**Appendices K**

As explained within the report, I decided to adopt the Java Swing and AWT frameworks to implement a graphical user interface, and to do this I used a number of Swing and AWT functionality, such as containers, components, layout managers and event listeners.

I am now going to outline the steps I took to implement the Swing and AWT functionality to develop the graphical user interface for the system for MSc Properties.

1. **Components**

As explained within the report, I implemented components to enable the user to be able to input data into the system and allow the system to display system state back to the user.

* 1. **JTable**

For this development, as it consists of managing data and documents, it requires managing lists of information, so I decided to implement a JTable to enable lists of information to be displayed back to the user through the JTable.

For me to implement a JTable I firstly had to create a class that extends AbstractTableModel, to enable me to define an implementation for the main methods for a TableModel, which can then be passed as a parameter for the initialisation of a JTable. As shown in Fig. 1, I have developed an ApplicationTableModel class for all application objects to be displayed in a JTable.

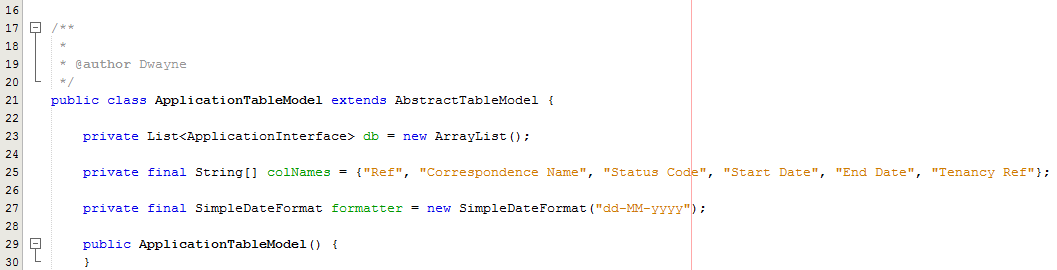


Fig. 1 – Extract from Application Table Model

As shown in Fig. 1, within the ApplicationTableModel class I define an ArrayList of Application objects, which will act as the database to the JTable, secondly, I define a String array, which will list the column names, and finally, as there will be dates displayed back to the user through the JTable, I define a SimeDateFormat object which will be used to format a Date object as a String to control the way the Date is displayed to the user.

As you can see from Fig. 2, I then provide an implementation to override the getColumnName(int column), getRowCount(), getColumnCount() and getValueAt(int rowIndex, int columnIndex) methods, which will reflect the number of columns defined in the class, and the database size defined through the ArrayList.



Fig. 2 – Extract from ApplicationTableModel (overriding methods)

Finally, I define a setData(List<ApplicationInterface> db) method, which will allow clients of the ApplicationTableModel to set the data of the TableModel, which will be used by the JTable to populate the table.

Once the ApplicationTableModel has been created, I then developed an ApplicationPanel class, which extends JPanel (a type of container), and as shown in Fig. 3 defines a JTable object, an ApplicationTableModel object, a JPopupMenu item (acts similar to a JMenubar item, except it pops up from a user action as a pose to being on the menu bar) and a StringListener object (a custom ActionListener).



Fig. 3 – Extract from ApplicationPanel class

As you can see from Fig. 3, the ApplicationTableModel is then initiated along with the JPopupMenu and the JTable, where I pass the ApplicationTableModel object to the JTable as a constructor parameter. I then create a Border for the JPanel in which the JTable is contained within, providing a border round the table. Lastly I create a number of JMenuItems for actions users are able to perform on an Application object that consists within the JTable, and add these JMenuItems to the JPopupMenu (I will define the implementation of the ActionListeners in a later chapter of this appendices).

As you can see from Fi. 4, I then go on to set the column size for the JTable by invoking setPrefferedWidth(int size), on each column from the ColumnModel assigned to the Table (ApplicationTableModel), I then set the title and layout of the ApplicationPanel, and then define a setData(List<ApplicationInterface> db) and refresh() methods to interact with the TableModel (to enable the client of ApplicationPanel to update the JTable through the TableModel), and lastly implement a setTableListener(StringListener listener) and getSelectedObjectRef(), which will allow the ApplicationPanel to notify the client of a user action, and allow the client to extract the selected item within the table respectively.

