RST JavaFX UI Pendant Report

Updated 12/1/2018

Introduction:

Since the last (monolithic) prototype program a new program was started from scratch following the Model-View-Controller design pattern. This pattern makes larger applications maintainable and use's separate FXML and CSS for the GUI view. The design pattern is also needed to correctly handle the updates coming from the robot which requires use of a listening UDP socket running on a different thread.

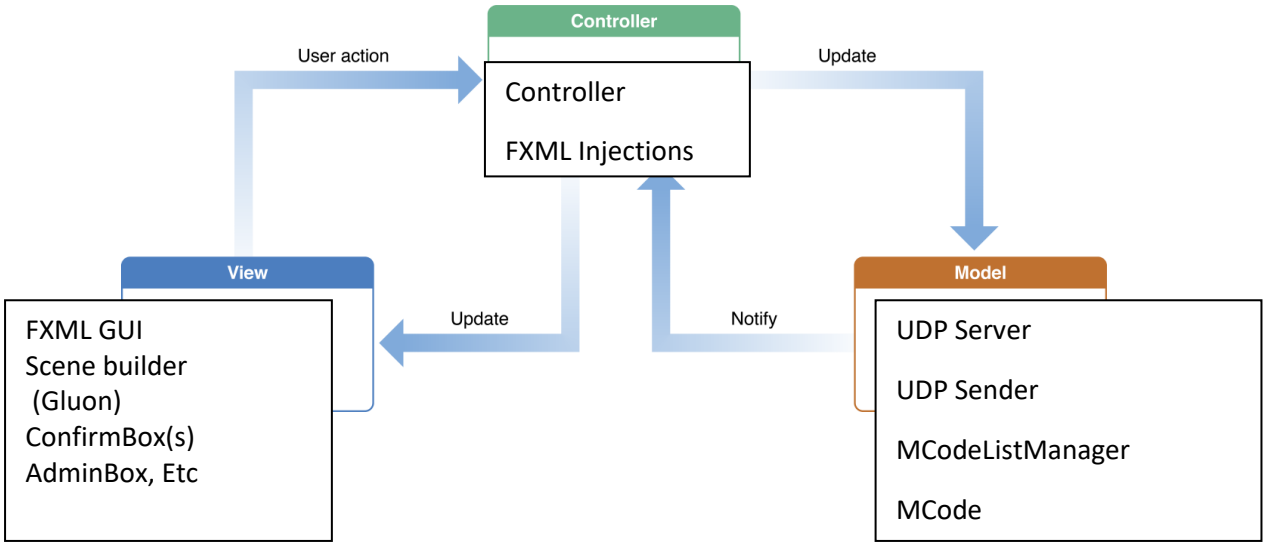

Figure 1: MCV design pattern which is implemented by default in JavaFX in the sample files.

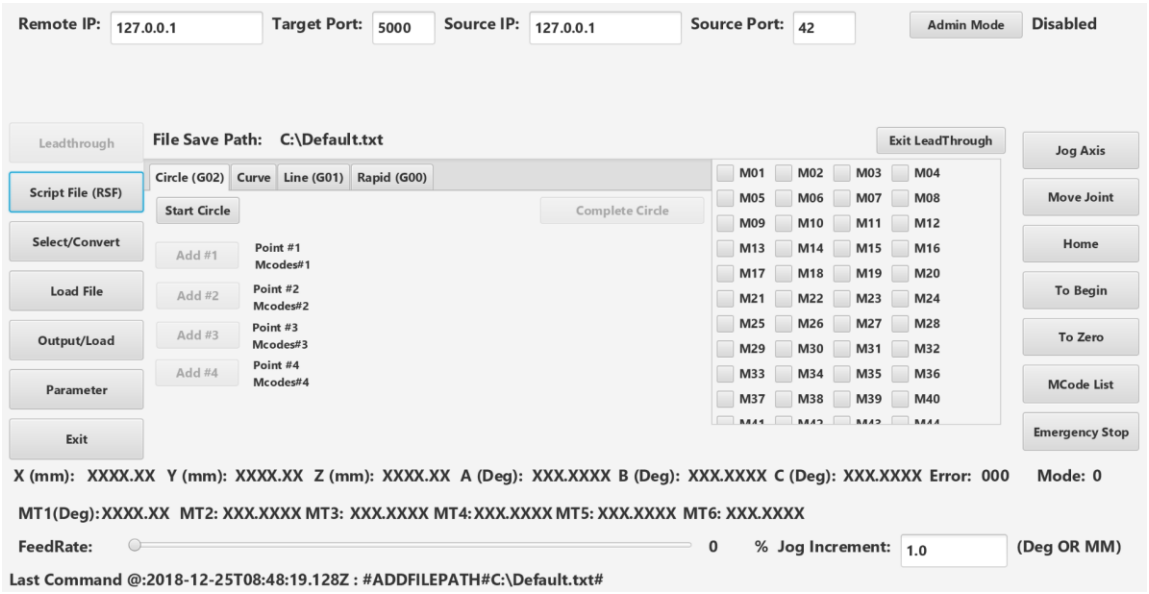

Figure 2: Main program screen in 'Leadthrough' mode, made in Gluon SceneBuilder

File structure breakdown and explanation:

1. Main.java

Similar to the "main" thread in a basic java program with some key changes for JavaFX. The class is declared as follows:

public class Main extends Application {

This means the Main class must extend (become a subclass) of the JavaFX 'Application' class, default methods within the parent 'Application' class are then replaced with 'overrides' as shown below.

```
@Override
public void start(Stage primaryStage) throws Exception{
```

This start method is automatically called as part of the JavaFX 'Application' class and is passed the 'primaryStage' object which represents the parent window of the JavaFX program. The 'throws Exception' keyword means that something inside this class may throw an exception, and suggests that whichever calling method have exception handling code, for example the FXML loader(discussed later) could throw an 'IOExpection' if an error occurs loading the fxml file.

