Criterion C: Development

Table of Content

Table of Content **1**

1. Arduino **2**
1.1 Initialization of variables 2
1.2. Void loop 3
1.3. Get input 3
1.4. Actuation of cap and reading the sensors 4
1.5 Average + Final results 5

2. Java **5**
2.1. UML Structure and OOP inheritance 5
2.2. LaunchWindow.java 6
2.2.1 - Button to create a new class 6
2.3. GatherDataMenu.java 7
2.3.1 - Initialize a transmission with Serial Monitor 7
2.3.2 - Serial transmission with Java 8
2.3.3 - String methods to store measurements in arrays 9
2.3.4 - Multithreading in Java 9
2.4 Save Data 10
2.4.1 Create connection with a database 10
2.4.2 Close database connection 10
2.4.3 Create tables in database (Statement) 10
2.4.3 Insert in database (Prepared statement) 11
2.4 Create Table 11
2.4.3 2D array for JTable 11
2.5 Load Data Menu 12
2.5.1 Select Database using JFileChooser 12
2.5.2 Search parameters in tables 13
2.6 LoadTable 13
2.6.1 Extract data and loop through ResultSet 13

3. Bibliography **15**

1. Arduino

To program in the Arduino program, I used the standard Arduino IDE. I'll present the methods in chronological order of the process. Look at Appendix 2 for full code.

1.1 Initialization of variables

```
#include <Servo.h> //This library allows an Arduino board to control servo motors

  //Variables to action the servo motor for the actuationTime
   unsigned char ledState = LOW;
   unsigned char State = LOW;
   unsigned char lastState = LOW;
   unsigned long ledCameOn = 0;

   unsigned long actuationTime = 2; //STORES THE ACTUATION TIME OF THE CAP (in seconds)

   int push; //Stores the number of cycles
   boolean EXIT = false; //Enables to exit ther loop of the actuation of the servo

  int timer = (actuationTime*10); //Amount of measurements per cycle

  //Variable for sensors
  unsigned long previousMillis = 0;
  unsigned long Time;
  int interval1 = 100; //Constant time between measurements in cycles.

  //Array for to store the measurements of the volume
  float VolumeSensor[50];

  //Variables for the pressure measurements
  float AveragePress[50];
  float averageA1 = 0; //Stores the 1 pressure sensor

  //varibles for differential pressure mreasurements
  float AverageDiff[50];
  float averageA2A3 = 0; //Stores the average of the differences of the two pressure sensors

  int END = 1; //Used to end the arduino process once enough cycles have been done

int pos = 0; // variable to store the servo position

Servo myservo; //Creates a servo object called "myservo"
```

Figure 1 - Screenshot of initialization of variable in the Arduino IDE

Before the Arduino process starts, the needed variables are initialised in *figure 1*. Each array has 50 elements because it is my client's maximum possible amount of cycles per bottle. In this Arduino program, I am not using objects because it is a straightforward process. Hence, I require several global variables.

1.2. Void loop

```
void loop() {

  getTime();
  ServoON();
  CycleSummary();

}
```

Figure 2 - Screenshot of void loop method

The method in *Figure 2* loop until the user decides to end the application by inputting the integer 9.

1.3. Get input

This method has a while loop that waits for user input. If the user presses on End in the Java application, it will send the number 9 through the Serial connection and the Arduino will run FinalResults().

```
void getTime(){

  //Get the number of seconds
  delay(100); Serial.println("Press ENTER to start or input 9 to quit:"); //Prompt User for Input

  while (Serial.available() == 0) { //Loops until the user enters a value to the Serial Monitor

  }

  END = Serial.parseInt(); //END stores the value from the user input

  if(END == 9){
    FinalResults(); //calls the method to end the application
  }

  State = HIGH;
  push = push + 1;
  EXIT = false;

  Serial.println("-------------------------------------------------------");   delay(100);
  Serial.print("Cycle "); Serial.println(push);  delay(100);
  Serial.print("the cap will be pressed for "); Serial.print(actuationTime); Serial.println(" seconds");  delay(100);
  Serial.println("-------------------------------------------------------");
  delay(100);

  actuationTime = actuationTime * 1000; //This calculation is needed because the program counts in milliseconds
  }
```