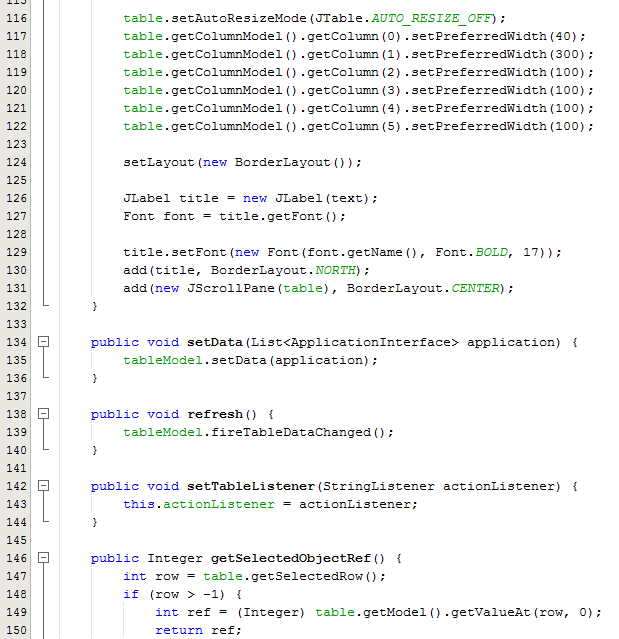


Fig. 4 – Extract from ApplicationPanel

To then use the JTable defined in ApplicationPanel class within a graphical user interface (GUI), I then just need to define and initialise an ApplicationPanel object within the client GUI and supply an ActionListener which will wait for the ApplicationPanel class to invoke a method on the listener object defined in the ApplicationPanel class and initialised through the client invoking setTableListener on the ApplicationPanel (this will be explained in more detail later in the Appendices, but shown in Fig. 5 below), and then add the ApplicationPanel to a container which will then display the JTable to the user.

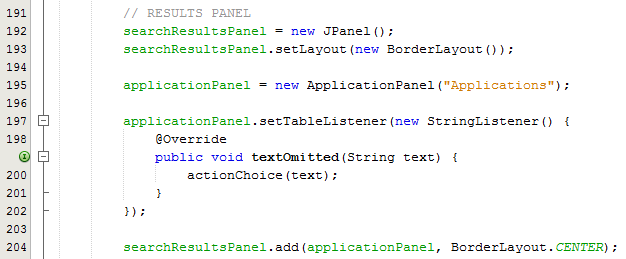


Fig. 5 – Extract from AppSearch class which is a client of the ApplicationPanel class.

N.B. The client is able to interact with the ApplicationPanel by invoking ApplicationPanel methods on the ApplicationPanel object such as setData(List<ApplicationInterface> db) and getSelectedObjectRef().

As this system consists of lists of a wide range of objects, I had to define an ApplicationPanel class and ApplicationTableModel class for each of the different system objects such as Address, AddressUsage, Notes, Tenancies etc.

By implementing a JTable in this was I am adhering to the Model View Controller (MVC) concept in which the model (ClientImpl class) is unaware of the view (ApplicationPanel - JTable) and the two entities communicate through the communicator class (AppSearch class), ensuring that the implementation of a JTable has low coupling with the model.

* 1. **JTree**

For this development, it requires a system configuration screen to allow users to manage the system elements for the system. So I decided to implement a JTree to display the different system elements as a vertical set of hierarchical information.

This would allow the different types of system elements to be used as branches within a JTree, and any actions that can be performed on these different elements, can then be displayed as children of these branches within the JTree.

For me to implement a JTree for the System Configuration screen, I firstly had to create a class, which I called SystemConfigHome that extends JPanel and define and initialise a JTree and pass as a constructor parameter to the JTree a DefaultMutableTreeNode within the SystemConfigHome screen, as shown in Fig. 6.