Although a main method is included in the main class to accept arguments passed at the command line when the program is started, it is not required, as the 'start' method is automatically called by JavaFX:

```
public static void main(String[] args) {  //← OPTIONAL NOT REQUIRED
   launch(args);
}
```

The following code creates the observable robot position objects and passes them to the Controller:

```
//Creates observable position object
ObservablePosition myPosition = new ObservablePosition();
//Creates observer
PositionObserver observer = new PositionObserver(myController);
//Adds observer to position (links the two objects)
myPosition.addObserver(observer);
```

This code creates the UDP server in a separate thread and passes it the Position Objects to they can be updated with new information.

```
//Creates a new UDP server object, passes observable position to the constructor
UDPServer  MyUDPServer = new UDPServer(myPosition);
//Makes UDP Server object runnable
Runnable r = () -> {MyUDPServer.run();};
//Spawns a new thread with the runnable object
Thread UDPServerThread = new Thread(r);
```

```
UDPServerThread.setDaemon(true);
UDPServerThread.start();
```

Finally this is an event filter for the entire program which listens for any keys that are pressed and passes them to the JogKeyHandler method in the controller, for example short cut keys on the pendant.

```
MainScene.addEventFilter(KeyEvent.KEY_PRESSED, keyEvent -> {
    myController.JogKeyHandler(keyEvent);
});
```

2. MainScreen.fxml

Once the 'start' method is called the first action is to load the FXML file using the FXML loader.

FXMLLoader loader = new FXMLLoader(getClass().getResource("MainScreen.fxml"));

In this case this loads the FXML file for the main screen of the program, which in the MVC pattern we are calling the **'View'.** The FXML does not need to be coded by hand, but is created by the open source Guon Scene builder package(URL). The FXML file is written in XML, very similar to an HTML web page syntax, CSS is also supported to make it easy to set uniform formatting across the program.

There are several important aspects of the FXML file, which defines the *containers* and *controls* which make up the scene for the 'View'. Containers are things like horizontal and vertical boxes (HBox and VBox among others) which *hold* other objects via a parent-child relationship in the FXML tags (<Opentag> </Closetag>). Controls are things like buttons, labels and everything else that you interact with in the program's ***'View',*** these are the objects which are placed within the 'containers'. Important aspects of the FXML are discussed below.

1. Controller designation

The FXML file to use a controller must designate it by including its reference in the opening tag of the 'root' container of the scene.

fx:controller="sample.Controller"

In this example the controller it is looking for is located in the package (Folder) '*sample' and* will be called ***Controller.java***. In the main classes start method the controller object can be obtained with the following command once the FXML loader loads the file, allowing the controller to be passed to other objects for example model classes which form the business logic of the program (more on this later).

Controller myController = loader.getController();

2. FXML *fx:id*

Every object in the FXML view can optionally have a unique identifier called the ***fx:id***, for example labels or buttons would need to use this feature if they need to be referenced within the controller.

<Label fx:id="XAxis" …. …. </Label>

In the 'Controller' file, these fx:id's are 'Injected' by including the following commands which creates the association between the 'view' and 'controller'.

@FXML private Label XAxis;

Once the object is 'injected' it can be referenced throughout the controller program, by the following example:

XAxis.setText(String.valueOf(Robot_State[0]));

The example command takes an array double at index 0 and converts it to a String before passing it to the text property of the label ***XAxis***.

3. Controller.Java

This file forms the interface between the model and the view in the MVC pattern. As discussed earlier the controller a special java class designated from the FXML file, and can be accessed from the both the Model(s) and the View.

The controller file starts with a basic class declaration and includes an empty no argument constructor and initialize method. For this type of program most of the logic occurs in the controller as there is little underlying logic beyond passing the commands onto the main robot controller.

```
public class Controller {
  public Controller(){
  }
```

1. Initialization method

```
  @FXML
  private void initialize(){
```

The initialize method is needed to initialize variables and add listeners to objects examples of important variables used in that method are shown below:

```
    feedrateint = 0; //integer which holds the current feedrate for the rest of the program
    jogIncrementDouble = 1.0; //Default jog increment
    AxisJogEnable = false; //Boolean values to enable the jog men
    JointJogEnable = false; //**
    selectedFile = new File("C:\\Default.txt");       //Set default save file path for leadthrough
```

Within the initialize method listeners are added to various object for example the following listener, which is connected to the slider for adjusting the feedrate, the listeners are added via a java 8 lambda expression which is an easy way to add an anonymous inner class without having to include a separate class file.

```
    FeedRateSlider.valueProperty().addListener(e -> {
      Double temp = FeedRateSlider.getValue();
      feedrateint = temp.intValue();
    });
```

Similar to a Listener a binding can be used to directly 'bind' the value of the slider to the label displaying the value (FeedRateDisplay):

```
FeedRateDisplay.textProperty().bind(
    Bindings.format(
        "%.0f",
        FeedRateSlider.valueProperty()
    )
);
```

Another series of listeners are added to validate input on various text boxes within the GUI to prevent the user from setting invalid values. These listeners use outside classes **IpValidate** & **PortValidate**:

```
TargetIP.textProperty().addListener((observable, oldValue, newValue) -> {
  if (!IpValidate.validateIP(newValue)) {
    TargetIP.setText(oldValue);
  }
});
SourcePort.textProperty().addListener((observable, oldValue, newValue) -> {
  if (!PortValidate.validatePort(newValue)) {
    SourcePort.setText(oldValue);
  }
});
```

If an invalid value is set, the listener reverts the textbox to the previous value. A similar listener is then set for the Jog Increment text box, a try, catch block is used in case the user creates a number format exception by entering a non-numeric text value, as part of the exception handling the old value would be restored, however since the user is automatically presented with a numeric keyboard it is not easy for them to enter non-numeric characters by accident:

```
JogIncrement.textProperty().addListener(((observableValue, oldValue, newValue) -> {
  if (!isNumeric(newValue)) {
    JogIncrement.setText(oldValue);
  } else {
    try {
      jogIncrementDouble = parseDouble(JogIncrement.textProperty().getValue());
    } catch (NumberFormatException NumEx) {
      JogIncrement.setText(oldValue);
    }
  }
}));
```

Additional input validation listeners are placed on the input fields of the jog to position menus, to restrict them to numeric values only:

```
TPX.textProperty().addListener(((observableValue, oldValue, newValue) -> {
  if (!NumberInputValid.Check(newValue)) {
    TPX.setText(oldValue);
  }
}));
```

Below that another set of listeners are added to check if the displayed position of the robot has changed, and only updates the values held in the input textboxes if the displayed values change.

```
XAxis.textProperty().addListener(((observableValue, oldValue, newValue) -> {
   if (!newValue.equals(oldValue)) {
      TPX.setText(newValue);
   }
}));
```