Figure 3 - Screenshot of getTime method

1.4. Actuation of cap and reading the sensors

```
void ServoON(){
  do{
    // If the Servo has been on for longer than the actuation time seconds then turn it off.
    if(servoState == HIGH) //Checks if the servo is on
    {
      if(millis()-servoCameOn > actuationTime){ //Checks if the current time minus the time that it came on is greater than the actuationTime

        myservo.write(0); //Resets the servo position
        servoState = LOW;
        EXIT = true;

        serialFlush(); //calls this method in order to clear the Serial.monitor (avoid errors of automatically inputing without user input)
        lastState = LOW;
        State = LOW;
      }
    }

    if(State != lastState) //If the current button state is different than the last button state
    {
      lastState = State; //last state is now the current state
      if((State == HIGH) && (servoState == LOW)) //If both the state of the button is high and the servo is not been activated
      {

        myservo.write(-180); //Servo is activated
        servoState = HIGH; //State of servo is High
        servoCameOn = millis(); //stores when the servo has been activated

        //Method reads the sensors while the cap is pressed
        readSensors();
      }
    }
  }while(EXIT != true); //Loop while Exit = false
}
```

Figure 4 - Screenshot of ServoON method

This method is used to press on the cap for $actuationTime * 1000$ amount of milliseconds. The challenging algorithmic thinking resides in the loop involved to press the servo for a specific amount of time. I use the millis() method to store the amount of time that has passed since the program has started running and 3 flags to actuate the Servo. There is a notable difference between "servoState" and "State" because "servoState" stores the physical state of the servo while "State" stores the state of the button.

```
  millisNow = millis(); //the time before the measurements

do{
      //if half 0.1 (value of interval1) seconds has passed, we get the readings of A1, A2, A3
      if (millis() - previousMillis >= interval1) {

        previousMillis = millis(); //stores the time when the measurements start

        //Stores the values of the pressure sensor that is alone A1
        PressAlone = analogRead(A1);
        Serial.println(PressAlone);
        averageA1 = averageA1 + PressAlone; //adds all the values together for the average


        //Stores the values of the two pressure sensors that will be differenced then added to a variable for the average.
        PressSensor1 = (analogRead(A2));
        PressSensor2 = (analogRead(A3));
        diffInPres = (PressSensor2 - PressSensor1);
        averageA2A3 = averageA2A3 + diffInPres; //adds all the values together for the average
      }

 }while(millis()-servoCameOn < actuationTime-10); //Loop while the servo has been on for less than or equal to the set time
                                                  //actuationTime-10 to avoid an extra measurements
}
```

Figure 5 - Screenshot of loop to read the sensors

Similarly to *Figure 4*, I use the millis() method to measure each 0.1 seconds in *Figure 5*. My client asked me to measure each 0.1 seconds, as it filtered out outliers (as seen in *Appendix 1.5*).

1.5 Average + Final results

The respective arrays are used to store the averages for each cycle, as push stores the number of cycles. Since *timer* stores the number of measurements per cycle, it divides the sum of all values to get an average.

```
//gets the average for the A1 sensor
AveragePress[push-1] = (averageA1 / (timer));

//gets the average for the difference in pressure A2 and A3
AverageDiff[push-1] = (averageA2A3 / (timer));

//Stores the reading of the water sensor at the end of each cycle.
VolumeSensor[push-1] = analogRead(A0);
```

Figure 6 - Screenshot of storing the information

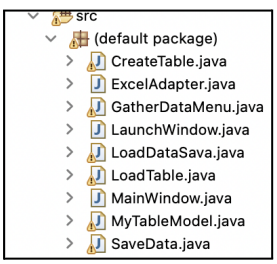
In Figure 7, the program extracts the values from the arrays and injects them into Strings to print them to the Serial monitor.

```
for(int c = 0; c < push; c++){    //Loop to create the Strings
  Vol = Vol + VolumeSensor[c] + " ";
  Pres = Pres + AveragePress[c] + " ";
  Diff = Diff + AverageDiff[c] + " ";

}
//Print the strings to the Serial monitor
Serial.print(push);
Serial.print("Pres" + Pres);
Serial.println("Dif" + Diff + "|");
Serial.println("Vol" + Vol + "?");

delay(100);
exit(0);
```

Figure 7 - Screenshot of final method

2. Java

2.1. UML Structure and OOP inheritance



My project has 7 classes that contain GUI and a "menu" of the application, and 2 classes from which methods are used. The main method can be found in MainWindow.java. This class structure enabled me to create each GUI as an object. In each class, I use a constructor to build the GUI.