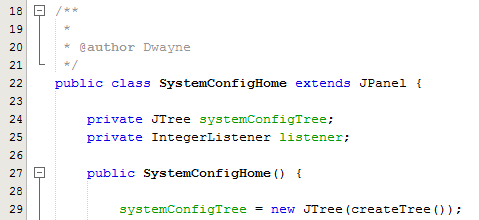


Fig. 6 – Extract from SystemConfigHome

As you can see from Fig. 6 and Fig. 7 I defined a method called createTree() within the SystemConfigHome class which returns a DefaultMutableTreeNode object.

As you can see from Fig. 7, I define and initialise a DefaultMuatbleTreeNode called top, which acts as the root node to the JTree defined in Fig. 6, I then create a number of DefaultMutableTreeNode objects to act as branches where similar to the root node, I pass a String object as a parameter to initialise the DefaultMutableTreeNode, which acts as a label for the branch, and then create a number of DefaultMutableTreeNode objects which act as children branches, but instead of just passing a String object I pass a JTreeInfo object (class which defines a String description and int key, as shown in Fig. 8), with a String as a label, and int as a key.

Depending on the set-up I defined in the design stage. Once I have created all of the branches and leafs (children branches), I then add the leafs (child branches) to the branches by invoking the add() method on the branch, and passing the leaf as a parameter as shown in Fig. 6.

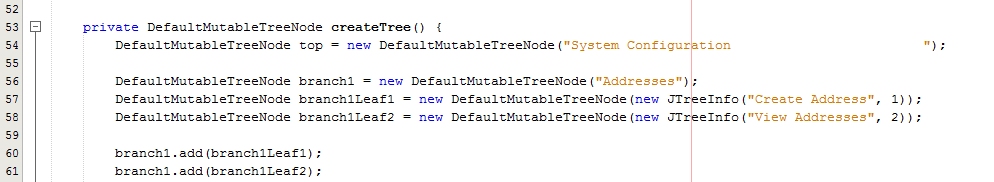


Fig. 7 – Extract from SystemConfigHome – createTree() method

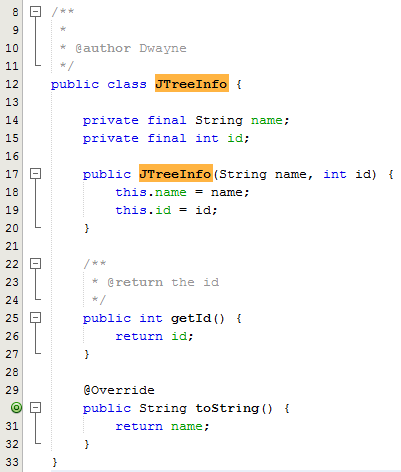


Fig. 8 – JTreeInfo class

I then add all of the branch nodes to the top level (root) node called top, again by invoking the add() method on the top node, and passing the branch nodes as a parameter, and then return the top level (root) node called top, as shown in Fig. 9.

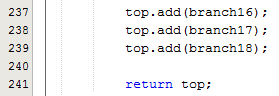


Fig. 9 – Extract from SystemConfigHome – createTree() method

As you can see from Fig. 10 (below), I have defined two methods called getSelectedTreeBranch() and setListener(), which allow a client of SystemConfigHome and the SystemConfigHome object to communicate and allow a client to extract the selected tree branch and the SystemConfigHome object to invoke a method on the listener object defined to notify the client of an event from the JTree.

