display of suitable error messages when input text fields are left blank, or when inappropriate data has been entered into these text fields. Of course, as well as input validation, we also need to test the basic functionality of the system. Figure 12.7 is one possible test log that may be developed for the purpose of testing the `Hostel` class.

We include a few sample screen shots produced from running the Student Hostel Application against this test log in Figs. 12.8, 12.9 and 12.10. We will leave the complete task of running *Student Hostel Application* against the test log as a programming exercise at the end of this chapter.

TEST LOG			
Purpose: To test the Hostel class			
Run Number:	Date:		
Action	Expected Output	Pass/ Fail	Reason for failure
Request for how many rooms			
Enter 5	Fire up the JavaFX GUI		
Display tenants	"Empty list" message		
Add tenant: Patel, Room Number blank	"Blank field" message		
Add tenant: blank, Room Number 1	"Blank field" message		
Add tenant: Patel, Room Number 1	Confirmation message		
Add tenant: Jones, Room Number 6	Error message: There are only 5 rooms		
Add tenant: Jones, Room Number 1	Error Message: Room 1 is occupied		
Add tenant: Jones, Room Number 2	Confirmation Message		
Display tenants	ROOM NAME		
	1 Patel		
	2 Jones		
List payments, Room Number 1	"Empty list" message		
Make payment: Room blank, Month			
January, Amount 100	"Blank field" message		
Make Payment: Room 1, Month			
blank, Amount 100	"Blank field" message		
Make payment: Room 1, Month			
January, Amount blank	"Blank field" message		
Make payment: Room 1, Month			
January, Amount 100	Confirmation message		
Make payment: Room 1, Month			
February, Amount 200	Confirmation message		
List payments: Room Number blank	"Blank field" message		
List payments, Room Number 1	MONTH AMOUNT		
	January £100		
	February £200		
	Total paid so far £300		
List payments: Room Number 2	"Empty list" message		
List payments: Room Number 5	"Room Empty" message		
Remove tenant: Room Number blank	"Blank field" message		
Remove tenant: Room Number 1	Confirmation Message		
Display tenants	2 Jones		
List payments: Room Number 1	"Room Empty" message		

Fig. 12.7 A test log to ensure the reliability of the *Hostel* class

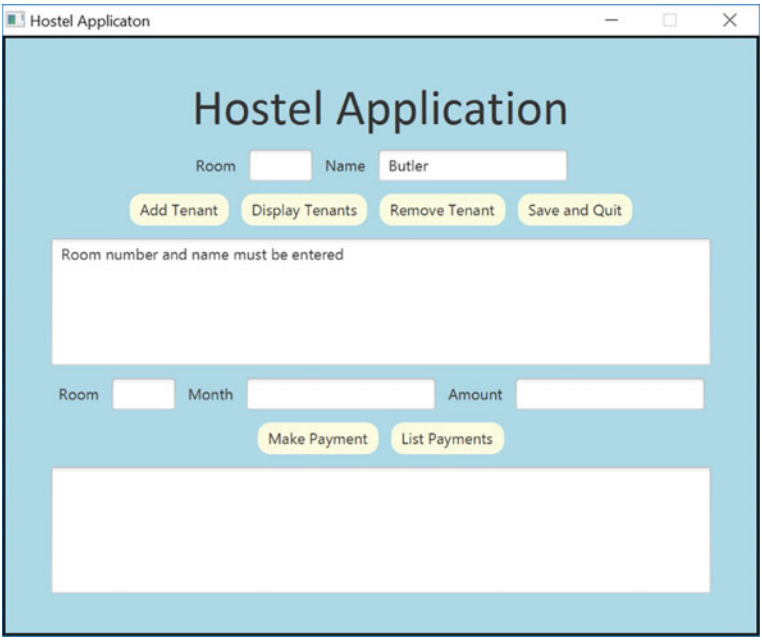

Fig. 12.8 Error messages are produced in *displayArea1*. In this case an attempt is made to add a tenant without filling in the *roomField*