Figure 8 - Class structure.

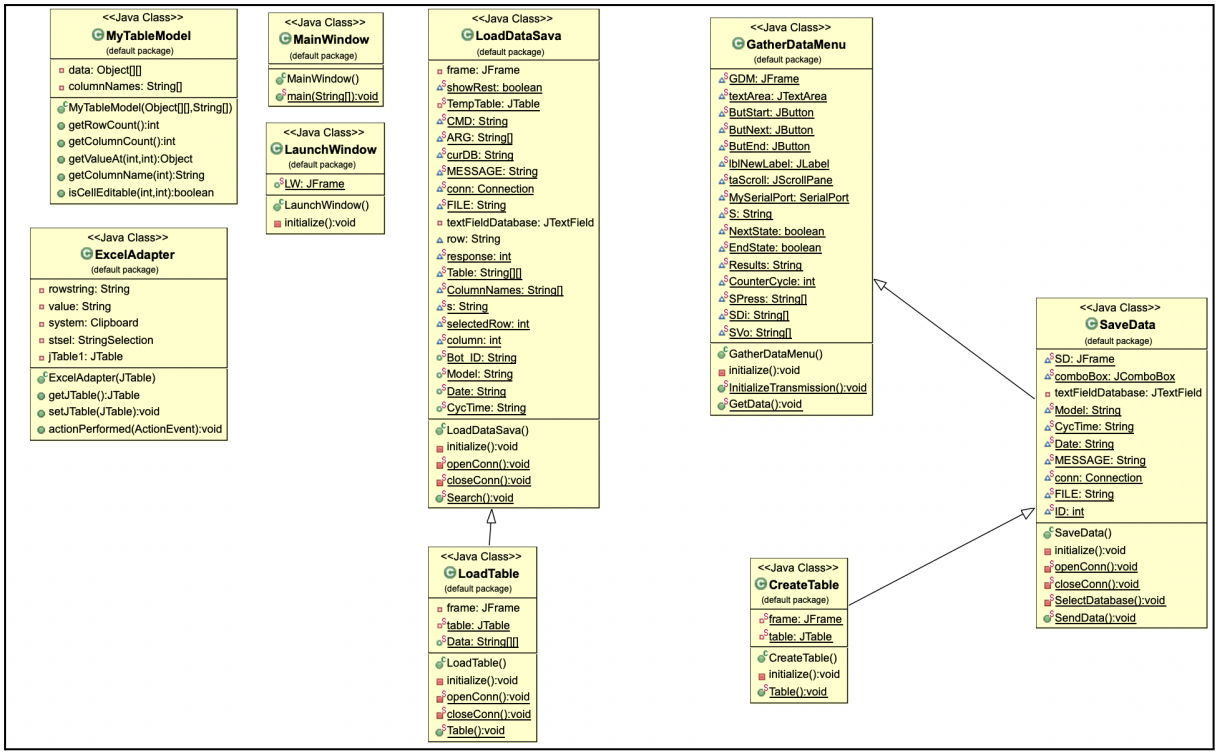


Figure 9 - UML diagram of Java project

I use inheritance to transfer static public variables in multiple classes (this is elaborated in later parts).

2.2. LaunchWindow.java

2.2.1 - Button to create a new class

```
JButton LoadTable = new JButton("Load Table"); //BUTTON TO LAUNCH LOAD TABLE MENU
LoadTable.setForeground(new Color(248, 0, 7));
LoadTable.addActionListener(new ActionListener() { //LoadTable Action Listener
    public void actionPerformed(ActionEvent e) {
        frame.dispose(); //Dispose the frame before creating a new instance
        LoadDataSava LD = new LoadDataSava(); //Create a new instance of the LoadDataSava class
    }
});
LoadTable.setBackground(new Color(99, 170, 255));
LoadTable.setBounds(63, 108, 130, 41);
frame.getContentPane().add(LoadTable); // Add to frame

JButton GatherData = new JButton("Gather Data"); //BUTTON TO LAUNCH LOAD TABLE MENU
GatherData.addActionListener(new ActionListener() { //GatherData Action Listener
    public void actionPerformed(ActionEvent e) {
        frame.dispose(); //Dispose the frame before creating a new instance
        GatherDataMenu GDM = new GatherDataMenu(); //Create a new instance of the GatherDataMenu class


    }
});
GatherData.setBounds(248, 108, 130, 41);
frame.getContentPane().add(GatherData); //Add to the frame
```