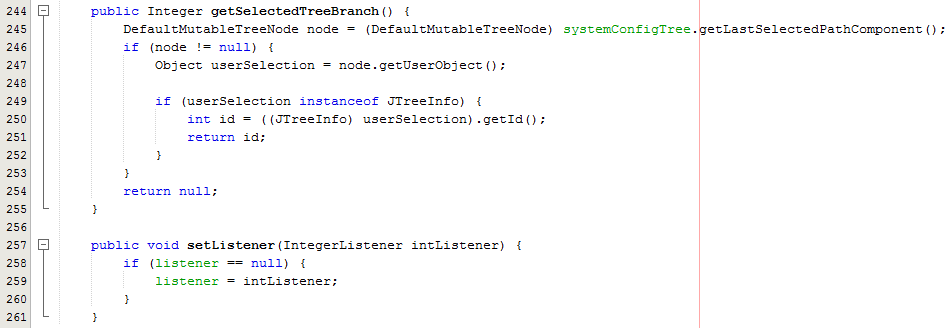


Fig. 10 – Extract from SystemConfigHome

To then use the JTree defined in SystemConfigHome class within a graphical user interface (GUI), I then just need to define and initialise a SystemConfigHome object within the client GUI and supply an ActionListener which will wait for the SystemConfigHome class to invoke a method on the listener object defined in the SystemConfigHome class and initialised through the client invoking setListener on the SystemConfigHome (this will be explained in more detail later in the Appendices, but shown in Fig. 11 below), and then add the SystemConfigHome to a container which will then display the JTree to the user.



Fig. 11 – Extract from SystemConfigFrame

For this project I decided to initialise the SystemConfigHome object within a class called SystemConfigFrame, which extended JFrame, and deals with invoking any methods on the model from user selections within the JTree, which is passed as an int value (the integer value assigned to the JTreeInfo) through the listener object (explained later in the appendices), as you can see from Fig. 11. The SystemConfigHome object is then added to the JFrame, again using the add() method. This implementation of the JTree, again adopts the MVC design pattern, enabling the view and the model to be loosely coupled.

* 1. **JFileChooser**

As this project consists of managing documents, it will require the user to be able to select a file for saving to the server, so I decided to implement a JFileChooser within both the create and update document dialogs, as you can see from below in Fig. 12. Firstly, to implement a JFileChooser I created a number of new classes that extend the FileFilter class and I implement 2 methods called accept(File file) and getDescription(), which are used to check the file passed as a parameter matches the desired extension and returns true, or false otherwise, and the other to return the description of files to filter for if this file filter is selected, as you can see from Fig. 13.

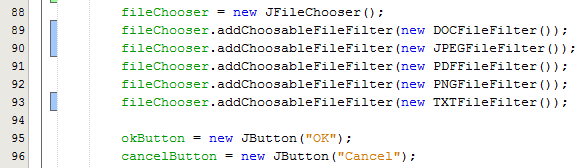


Fig. 12 – Extract from CreateDocument class

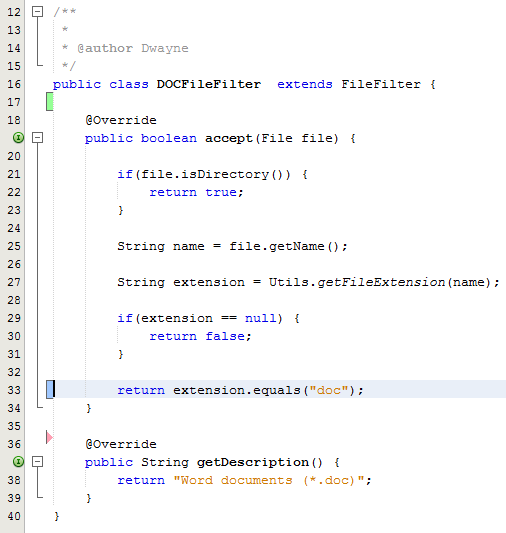


Fig. 13 – DOCFileFilter class

Once I had implemented the desired file filter classes, and declared and initialised a JFileChooser object, I can then invoke the add method on the JFileChooser object and pass the FileFilter class as parameter.

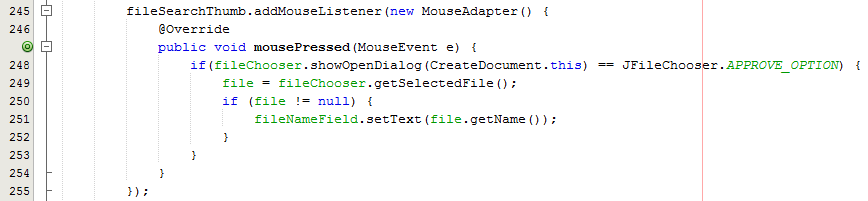


Fig. 14 – Extract from CreateDocument class

Now the JFileChooser is initialised and set up to allow the user to filter their file search, I have to invoke the showOpenDialog() method on the JFileChooser object which will pop up an open file dialog, allowing the user to select a file from their local file directory, and if the user selects a file and selects the OK option, then the system will return the selected file from the FileChooser and is then able to invoke an action on this file object, of which in this instance, I just set a text field within the graphical user interface to display the file name back to the user, for confirmation that they have selected the correct file, of which the user can then select OK to create the document, and then confirm the creation of the document.