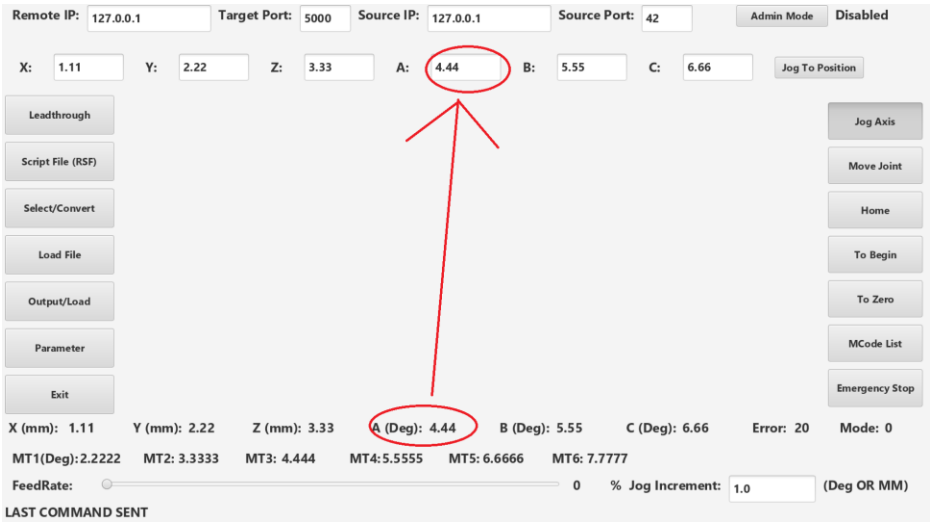

Figure 3: Listener which updates text box only when the robots position has changed.

The final block of code in the initialize method sets the type of keyboard to be displayed for each of the text input boxes which in all cases is numeric:

```
JogIncrement.getProperties().put("vkType", "numeric");
```

2. FXML Object injection

The next phase of the program involves the 'injection' of objects from the FXML 'View' into the controller so they can be used. This is achieved through the declarations shown in the examples below, where each object name must correspond to an FX:ID field in the FXML file. For example:
fx:id="MainPane"

```
@FXML
private BorderPane MainPane;
@FXML private TextField SourcePort;
@FXML private Slider FeedRateSlider;
@FXML private Label FeedRateDisplay;
@FXML private Label XAxis;
```

Etc.

3. Main controller methods

Once all of these objects are loaded the main methods of the program can be run. Any of these methods can be used from the view by adding its name to an objects **OnAction** property, for example see figure 4 for the mcode check boxes which activate the **#McodeEdit** method when they are clicked.

The more complex of those methods will be discussed here with the **McodeEdit** as the first example. This method is called by all of the Mcode boxes used in the program. When the method is called it is passed an action event by the calling method (**ActionEvent AE**), which provides information on the source of the event via **AE.getSource(),** using that information and string extraction via **AE.getSource().toString().substring(13, 15)** the index of the calling checkbox is found based on the position of its fx id in the event source string (Example: CheckBox[id=M**17**, styleClass=check-box]'M17'). Once this information is known further commands obtain the state of the calling checkbox via: **CheckBox SelectedBox = (CheckBox) AE.getSource()** and **SelectedBox.isSelected()).** Once the index and state of the MCode is known it is then updated in the Arraylist via: **mlistMGR.get(index).enabledProperty().setValue(X);**

```
@FXML
public void McodeEdit(ActionEvent AE) {
   //Modifies the Arraylist storing all the mcodes when the checkboxs are changed
   int index;
   String indexString;
   indexString = AE.getSource().toString().substring(13, 15);

   try {
      index = Integer.parseInt(indexString); // Casting, not the best way but works
      if ( AE.getSource() instanceof CheckBox) {
         CheckBox SelectedBox = (CheckBox) AE.getSource();
         if (SelectedBox.isSelected()) {
            mlistMGR.get(index).enabledProperty().setValue(true);
         } else {
            mlistMGR.get(index).enabledProperty().setValue(false);
         }
      }
   } catch (Exception Ex) {
      //Debugg
      System.out.println(Ex.toString());
      System.out.println(Ex.getMessage());
      System.out.println(Ex.getLocalizedMessage());
   }
}
```

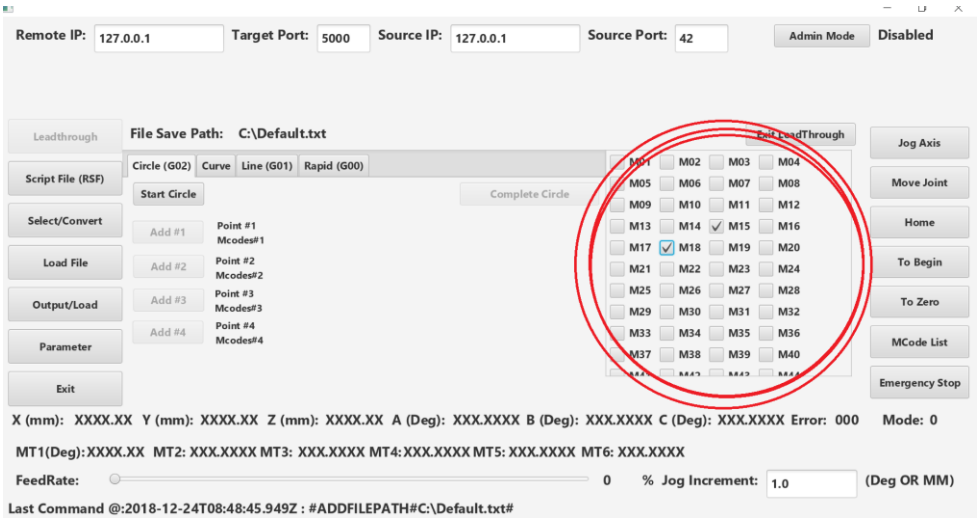

Figure 4: Mcode menu within the lead through mode.