Figure 10 - Button to create class (Appendix 3 for full Java code)

I use the ActionListener to create new instances of classes.

2.3. GatherDataMenu.java

2.3.1 - Initialize a transmission with Serial Monitor

```
ButStart.setEnabled(false); //Once Start has been pushed, it cannot be pushed again
//Variables for the Reception
int BaudRate = 9600;
int DataBits = 8;
int StopBits = SerialPort.ONE_STOP_BIT;
int Parity   = SerialPort.NO_PARITY;

SerialPort [] AvailablePorts = SerialPort.getCommPorts(); //Initializes the SerialPort array object and storing the available ports on the computer

//Open the first Available port
MySerialPort = AvailablePorts[AvailablePorts.length - 2]; //Specific to my computer

// Set Serial port Parameters
MySerialPort.setComPortParameters(BaudRate,DataBits,StopBits,Parity);//Sets all serial port parameters at one time
MySerialPort.setComPortTimeouts(SerialPort.TIMEOUT_READ_BLOCKING, 1000, 0); //Set Read Time outs
MySerialPort.openPort(); //open the port

if (MySerialPort.isOpen()) { //If the Connection is open
    textArea.append("\n" + MySerialPort.getSystemPortName() + " is Open ");
}else {
    textArea.append("  ");
    JFrame f = new JFrame();
    JOptionPane.showMessageDialog(f, "ERROR: Arduino is not connected");
    frame.dispose();
    GatherDataMenu LWD = new GatherDataMenu(); //Creates an new instance of the class if
}

MySerialPort.flushIOBuffers();
System.out.println("Initialization complete");
```

Figure 11 - Serial communication

As I am not aware how to create a Serial connection from scratch, I am using the jSerialComm library and the specimen code that it provides to create a connection to the serial connection created with the Arduino[1]. Methods such as *.openPort()* are used in abstraction, as they are called from a client class in the jSerialComm library. I am not aware of the actual process of the methods.

[1] Fazecast, "jSerialComm," GitHub, accessed February 16, 2023, https://github.com/Fazecast/jSerialComm.

2.3.2 - Serial transmission with Java

```
//After Initilaization is finished, I start the exchange of data between Java and Arduino.
boolean ENDLOOP = true;
try {
    textArea.append(" starting serial transmission method\n");

    while (ENDLOOP == true){

        NextState = false;
        EndState = false;

        byte[] readBuffer = new byte[200]; //Creates a byte object because bytes are read from the Serial monitor and then converted to strings
        int numRead = MySerialPort.readBytes(readBuffer, readBuffer.length); //Counts the number of bytes in readBuffer
        S = new String(readBuffer, "UTF-8"); //convert bytes stored in readBuffer to String
        textArea.append(S + "\n"); // Prints 'S' to the terminal

        int a = S.indexOf(":"); //Gets the index of ':' in String 'S'

        if(a != -1){ //If ':' is in the String S (meaning that user input has been prompted)
            ButNext.setEnabled(true); //buttons are enabled
            ButEnd.setEnabled(true);

            while((NextState == false) && (EndState == false)) { //While no button has been pressed (this while loops freezes the program until user input

                ButNext.addActionListener(new ActionListener() { //ActionListener for Next button
                    public void actionPerformed(ActionEvent e) {
                        NextState = true;

                    }
                });

                ButEnd.addActionListener(new ActionListener() { //ActionListener for End button
                    public void actionPerformed(ActionEvent e) {
                        EndState = true;

                    }
                });
            }

            ButNext.setEnabled(false); //Buttons are disabled after they have been pushed
            ButEnd.setEnabled(false);

            //If there has been 50 cycles then the program ends.
            if(CounterCycle == 50) {
                EndState = true; //This statement makes the loop end
                NextState = false;
                textArea.append("   Limit of cycles has been reached\n");
            }

            //If End button has been pushed
            if(EndState == true){
                byte[] WriteByte = new byte[1]; //Byte to write to the Serial Monitor
                WriteByte[0] = 57; //send 9 to end the program (9 is the argument that will trigger the end of the program in Arduino)
                MySerialPort.writeBytes(WriteByte,1); //Method to write element

                TimeUnit.SECONDS.sleep(1); //Wait that the Serial Monitor prints the final data
                ENDLOOP = false; //Stop the loop
            }


            if(NextState == true){//SEND a byte so that Arduino thinks that enter was pressed, so that the Arduino program runs
                byte[] WriteByte = new byte[1];
                WriteByte[0] = 65; //write an element to the serial monitor, so that Arduino program continues running
                MySerialPort.writeBytes(WriteByte,1);

                a = -1; //Resets the condition for next if statement.
                CounterCycle++; //The varibale that stores the number of cycle increases
            }
        }
    }

} catch (Exception e){ //Catch for try
    e.printStackTrace();
}
```