Additionally, to the above explained components, I also implemented a number of other Java Swing and AWT components to assist me in providing data entry options for the user, and to display system state back to the user. The additional components are listed below:

* JButton – Is a push button, that can be clicked by the user
* JCheckBox – Is a checkbox that can be selected or deselected by the user
* JComboBox – Is a drop down list, where the user can select a value form a list
* JMenu – Is a pop up window, that is displayed when the user selects and item on a JMenuBar
* JSpinner – Is an input field, that allows the user to select a number or object from an ordered sequence.
* JTextField – Is a text input field, for 1 line of text.
* JPasswordField – Is a text input field, where the original character’s input is hidden from the display.
* JTextArea – Is a text input field, for multiple lines of text.
* JLabel – Is a display area, that can display either text or an icon.
* JSeperator – Is a general purpose component that implements a divide line, which can be used to separate menu options.

These additional components, again need to be added to a container to enable them to be displayed back to the user, through a GUI to enable the user to interact with the system.

1. **Containers and Layout Managers**

As explained in the report and earlier in this appendices, I decided to implement a number of Java Swing and AWT components, and as explained, this will require containers to group components together within the container, and then a layout manager to manage how these components will be laid out within the container.

The implementation of containers and layout managers, in essence go hand in hand, as a container needs to be instructed on how to manage the layout of the components added to a container, and therefore when I have created containers to hold components, I will then set the layout of the container by setting a layout manager to the container.

I am now going to go through the implementation of the different containers (such as JFrame, JDialog and JPanel) and layout managers (such as BorderLayout, GridBagLayout and FlowLayout), I have used for this development. For this project, I used a number of containers and layout managers, due to the wide variety of components that need to be displayed to the user through the GUI.

Firstly, I am going to discuss the different types of Containers I have adopted to assist me with displaying components.

JFrame – Is a top level container, with a title and a border

JDialog – Is a top level container, that provides a dialog window for message/input

JPanel – Is a lightweight general purpose container.

JTabbedPane – Is a general purpose container, that displays different groups of components, by clicking on a tab to display that group of components

JScrollPane – Is a lightweight general purpose container, that provides a scrollable view.

For the development of a GUI, the application needs to have at least one top level container. Now I have outlined the different components I have used within the development, I am now going to explain how I have implemented the different layout managers I have used.

* 1. **GridBagLayout**

GridBagLayout is one of the most flexible and most complex layout managers to implement, but enables me to layout components within a container horizontally, vertically, and of any size component, meaning that it provides me with freedom to align components within a container, to how the system requirements define.



Fig 15 – Extract from LoginForm class, extract from layoutComponents()

As you can see from Fig. 15, to implement grid bag layout I had to firstly set the layout of the panel in which my form is going to be within, by invoking setLayout() and passing as a parameter a new GridBagLayout object. I then had to declare and initialise a GridBagConstraints object, which is used to define the layout of any components added to the panel.

Once I have a GridBagConstrains object I then invoke methods from the GridBagConstraints class to set the constraints of any components added. The basic constraints are gridx() and gridy() which defines what position I am going to add a component on the screen, and gridwidth() and gridheight() determines what size the cell will take up on the screen.

* 1. **BorderLayout**

BorderLayout is one of the most widely used layout managers, and lays out components within a container that is divided up into five regions (NORTH, SOUTH, EAST, WEST, and CENTER), and is the default layout manager for the root content pane of a JFrame.