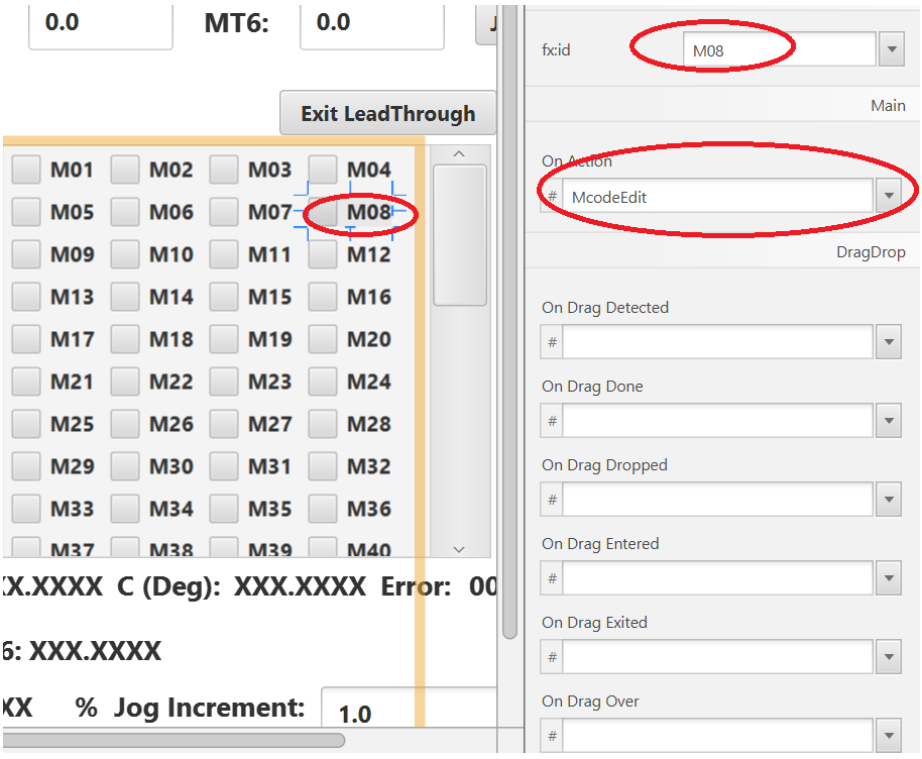

Figure 5: Shows the interaction between SceneBuilder and the Mcode Menu, with the FX:ID and On Action Method highlighted.

The next few commands are used in the lead through mode, these commands are very simple and move from one point to the next allowing the user to jog and then select the points they want to

stop and save. The Circle lead through method is included here for the purpose of an example as the other methods are very similar.

```
//Commands for circle mode in leadthrough
  @FXML
  public void StartCircle(){
    //Instructs the controller to start circle teach mode
    sendUDP("#CIRCLESTART#");
    //Disables itself and adds the next button
    BStartCircle.disableProperty().setValue(true);
    BCircleP1.disableProperty().setValue(false);
    ExitLeadThrough.disableProperty().setValue(true);

    //Disables other tabs while in leadthrough;
    LTCurveTab.disableProperty().setValue(true);
    LTLineTab.disableProperty().setValue(true);
    LTRapidTab.disableProperty().setValue(true);

    // Clears all labels
    CP1Label.setText("");
    CP1LabelM.setText("");
    CP2Label.setText("");
    CP2LabelM.setText("");
    CP3Label.setText("");
    CP3LabelM.setText("");
    CP4Label.setText("");
    CP4LabelM.setText("");
  }

  @FXML
  public void AddCircleP1(){
    //Disables itself
    BCircleP1.disableProperty().setValue(true);
    BCircleP2.disableProperty().setValue(false);
    //Sends network command
    sendUDP(CState.AddPoint(feedrateint) + mlistMGR.toString());
    CP1Label.setText(RobotAxisPostion());
    CP1LabelM.setText(mlistMGR.toString());
  }

  @FXML
  public void AddCircleP2(){
    //Disables itself
    BCircleP2.disableProperty().setValue(true);
    BCircleP3.disableProperty().setValue(false);
    //Sends network command
    sendUDP(CState.AddPoint(feedrateint) + mlistMGR.toString());
    CP2Label.setText(RobotAxisPostion());
    CP2LabelM.setText(mlistMGR.toString());
  }

  @FXML
  public void AddCircleP3(){
```

```
    //Disables itself
    BCircleP3.disableProperty().setValue(true);
    BCircleP4.disableProperty().setValue(false);
    //Sends network command
    sendUDP(CState.AddPoint(feedrateint) + mlistMGR.toString());
    CP3Label.setText(RobotAxisPostion());
    CP3LabelM.setText(mlistMGR.toString());
}

@FXML
public void AddCircleP4(){
    //Disables itself
    BCircleP4.disableProperty().setValue(true);
    BEndCircle.disableProperty().setValue(false);
    //Sends network command
    sendUDP(CState.AddPoint(feedrateint) + mlistMGR.toString());
    CP4Label.setText(RobotAxisPostion());
    CP4LabelM.setText(mlistMGR.toString());
}


@FXML
public void EndCircle(){
    //Instructs the controller to end circle teach mode
    sendUDP("#CIRCLESTOP#");
    //Disables itself and adds the next button
    BStartCircle.disableProperty().setValue(false);
    BEndCircle.disableProperty().setValue(true);
    ExitLeadThrough.disableProperty().setValue(false);

    //Enables other tabs while leaving leadthrough;
    LTCurveTab.disableProperty().setValue(false);
    LTLineTab.disableProperty().setValue(false);
    LTRapidTab.disableProperty().setValue(false);
}
```

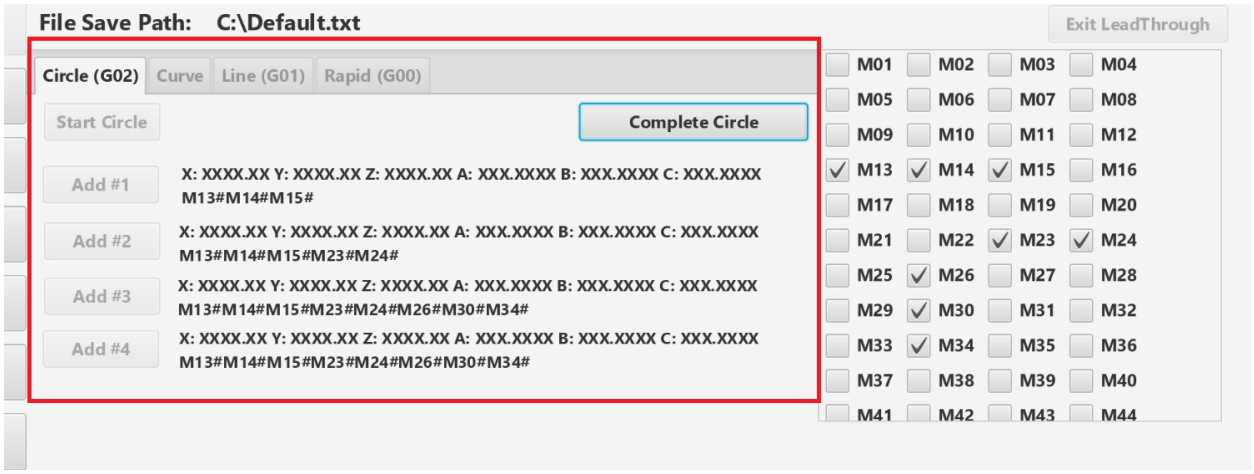

Figure 6: Example of use of the circle mode in lead through, the 6 axis coordinates of each point as well as active Mcodes are saved to the screen each time. Buttons are disabled during each mode so the user cannot leave the mode until it is completed.