Figure 12 - Nested loops for serial transmission

In Figure 12, I use nested while loops to access the data printed by Arduino to the Serial Monitor through Java[2]. The first while loop is used to repeat the transmission process for each cycle, and the nested while loop is used to prompt user input. *EndState*, *EndState*, and *ENDLOOP* are used as flags to break these loops.

[2] Fazecast, "jSerialComm," GitHub.

2.3.3 - String methods to store measurements in arrays

```
    //***Split 'Results' and store it in arrays

    //Get the indexes of characters in the String to split the strings into multiple subStrings
    int P1 = Results.indexOf("s");
    int P2 = Results.indexOf("f");
    int P3 = Results.indexOf("|");
    int P4 = Results.indexOf("?");

    //Create a substring from Results
    String TempPres = Results.substring(P1+1, P2-3);
    String TempDif = Results.substring(P2+1, P3-1);
    String TempVolume = Results.substring(P3+4, P4-1);

    //Store measurements in the arrays based on the substrings
    SPress = TempPres.split(" ");
    SDi = TempDif.split(" ");
    SVo = TempVolume.split(" ");

    textArea.append("Data was successfully imported to Java");

    frame.dispose();
    SaveData SD = new SaveData(); //Create a new object of SaveData that will querry the data to a database

}
```

Figure 13 - Store the measurements

I use *String* methods to divide 'Results' into arrays. Float values in the Arduino are stored as Strings in Java, as they can be sent as Strings in the database and they will be automatically converted to the 'Real' type in SQLite.

2.3.4 - Multithreading in Java

```
ButStart.addActionListener(new ActionListener() { //ActionListener for ButStart
    public void actionPerformed(ActionEvent e) {
        ButStart.setEnabled(false); //Turning off the ButStart, so that it cannot be pressed again
        new Thread(() -> {          //Creating a new thread
            InitializeTransmission();
            GetData();
        }).start();
    }
});
```

Figure 14 - Multi-Threading

Before creating another thread for these methods, my program would freeze. As I researched the problem, I found out that the methods froze because they were being executed on the event dispatch thread (EDT)[3]. Long-running tasks, such as the nested while loops in Figure 12, block the EDT from handling other events, causing the GUI to freeze. Hence, I execute these methods on a separate thread, so that they run in the background without blocking the EDT[4]. This is done by extending the Thread class with inheritance and starting a new Thread (*Figure 14)*.

[3]Special thread in the Java Swing framework that is responsible for handling all events and updates to the GUI components.
[4] "The Event Dispatch Thread," Oracle, https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html.

2.4 Save Data

2.4.1 Create connection with a database

```
static Connection conn = null; //Sqlite data type for connections
```

```
try {
    //Connect to the database
    //FILE refers to the variable that stores the name of the database inputted by user
    conn = DriverManager.getConnection("jdbc:sqlite:" + FILE); //Stores the connection created in 'conn'
                                                               //Connection is created in the same directory as the Java project
    MESSAGE = "DB \"" + FILE + "\" selected.";

} catch (SQLException EX) {
    MESSAGE = "exception: " + EX; //Catch an error
    JFrame f = new JFrame();
    JOptionPane.showMessageDialog(f, "ERROR: try again"); //Error message
    SaveData sd = new SaveData(); //Create a new instance

}
```

Figure 15 - Open connection

This method is called prior to interactions with the database. It creates a connection with a database.