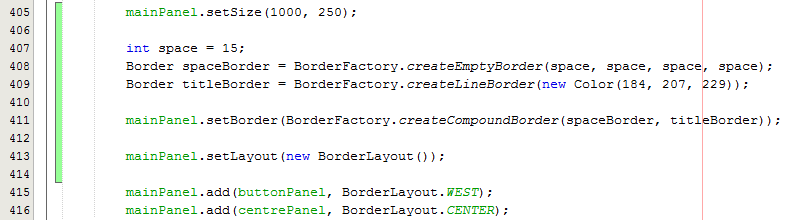


Fig. 16 – Extract from AppDetails class

As you can see from Fig. 16, to implement border layout I had to firstly set the layout of the panel in which my form is going to be within, by invoking setLayout() and passing as a parameter a new BorderLayout object.

Now the layout of the container has been defined, I then just need to add the components to the region of the container I require the component to be added to, and for the above example, I am adding a buttonPanel (JPanel) to the WEST region, by invoking add() method on the BorderLayout panel, and passing the component to be add (buttonPanel) and the region to be added to, which requires a call on the BorderLayout class to return the specific region, for example BorderLayout.WEST to add to the WEST region.

* 1. **FlowLayout**

The final layout manager I am going to discuss is the FlowLayout manager, which arranges components added to a container in a defined directional flow.

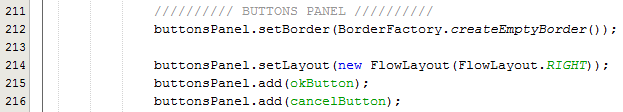


Fig. 17 – Extract from AppSearch class

As you can see from Fig. 17, to implement flow layout I had to firstly set the layout of the panel in which my form is going to be within, by invoking setLayout() and passing as a parameter a new FlowLayout object, and passing as a parameter an integer value to define the alignment flow of the components to be added to the container. To get the required in value for the alignment I can invoke a method on the FlowLayout class, which will return an integer value, which is associated with the required flow alignment.

Now the layout of the container has been defined, I then just need to add the components to the container, in the order in which I require them to be displayed. And from the example above, as you can see I have set the Layout to align from right to left, by passing FlowLayout.RIGHT as a parameter to the construction of the new FlowLayout object, and then add an OK button and a Cancel button which will result in the OK Button being added first to the panel, right aligned to the edge of the panel, and then when the Cancel button is added, the OK Button is pushed along to the left and the Cancel button then becomes the most right aligned button, and so on.

1. **Event Listeners**

An Event Listener is an interface that is responsible for handling events, that have occurred and passing an object back to an object to any objects that the listener has been assigned to, allowing for two objects that need to communicate between each other to not be coupled together resulting in the objects being loosely coupled.

For this development I implemented a number of custom listener interfaces that enable a number of different types of objects to be passed back as a response to an event occurring. As you can see from Fig. 18, to create a listener interface, I define an interface that has one method to implement, such as the StringListener interface in Fig. 18.

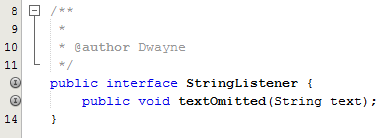


Fig. 18 – StringListener interface

Once I have defined the interface for the listener, I then have to create a instance of the listener object and implement all of the listener interface methods, for the StringListener example above, I just need to provide an implementation for the textOmitted(String text) method, as shown in Fig. 19.

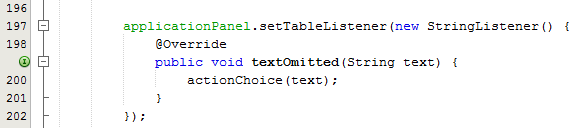


Fig. 19 – Extract from AppSearch class

As shown in Fig. 19 (and also briefly discussed in JTable section), the applicationPAnel object is an instance of ApplicationPanel, and as shown in an earlier section called JTable, the ApplicationPanel class has a setTableListener method which takes a StringListener object as parameter, and then this StringListener is then assigned to the listener field defined in the ApplicationPanel class.

So once this setTableListener() method is invoked on the ApplicationPanel object within AppSearch class, this assigns the StringListener object to the ApplicationPanel object, and going back to the ApplicationPanel class, waits for an action to occur on the ApplicationPanel (JTable), which will invoke a method defined within the ApplicationPanel class, which invokes the textOmitted() method on the StringListener object and passes a String value as parameter to the method, as shown in Fig. 20.