Similar modes exist for Curve, Line and Rapid modes, all following the same design.

The next section holds the functions to start and end the lead through, these functions disable and reenable various buttons as well as displaying the main tab pane for the lead through menu (**LeadThroughParentVBox**), which is shown in figure 5 above.

```
@FXML
public void LeadThrough() {
   selectedFile = FileBox.display(selectedFile);
   LeadThroughParentVBox.setVisible(true);
   try {
      FilePathLabelDisplay.setText(selectedFile.getPath());
   } catch (Exception ex) {
      System.out.println(ex.toString());
   }
   sendUDP("#ADDFILEPATH#" + selectedFile.getPath() + "#");
   BLeadThrough.disableProperty().setValue(true);       //Disable the leadthrough button
}

@FXML
public void ExitLeadThrough() {
   //Hide the leadthrough menu and set button to enabled again
   LeadThroughParentVBox.setVisible(false);
   BLeadThrough.disableProperty().setValue(false);
   sendUDP("#ENDLEADTHROUGH#");
}
```

The Admin mode button, which launches a separate AdminBox.Java class:

```
@FXML
public void AdminDialog() {
   AdminEnable = AdminBox.display("ADMIN MODE","SWITCH TO ADMIN MODE?");
   if(AdminEnable){
```

```
        AdminLabel.setText("ENABLED");
        MainPane.setStyle(RedBgStyle);          //Set Red color to warn about admin mode
    } else {
        MainPane.setStyle(null);          //Reset color back to normal
        AdminLabel.setText("DISABLED");
    }
}
```

The Jog and Joint movement mode buttons, which are configured in a mutually exclusive toggle group:

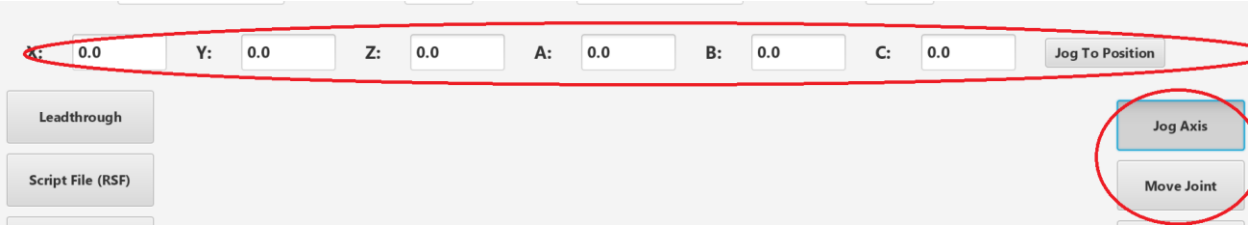

Figure 7: Toggle group for jog mode, also controls when hotkeys are enabled for jogging.

```
@FXML
public void JogAxisMode() {
    if (JogToggleButton.isSelected()) {
        AxisJogToPosition.setVisible(true);
        JointJogToPosition.setVisible(false);
        AxisJogEnable = true;
        JointJogEnable = false;
    } else {
        AxisJogEnable = false;
        AxisJogToPosition.setVisible(false);
    }
}
```

```
@FXML
public void JointAxisMode() {
    if (JointToggleButton.isSelected()) {
        JointJogEnable = true;
        AxisJogEnable = false;
        AxisJogToPosition.setVisible(false);
        JointJogToPosition.setVisible(true);
    } else {
        JointJogEnable = false;
        JointJogToPosition.setVisible(false);
    }
}
```

```
@FXML
public void JointToPosition(){
    // NOTE Zero value for MT0 when it is not in use
```

```
    String Command = "JOINTTOPOSITION#" + feedrateint + "#0#" + TPMT1.getCharacters() + "#" +
TPMT2.getCharacters() + "#" + TPMT3.getCharacters() + "#" + TPMT4.getCharacters() + "#" +
TPMT5.getCharacters()+ "#" + TPMT6.getCharacters();
    sendUDP(Command);
  }

  @FXML
  public void JogToPosition(){
    String Command = "JOGTOPOSITION#" + feedrateint + "#" + TPX.getCharacters() + "#" + TPY.getCharacters()
+ "#" + TPZ.getCharacters() + "#" + TPA.getCharacters() + "#" + TPB.getCharacters() + "#" + TPC.getCharacters();
    sendUDP(Command);
  }
```

Finally there is the update robot position method which is called by the UDP server from another thread, which will be discussed more in another section. This method updates all the onscreen values related to the robots axis and motor positions:

```
  public void updatePosition(double[] Robot_State) {
    LastRobotState = Arrays.copyOfRange(Robot_State, 0,13);
    // Sets all the labels along bottom of screen
    XAxis.setText(String.valueOf(Robot_State[0]));
    YAxis.setText(String.valueOf(Robot_State[1]));
    ZAxis.setText(String.valueOf(Robot_State[2]));
    AAxis.setText(String.valueOf(Robot_State[3]));
    BAxis.setText(String.valueOf(Robot_State[4]));
    CAxis.setText(String.valueOf(Robot_State[5]));
    // Sets all the labels along bottom of screen
    MT1Axis.setText(String.valueOf(Robot_State[7]));
    MT2Axis.setText(String.valueOf(Robot_State[8]));
    MT3Axis.setText(String.valueOf(Robot_State[9]));
    MT4Axis.setText(String.valueOf(Robot_State[10]));
    MT5Axis.setText(String.valueOf(Robot_State[11]));
    MT6Axis.setText(String.valueOf(Robot_State[12]));
    Double Estate = Robot_State[13];
    int EstateInt = Estate.intValue();
    Error_State.setText(String.valueOf(EstateInt));
    Double Mstate = Robot_State[14];
    int MstateInt = Mstate.intValue();
    Mode_State.setText(String.valueOf(MstateInt));
  }
```

The controller also includes a general method for passing commands to the main software via the IP and port information which is specified in the GUI as well as displaying the most recent command in the GUI itself:

```
  public void sendUDP(String Message) {
    UDPClient Sender = new UDPClient(SourceIP.getCharacters().toString(), TargetIP.getCharacters().toString(),
Integer.parseInt(SourcePort.getCharacters().toString()), Integer.parseInt(TargetPort.getCharacters().toString()));
    Sender.UDPSend(Message);
    Sender = null;
    System.gc();
```

```
    CommandLabelLog.setText("Last Command @:" + Instant.now() + " : " + Message);
  }
```