2.4.2 Close database connection

```
try {
    if (conn != null) conn.close(); //If a conn has been created then close the conn

} catch (SQLException EX) { //If SQL catches an error
    if ((EX.getErrorCode() == 50000) && ("XJ015".equals(EX.getSQLState()))) {
        System.out.println("SQLite shut down properly");

    } else {
        System.err.println("SQLite did not shut down properly");
        System.err.println(EX.getMessage());
        closeConn(); //Call the method again to shut down the connection

    }

}
```

Figure 16 - Close SQLite connection

This method closes the connection. It should be called after all methods who open a database connection

2.4.3 Create tables in database (Statement)

```
Statement stmt; //Variable that is used to execute SQL statement
String strSQL = ""; //Variable to write the SQL statement in String

openConn(); //Open connection to current database

// Try to create the table "tMeasurements" IF it doesn't already exist
try{
    stmt = conn.createStatement();
    strSQL = "CREATE TABLE IF NOT EXISTS tMeasurements("
    + " Mes_ID integer PRIMARY KEY AUTOINCREMENT," //Create the primary key column
    + " Mes_Press Real not null, Mes_Diff Real not null, Mes_Disch Real not null, Mes_Vol Real not null," //Create needed tables with datatype
    + " MesBot_ID integer not null, MesCyc_ID integer not null," //Create tables that will be used for foreign keys
    + " FOREIGN KEY(MesBot_ID) REFERENCES tBottle(Bot_ID)," //Create a reference to foreign keys
    + " FOREIGN KEY(MesCyc_ID) REFERENCES tCycle(Cycle_ID));"; //Create a reference to foreign keys

    stmt.execute(strSQL); //execute the SQLite statement
    System.out.println("Table tMeasurements is ready"); //feeback

} catch (SQLException EX) {
    System.out.println(EX.getMessage());

} finally {
    // Close the connection to database
    closeConn();
}
```

Figure 17 - Create tables using SQLite (See Appendix B)

With the use of SQL language, I can create tables in the database. Figure 17 uses a SQLite statement, which is a string of SQL code that is used to perform various database operations.

In this case, it is used to create a new table by passing the statement to an SQLite database engine, which then interprets the statement and performs the requested operation on the database[5].

2.4.3 Insert in database (Prepared statement)

```
//Insert measurements into tMeasurements
String sqlMeas = "INSERT INTO tMeasurements VALUES(?,?,?,?,?,?,?)"; //Initialize a sql prepare statemnt

try{
     Statement stmt;
     ResultSet rs = null;
     String strSQL = "";

     strSQL = "SELECT Bot_ID" //This SQL script will retrieve the ID of the bottle that was just created.
            + "FROM tBottle"
            + "WHERE Bot_ID = (SELECT MAX(Bot_ID) FROM tBottle);";
     stmt = conn.createStatement();
     rs = stmt.executeQuery(strSQL); //RS with the executed query

     if (rs.next()) { //retrieve the ID from the resultset
         ID = rs.getInt("Bot_ID");
     }

     for(int c = 0; c < CounterCycle; c++){ //Loop to Insert measurements stored in arrays in the database

            PreparedStatement pstmt = conn.prepareStatement(sqlMeas); //An object that represents a precompiled SQL statement

            //The integer in the arguments of 'setString' refers to the index of the question mark in the prepared statement

            pstmt.setString(1, null);     //Data is sent as string and translated to REAL according to the datatype of the columns in the database
            pstmt.setString(2, SPress[c]); //Pressure
            pstmt.setString(3, SDi[c]);//Differential pressure
            pstmt.setString(4, SDi[c]);//Discharge rate
            pstmt.setString(5, SVo[c]);//Volume
            pstmt.setInt(6, ID); //BotID
            pstmt.setInt(7, c+1); //Cycle#

            pstmt.executeUpdate(); //Execute statement
            pstmt.close(); //close statement to recreate one for each iteration
    }


} catch (SQLException e){
        System.out.println("Error: " + e.getMessage());
}
```

Figure 18 - Insert measurements

In Figure 18, I create a prepared statement, a parameterized SQL statement, to insert the measurements stored in the arrays into the database without predefined values. By extending this class by 'GatherDataMenu' with inheritance, I can use the static variables such as, 'CounterCycle', 'SPress', 'SDi', and 'SVo', to initialise the prepared statement.