Now the method in which the ApplicationPanel (JTable) is going to communicate with its client(s) has been defined through the StringListener, I am going to show how the user interaction with the JTable is detected, initiating one of the methods to communicate with the ApplicationPanel.

Furthermore, there is not only the custom event listener interfaces that I have developed, but a wide range of standard listener objects for events, such as mouse clicks (MouseEvent, MousePressed etc.), or button pushes (ActionListsner).

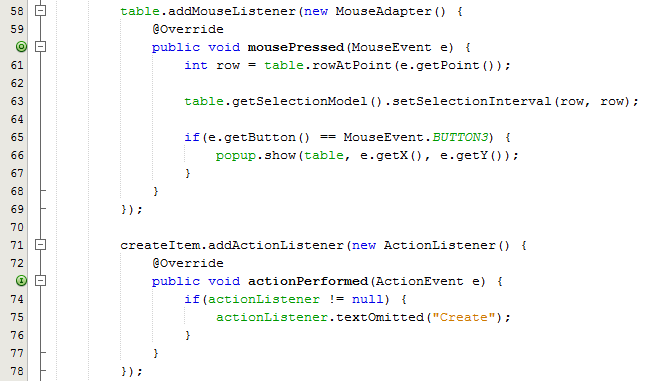


Fig. 20 – Extract from ApplicationPanel class

For the ApplicationPanel implementation, I decided to assign a MouseListener object to the JTable defined within the ApplicationPanel class, which listens for mouse events, such as mousePressed(), mouseClicked() etc. allowing for the ApplicationPanel to be notified when a defined action occurs on the JTable, in this example I assign a MouseAdapter, which allows me to provide an implementation for the mousePressed () method, which is passed a MouseEvent object (holds state of the mouse event such as mouse point position, what button was clicked, etc.), which allows me to then invoke methods on the MouseEvent object to get information to enable me to popup a menu, for the user to perform actions.

Now I have been able to open a JPopupMenu upon user interaction with the JTable, I am then able to provide options for further actions the user can perform, which have been done in the form of JMenuItem objects, which I then assign ActionListener objects to each of the menu items, meaning that if the user selects a specific option, an action will then occur, as shown in Fig. 20.

So if the createItem JMenuItem within the JPopupMenu was clicked on, then the actionPerformed() method is invoked, which checks to see if the ActionListener object declared in the ApplicationPanel class is not null (which was instansiated when the setTableListener method was invoked within AppSearch class), and if so then invokes the textOmitted() method on the StringListener object, which then passes the control back to the AppSearch class along with the String value, and I am then able to interpret that String to determine what action to perform, as shown in Fig. 19 and Fig. 21.

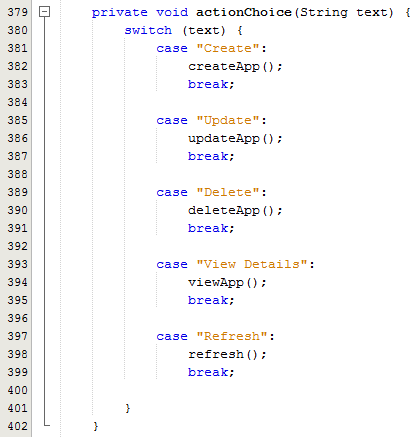


Fig. 21 – Extract from ApplicationPanel

N.B. actionChoice() method is invoked within the implementation of the textOmitted method for the StringListener, passing the String value from ApplicationPanel as parameter, which is then passed to a switch statement to decide what action to take, depending on the String value.

During this project I implemented a number of listeners such as:

* StringListener (Custom Listener for a String object)
* IntegerListener (Custom Listener for an Integer object)
* StringArrayListener (Custom Listener for an Array of String objects)
* ActionListener (Standard Listener)
* MouseListener (Standard Listener)

By implementing the graphical user interface in this manner, I am reducing the level of coupling between the model, and the view, meaning that I am adopting the model view controller design pattern.