Several of the buttons have methods which open confirmation boxes or more complex menus in the case of the home button:

```
  @FXML
  public void toBegin() {
    Boolean toBeginChoice = ConfirmBox.display("To Begin", "Go to Begin Point?");
    if (toBeginChoice){
      sendUDP(CState.SpecialCommand(feedrateint, 1));
    }
  }

  @FXML
  public void toZero() {
    Boolean toZeroChoice = ConfirmBox.display("To Zero", "Go to Zero Point?");
    if (toZeroChoice){
      sendUDP(CState.SpecialCommand(feedrateint, 2));
      }
  }


  @FXML
  public void DisplayHome() {
    //Dummy holding variable for choice box selection
    int HomeChoice = 0;
    HomeChoice = HomeBox.display(AdminEnable);
      switch (HomeChoice) {
        case 1:
          System.out.println("Go home selected");
          sendUDP(CState.SpecialCommand(feedrateint, 3));
          break;
        case 2:
          System.out.println("Set home selected");
          sendUDP(CState.SetHome());
          break;
      }
  }
```

There are then a larger number of helper methods for jogging specific axis, these methods are triggered by hotkeys mapped later in the program:

```
  @FXML
  public void jogXPos() {
    //public static String Jog(int Feed, int Axis, boolean isNeg, Double Increment)
    sendUDP(CState.Jog(feedrateint, 0, false, jogIncrementDouble));
  }
…
  @FXML
  public void jogMT1Pos() {
    //public static String Jog(int Feed, int Axis, boolean isNeg, Double Increment)
    sendUDP(CState.Jog(feedrateint, 7, false, jogIncrementDouble));
```

```
  }


  public static boolean isNumeric(String s) {
    return s.matches("((-|\\+)?[0-9]+(\\.[0-9]+)?)+");
  }

  public String RobotAxisPostion() {
    String Postion = "X: " + XAxis.getText() + " Y: " + YAxis.getText() + " Z: " + ZAxis.getText() + " A: " +
AAxis.getText() + " B: " + BAxis.getText() + " C: " + CAxis.getText();
    return Postion;
  }
```

Hot keys are handled with the following code, which only activates when either the Jog or Joint movement modes are enabled. This command is

```
@FXML
  public void JogKeyHandler(KeyEvent event) {
    //System.out.println("Event Detected Key: " + event.getCode().getName() + " Name: " +
event.getEventType().getName());
    //Debugg: Get Key Press for Switch Statement
    //System.out.println(event.getSource().toString());

    String KeyPressed = event.getCode().getName();
    //System.out.println(KeyPressed);

    //Check for special buttons on the pendant F5-F7
    //Context of these buttons changes depending on the mode

    switch (MstateInt) { //Checks if the pendant is in manual mode
      case 0: //Manual mode
        if (event.getCode() == KeyCode.F6) {
          System.out.println("F5 Detected in manual mode" + event.getCode().getName());

          //Since jog and jog joint are toggle buttons...
          if (MainPane.getScene().focusOwnerProperty().get() instanceof ToggleButton){
            ToggleButton TActiveB =(ToggleButton) MainPane.getScene().focusOwnerProperty().get();
            TActiveB.fire();
            //Debugg
            //System.out.println("Selected Button to fire" + TActiveB.toString());
          }

          //If a button is in focus fires the selected button
          if (MainPane.getScene().focusOwnerProperty().get() instanceof Button){
            Button ActiveB =(Button) MainPane.getScene().focusOwnerProperty().get();
            ActiveB.fire();
            //Debugg
            //System.out.println("Selected Button to fire" + ActiveB.toString());
          }

          //Debugg
          //System.out.println("Key Event" + ke.toString());
```

```
            //System.out.println("Focus owner" + MainPane.getScene().focusOwnerProperty().get().toString());
          }
          if (event.getCode() == KeyCode.F7) {
            System.out.println("F6 Detected in manual mode" + event.getCode().getName());
            JogToggleButton.fire(); //Enter the jog mode
          }
          if (event.getCode() == KeyCode.F8) {
            System.out.println("F7 Detected in manual mode" + event.getCode().getName());
            JointToggleButton.fire(); //Enter the joint jog mode
          }
          break;
        case 1:
          break; // do nothing because pendant is locked
        case 2: // Run mode
          if (event.getCode() == KeyCode.F6) {
            System.out.println("F5 Detected in Run mode" + event.getCode().getName());
            //Code to run the program
            sendUDP("#RUNPROGRAM#");
          }
          if (event.getCode() == KeyCode.F7) {
            System.out.println("F6 Detected in Run mode" + event.getCode().getName());
            //Code to Pause the program
            sendUDP("#PAUSEPROGRAM#");
          }
          if (event.getCode() == KeyCode.F8) {
            System.out.println("F7 Detected in Run mode" + event.getCode().getName());
            //Code to stop the program??
            sendUDP("#STOPPROGRAM#");
          }
          break;
      }


        //For Axis Jog mode only when enabled and in manual mode (0)
      if (AxisJogEnable && MstateInt == 0) {
        switch (KeyPressed) {
          case "A":
            jogXPos();
            break;
          case "B":….
…
        if (JointJogEnable && MstateInt == 0) {
          switch (KeyPressed) {
            case "A":
              jogMT1Pos();
              break;
            case "B":
              jogMT1Neg();
              break;
            case "C":
etc…
```

This keyevent is generated by an event filter placed in the Main.Java class which filters key events for the entire program not just one scene:

```
MainScene.addEventFilter(KeyEvent.KEY_PRESSED, keyEvent -> {
   myController.JogKeyHandler(keyEvent);
});
```

4. UDPServer.java

1. Introduction

For this program the UDPserver would in some respects be considered part of the 'Model' within the MVC pattern. An extra complication is added due to the need for the server to run in a different thread from the JavaFX main application. The reason for this is the while loop inside the UDPServer which run infinity waiting for packets on port 5000 and parsing those packets into the data structure needed by the program to display the robots position and error states.