2.4 Create Table

2.4.3 2D array for JTable

```
public static void Table() {

    String [] ColumnNames = {"Cyc_ID", "Pressure", "Differential", "Discharge rate", "Volume"}; //String array for the names of the columns
    String Data [][] = new String[CounterCycle][5]; //2D array that stores the Data used to create the JTable
                                                    //CounterCycle is the amount of cycles, hence measurements, for a specific bottle
                                                    //initialize the array with 5 column because there are 5 different variable that are displayed.

    for(int c = 0; c < CounterCycle; c++) { //Loop to put data in 2D array

        Data[c][0] = Integer.toString(c + 1);
        Data[c][1] = SPress[c];
        Data[c][2] = SDi[c];
        Data[c][3] = SDi[c];
        Data[c][4] = SVo[c];
    }

    table = new JTable(new MyTableModel(Data, ColumnNames)); //Initialize JTable with a table model that makes cell not editable but selectable
    table.setCellSelectionEnabled(true);
    table.setBackground(Color.pink);
    ExcelAdapter myAd = new ExcelAdapter(table); //Class that enables copy and paste for the table
    table.setBounds(70, 50, 422, 226);

}
```

Figure 19 - Create Table

[5] "SQL As Understood By SQLite," SQLite, accessed February 16, 2023, https://www.sqlite.org/lang.html.

In this method, I initialise the table with the measurements that have been recorded. I am able to use the arrays of measurements created in the previous class because this class inherits all of the variables. The 2D array is initialised using a for loop. Then, I use a table model to create a table based on a table model that does not allow cells to be edited. Finally, the table is attached to an ExcelAdapter which allows the measurements in the table to be copied to an spreadsheet format. This object is created in abstraction with a client class[6].

2.5 Load Data Menu

In this class, the user opens a database, searches the bottle he needs to extract data from, through the use of a table and search parameters in a combo box, and creates a table with the measurements of the bottle.

2.5.1 Select Database using JFileChooser

```
btnOpenDatabase.addActionListener(new ActionListener() { //Action listener for database button
    public void actionPerformed(ActionEvent e) {
        if(e.getSource()==btnOpenDatabase) {

            //Using a JFile chooser to choose the DATABASE
            JFileChooser FC = new JFileChooser(); //Open file chooser
            FC.setCurrentDirectory(new File("."));
            response = FC.showOpenDialog(null); //variable to check the responde of the file chooser

            if(response == JFileChooser.APPROVE_OPTION) { //If the approve option has been selected
                FILE = FC.getSelectedFile().getName();  //Store the name of the file selected

                JFrame f = new JFrame();
                JOptionPane.showMessageDialog(f, "Database Selected: " + FILE); //Option Pane to say that Database was selected.

                frame.getContentPane().add(comboSearch);  //When Database has been selected, other elements are placed on the GUI
                frame.getContentPane().add(lblNewLabel_1);
                frame.getContentPane().add(btnCreate);

            }
            btnOpenDatabase.setText(FILE); //The button now displaying the name of database selected.
        }
    }
});
```

Figure 20 - Select Database/file choose

With this method, I use the JFileChooser object to select the database from which the user needs to load data from.

[6] Info World, last modified September 20, 1999, accessed February 16, 2023, https://www.infoworld.com/article/2077579/java-tip-77--enable-copy-and-paste-functionality-between-swing-s-jtables-and-excel.html.