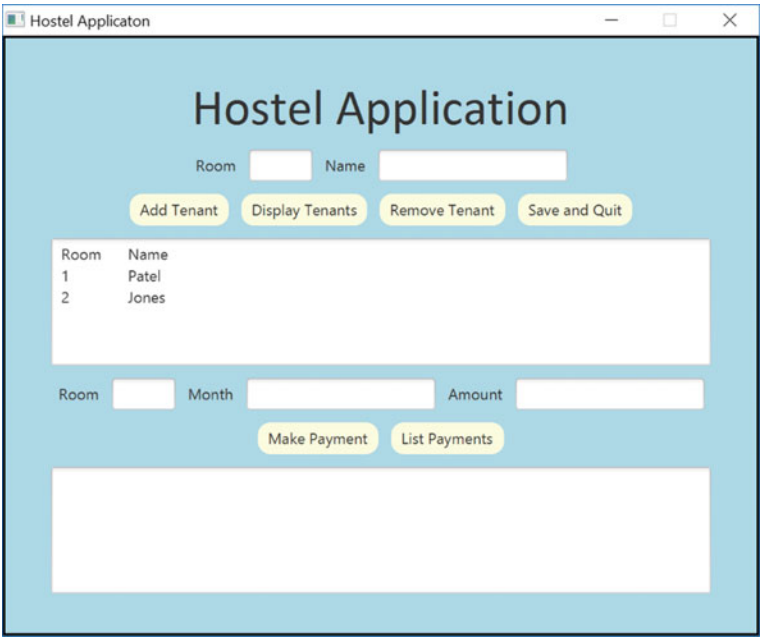

Fig. 12.9 The *displayArea1* is also used to display a list of tenants entered

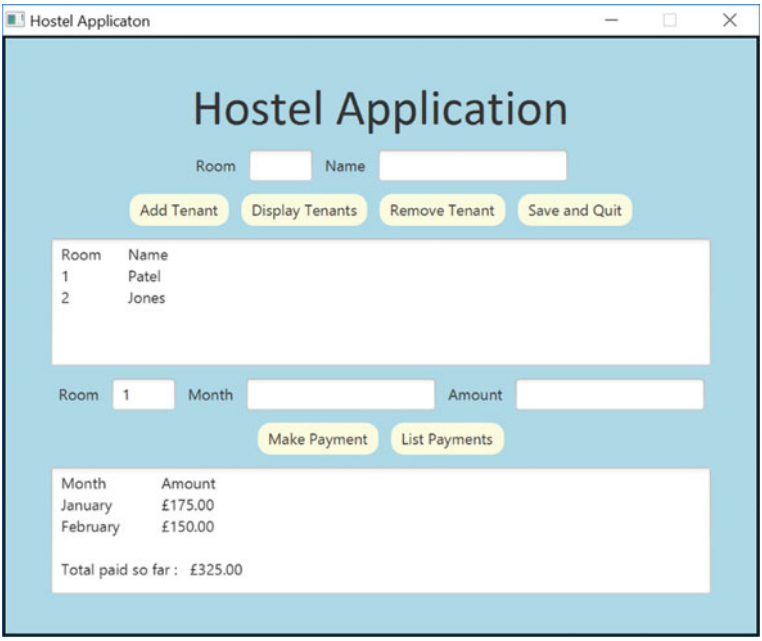

Fig. 12.10 Details of the payments for room 1 are displayed in *displayArea2* when the *List Payments button* is pressed

12.8 What Next?

Congratulations—you have now completed your first semester of programming; we hope you have enjoyed it. Many of you will be going on to at least one more semester of software development and programming—so what lies ahead?

Well, you have probably realized that there are still a few gaps in your knowledge and that some of the stuff that you have learnt can be developed further to give you the power to write multi-functional programs. Think, for example, about the case study we developed in this chapter; you will need to learn how to write the code that stores the information permanently on a disk; also, the JavaFX user interface could be made to look a bit more attractive; and it would be helpful if we had a collection class other than an `ArrayList` that allows us to access an object via a simple key (such as a room number) rather than having to search every item in the collection.

And there is lots more; the standard Java packages provide classes for many different purposes; there is more to learn about interfaces; about dealing with errors and exceptions; about network programming; about accessing information stored in a database and about how to write programs that can perform a number of tasks at

the same time. As well finding out more about the standard Java packages, there is more to learn about how to deploy your own programs as packages.

Does all this sound exciting? We think so—and we hope that you enjoy your next semester as much as we have enjoyed helping you through this one.

12.9 Self-test Questions

1. Referring to the examples in this chapter and the previous chapter, explain the difference between *unit testing* and *integration testing*.
2. Use pseudocode to design the event handling routine for a `search` button. Clicking on the button should display the name of the tenant in the room entered in the `roomField` text box. The name is to be displayed in `displayArea1`. If no tenant is present in the given room, an error message should be displayed in `displayArea1`.
3. Modify the screen design in Fig. 12.1 to include the `search` button discussed in the question above.
4. How else might you improve the application developed in this case study?

12.10 Programming Exercises

You will need to copy the entire suite of classes that make up the student hostel system from the accompanying website.

1. Run the `Hostel` application against the test log given in Fig. 12.7.
2. Modify the `Hostel` class by adding the `search` button you considered in self-test questions 2 and 3 above.
3. Make any additions to the `Hostel` class that you considered in self-test question 4 above. For example you might want to include additional validation to ensure that negative money values are never accepted for payments.