2. Threading

The multi-threading is implemented via 'runnable' interface by extending the 'Thread' class and including a ***run()*** method. Relevant code below:

```
public class UDPServer extends Thread {

    public void run() {
```

The calling methods in the main class creates the separate thread using the following block of code, which first instantiates a UDPserver object, makes that object 'runnable', and finally starts the thread using the ***.start()*** method:

```
UDPServer  MyUDPServer = new UDPServer(myPosition);
Runnable r = () -> {MyUDPServer.run();};
Thread UDPServerThread = new Thread(r);
UDPServerThread.setDaemon(true);
UDPServerThread.start();
```

3. UDP Server loops to accepts packets

Once inside the new myUDPserver thread several commands start the server, binding it to port 5000 of the localhost (127.0.0.1) interface. Just before the while loop the DatagramSocket is created, once inside the infinite while loop a byte array (buf) is created to hold the incoming datagram packets which are held in 'packet'. The while loop then waits(blocks) until a packet is received with the method ***udpSocket.receive(packet)*** and the contents of which are dumped to the string variable ***msg*** via the following command ***msg = new String(packet.getData()).trim().***

```
this.port = 5000;
initAddr = InetAddress.getByName("127.0.0.1");
```

```
//Creates UDP socket listener
this.udpSocket = new DatagramSocket(this.port, initAddr);
while (true) {
   byte[] buf = new byte[256];
   DatagramPacket packet = new DatagramPacket(buf, buf.length);
   // blocks until a packet is received
   udpSocket.receive(packet);
   msg = new String(packet.getData()).trim();
```

4. UDP server parses the incoming data

Once a packet of string data has been accepted by the program it must be determined to be valid and converted into the correct format. This is achieved in several steps outlined in the code below. First the msg packet is split based on the '#" delimiter into an array of the length 14 via the **String.split("delimiter")** and **Arrays.copyOf()** method. Regardless of the contents of the string the array will have length 14 with the extra places occupied by null values.

Once the String array is split it is parsed with a Java 8 Array stream method, which uses a filter to remove null and empty strings and non numeric values using the **isNumeric()** method.

The array is then tested with an if statement to see if it still has length 14 with numeric values as these are the only values allowed at this point. If this condition is met the string array values are converted into doubles, and passed into a new array called ***Robot_State*** and passed to the observable position object.

```
String[] ss = Arrays.copyOf(msg.split("#"),15);

ss = Arrays.stream(ss)
      .filter(s -> (s != null && s.length() > 0 && isNumeric(s)))
      .toArray(String[]::new);

if (ss.length == 15){
   double Robot_State[] = new double[15];
   for (int i = 0; i < ss.length; i++) {
         Robot_State[i] = parseDouble(ss[i]);
   }
   position.setPosition(Robot_State);
} else {
            // DEBUGGING MESSAGES
   System.out.println("Array was not parsed correctly, should be numeric only with length 15");
   System.out.println("Debug(message contents):" + msg);
            }

public static boolean isNumeric(String s) {
   return s.matches("((-|\\+)?[0-9]+(\\.[0-9]+)?)+");
}
```

5. UDPClient.Java

This part of the program is designed to send a single packet one at a time, it accepts the following constructor:

```
public UDPClient(String sourceAddr, String destinationAddr, int sport, int tport)  {
```

All of this information is obtained from the top bar in the GUI:

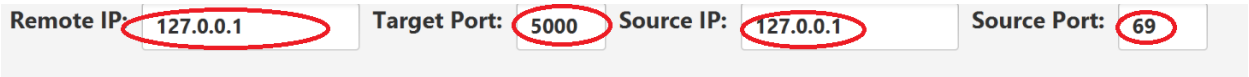

Figure 8: Information for sending packets to the server

Only once the UDP send method is called containing the relevant message is the actual packet constructed and sent via (Code condensed):

```
public void UDPSend (String Message)  {
        udpSocket = new DatagramSocket( this.sourcePort,this.localAddress);
        byte[] b = Message.getBytes(StandardCharsets.UTF_8);
        DatagramPacket p = new DatagramPacket(b, b.length, serverAddress, targetPort);
        this.udpSocket.send(p); // Sends the packet
```

And then the socket is closed and the memory cleaned up:

```
     udpSocket.close();
    //Clears the memory by running the garbage collector
    System.gc();
```

6. ObservablePostion.java

This class is used to hold the robots state when received from the main software by the UDP server class, and extends the observable class, the main purpose of this class is check if the position has actually changed or the robot is just reporting the same position over and over again, if the position has changed it notifies the observer(s):

```
public class ObservablePosition extends Observable {

  public ObservablePosition(){
  }

  public double[] getPosition() {
        return Position;
  }

  public void setPosition(double[] positionInput){
    if(Arrays.equals(Position,positionInput)){
      //System.out.println("Debug: Arrays have NOT Changed");
```

```
    } else {
      //System.out.println("Debug: Arrays HAVE Changed");
      //Updates the position to the new position
      Position = positionInput;
      //Notify observers and send out an update
      setChanged();
      notifyObservers();
    }
  }
```

7. PositionObserver.java

This is the observer which is attached to the Position observer, when the two objects are created in the main method. From main:

```
    //Creates observable position object
    ObservablePosition myPosition = new ObservablePosition();
    //Creates observer
    PositionObserver observer = new PositionObserver(myController);
    //Adds observer to position (links the two objects)
    myPosition.addObserver(observer);
```

```
public class PositionObserver implements Observer {
  private ObservablePosition myPosition;
  private Controller obController;
  //Observer which is attached to the robot position object, when the observed values change
  //Updates are send to the main GUI controller

  public PositionObserver(Controller myController){
    this.obController = myController;
  }
  @Override
  public void update(Observable observable, Object arg){
    myPosition = (ObservablePosition) observable;
    //Debugg
    //System.out.println("Position Observer has been updated" +
Arrays.toString(myPosition.getPosition()));
    //Sends the update to the application thread
    Platform.runLater(() -> {obController.updatePosition(myPosition.getPosition()); });
    //Garbage Collect
    System.gc();
  }}
```

The purpose of this class is to update the main method when the position of the robot (or error state or mode) is changed. This occurs with the following special command:

```
Platform.runLater(() -> {obController.updatePosition(myPosition.getPosition()); });
```

The run.Later command is used because the UDPServer, PositionObserver and ObservablePosition are in a separate thread from the main javaFX thread, in this way the GUI is updated once it is convenient

8. MainWrapper.java

This is a wrapper class that wraps the main method in such a way to correctly enable the virtual keyboard.

```
public class MainWrapper {
   //This is wrapper class to enable embedded mode, virtual keyboard, and touch.

   public static void main(String[] args) throws Exception
   {  // application - package name
      Class<?> app = Class.forName("sample.Main"); //Wraps the main class
      Method main = app.getDeclaredMethod("main", String[].class);
      System.setProperty("com.sun.javafx.isEmbedded", "true");
      System.setProperty("com.sun.javafx.touch", "true"); //Enables touch screen
      System.setProperty("com.sun.javafx.virtualKeyboard", "javafx"); //Enables on screen keyboard
      Object[] arguments = new Object[]{args};
      main.invoke(null, arguments); // calls the main class with no arguments
   }
}
```

9. MCode.Java

This is a javabean which represents the state of a specific Mcode, this is all done with a SimpleBooleanProperty:

```
public class MCode implements Serializable {
   private BooleanProperty enabled = new SimpleBooleanProperty();
   //Empty constructor
   public MCode(){
   }
   //Full Constructor
   public MCode(boolean state){
      this.enabled.setValue(state);
   }
   //Set and get methods
   public boolean isEnabled() {
      return enabled.get();
   }
   public BooleanProperty enabledProperty() {
      return enabled;
   }
}
```

10. MCodeListManager.Java

This is an Array list which holds 99 mcode java bean objects and adds some extra functionality to the list the main functions are the constructor which starts by creating 99 mcodes set to false, and the **toString()** method which produces concatenated list of mcodes which can be appended to other commands using the format: **M01#M02#M22#M99#**

```
public class MCodeListManager extends ArrayList<MCode> {
  //Extends the ArrayList class
  //Extends how the Mcode list is created, so it starts with 99 Mcode objects at index 1-99
  //Adds a toString method with concatenates all the mcode in the list into a formatted string separated by
  //# signs for example M01#M02#M05#M99# etc.

  public MCodeListManager(){
    //mlist = new ArrayList<MCode>(99);
    ListIterator iter = this.listIterator();
    for (int i=1; i < 100 ; i++ ){
      iter.add(new MCode(false));
    }
  }

  public String toString(){
    String McodesString = "";
    for (int i=1; i < 99 ; i++ ){
      boolean active = (this.get(i).isEnabled());
      if(active){
        McodesString = McodesString + "M" + i + "#";
      }
    }
    return McodesString;
  }
}
```

11. UDP Testing program

This program is used to simulate the robot main software and sends a stream of robot coordinates at 100 ms intervals to the GUI interface.

The program is a jar file **UDP_Testing_Code.jar**, which is executed via the following command:

java -cp UDP_Code.jar com.pgx.java.socket.UDPClient

And Produces the following output when run on the same host as the gui interface 127.0.0.1:

```
-- Running UDP Client at  --
Start Time:6538873809608
1.11#2.22#3.33#4.44#5.55#6.66#1.1111#2.2222#3.3333#4.444#5.5555#6.6666#7.7777#0#0#
Current Time: 1 Difference since last packet: 1
1.11#2.22#3.33#4.44#5.55#6.66#1.1111#2.2222#3.3333#4.444#5.5555#6.6666#7.7777#1#0#
Current Time: 101 Difference since last packet: 100
```

1.11#2.22#3.33#4.44#5.55#6.66#1.1111#2.2222#3.3333#4.444#5.5555#6.6666#7.7777#2#0#
Current Time: 203 Difference since last packet: 102
1.11#2.22#3.33#4.44#5.55#6.66#1.1111#2.2222#3.3333#4.444#5.5555#6.6666#7.7777#3#0#
Current Time: 310 Difference since last packet: 107
1.11#2.22#3.33#4.44#5.55#6.66#1.1111#2.2222#3.3333#4.444#5.5555#6.6666#7.7777#4#0#

12. ConfirmBox.Java, AdminBox.Java, HomeBox.Java, FileBox.Java

Are all variations on the confirm box class and provide a pop-up which covers the entire screen in some cases return a Boolean output, these classes are very simple and should be self explanatory I will provide explanations for the more complex commands below:

Since these are independent javafx classes, they have their own scene and controller, and therefore lack access to the main controller method.

For example the command below adds a listener for the F6 key so that whenever it is pressed the selected button (if it is a button) is fired. The example below is from the HomeBox class which only contains buttons.

```
scene.addEventFilter(KeyEvent.KEY_PRESSED, keyEvent -> {
   if (keyEvent.getCode() == KeyCode.F6) {
      // Debugg
      // System.out.println("key event detected" + keyEvent.toString());
      // If a button is selected fire it
      if (scene.focusOwnerProperty().get() instanceof Button) {
         Button ActiveB = (Button) scene.focusOwnerProperty().get();
         ActiveB.fire();
      }
   }
});
```

In the admin box class this listener is used differently, because the scene contains both buttons and password fields, if the F6 key which we are using as the enter key is pressed selection focus is moved to the confirm button allowing the user to press F6 again to leave the dialog.

```
scene.addEventFilter(KeyEvent.KEY_PRESSED, keyEvent -> {
   if (keyEvent.getCode() == KeyCode.F6) {
      //Debugg
      //System.out.println("key event detected" + keyEvent.toString());
      //If a button is selected fire it
      if (scene.focusOwnerProperty().get() instanceof Button) {
         Button ActiveB = (Button) scene.focusOwnerProperty().get();
```

```
            ActiveB.fire();
        }
        //If password box is selected move to confirmation box
        if (scene.focusOwnerProperty().get() instanceof PasswordField) {
            yesButton.requestFocus();
        }
    }
});
```

And finally in the file class the command is handled slightly differently, as it container text fields and buttons, once again if the F6 key is pressed the focus is shifted to the next button which would confirm the selection and leave the dialog.

```
scene.addEventFilter(KeyEvent.KEY_PRESSED, keyEvent -> {
    if (keyEvent.getCode() == KeyCode.F6) {
        // Debugg
        // System.out.println("key event detected" + keyEvent.toString());
        //If a button is selected fire it
        if (scene.focusOwnerProperty().get() instanceof Button) {
            Button ActiveB = (Button) scene.focusOwnerProperty().get();
            ActiveB.fire();
        }
        //If enter box is selected move to confirmbox
        if (scene.focusOwnerProperty().get() instanceof TextField) {
            Confirm.requestFocus();
        }
    }
});
```

13. Extra Files (CSS)

Two small CSS files are included and mainly set font sizes for various objects.

Possible improvements

The code could be cleaned up by placing more functionality out of the controller and into java beans, for example for the lead through functions. This would make the controller program smaller and more concise and would allow the java beans to be accessed from menus outside of the main menu.

Future and remaining work

Most future work with the program involves testing it with the main software. As well as functionality for a few more buttons and data sets. The buttons that need to be added are (Script File, Select, Convert, Load File, Output/Load, Parameter). The parameter menu will need to have a way to edit the Access formatted database from the main software this is planned to be made available in a .CSV file.

Other functionality would involve adding detailed handling of error codes.