2.5.2 Search parameters in tables

```
switch(s) { //Switch statement that selects the SQL script depending on the search paramenter of the user.
//s is the option chosen from the comboBox
   case("Date"):
        // Select all from the bottle with the Date as the reference column
        strSQL = "SELECT Bot_Date, Bot_ID, Bot_Model, Bot_CycTime "
               + " FROM tBottle"
               + " ORDER BY Bot_Date ASC;";

    case("Bot_ID"):
       // Select all from the bottle with the Bot_ID as the reference column
        strSQL = "SELECT Bot_ID, Bot_Date, Bot_Model, Bot_CycTime "
                  + " FROM tBottle"
                  + " ORDER BY Bot_ID ASC;";

    case("Bot_Model"):
       // Select all from the bottle with the Bot_Model as the reference column
        strSQL = "SELECT  Bot_Model, Bot_ID, Bot_Date, Bot_CycTime "
                  + " FROM tBottle"
                  + " ORDER BY Bot_Model ASC;";

    case("Bot_CycTime"):
       // Select all from the bottle with the Bot_CycTime as the reference column
        strSQL = "SELECT Bot_CycTime, Bot_ID, Bot_Date, Bot_Model "
                  + " FROM tBottle"
                  + " ORDER BY Bot_CycTime ASC;";
}
```

Figure 21 - Switch statement

Instead of using multiple ‘if’ statements, I use a switch statement to create a SQL script based on the search parameter chosen by the user. This allows for more structure and readability in the code.

```
stmt = conn.createStatement(); //Allign the statement with the SQLite connection
rs = stmt.executeQuery(strSQL); //Execute the query

//Create a new query to store the number of rows in tBottle. This number will be used to initialize the number of rows in the table (2D array).
String strSQL1 = "SELECT COUNT(*) FROM tBottle";
stmt1 = conn.createStatement();
rowNum = stmt1.executeQuery(strSQL1);

//Initialize 2D array for the table that will be used to search the needed bottle
Table = new String[(((Number) rowNum.getObject(1)).intValue())][4]; //Intialize array with correct numebr

ResultSetMetaData rsmd = rs.getMetaData(); //Using a resultSetMetaData object that providesinformation about the columns in a ResultSet.

for(int c = 0; c < rsmd.getColumnCount(); c++) { //Looping throuhg rsmd to store the column names in the right order.
     ColumnNames[c] = rsmd.getColumnName(c+1);   //Store in ColumnNames to create table.
}
```

Figure 22 - Initialize 2D array

I use a new SQL script to store the number of rows for the 2D array. Next, I created a ResultSetMetaData to store the names of the columns in the array. The 2D array needed to be initialised with the right amount indexes as a table could not be created with null values.

2.6 LoadTable

This class creates a table with the measurements of the bottle chosen in ‘Load Data Menu’.

2.6.1 Extract data and loop through ResultSet

```
// Select all from the bottle with the Date in front
strSQL = "SELECT MesCyc_ID, Mes_Press, Mes_Diff, Mes_Disch, Mes_Vol "
       + " FROM tMeasurements "
       + " WHERE MesBot_ID = " + Bot_ID //Using the Bot_ID that was stored from the user parameter.
       + " ORDER BY MesBot_ID ASC;";

stmt = conn.createStatement();
rs = stmt.executeQuery(strSQL); //ResultSet that stores executed script
```

Figure 23 - Resultset with inherited variable

The variable ‘Bot_ID’ has been inherited from the parent class Load Data Menu. This queries the data from the requested bottle.

```
int c = 0; //Loop through the ResultSet in order to store the measurements in the 2D array to create the table
while (rs.next()) { //While the resultSet has more items.

     Data[c][0] = rs.getString(ColumnNames[0]);
     Data[c][1] = rs.getString(ColumnNames[1]);
     Data[c][2] = rs.getString(ColumnNames[2]);
     Data[c][3] = rs.getString(ColumnNames[3]);
     Data[c][4] = rs.getString(ColumnNames[4]);

     c++;
}
```

Figure 24 - Loop through result set

The loop through the resultset stores the data from each column at ‘c’ row until there are no more elements in resultnext.

3. Bibliography

"The Event Dispatch Thread." Oracle.
https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html.

Fazecast. "jSerialComm." GitHub. Accessed February 16, 2023.
https://github.com/Fazecast/jSerialComm.

Info World. Last modified September 20, 1999. Accessed February 16, 2023.
https://www.infoworld.com/article/2077579/java-tip-77--enable-copy-and-paste-functionality-between-swing-s-jtables-and-excel.html.

"SQL As Understood By SQLite." SQLite. Accessed February 16, 2023.
https://www.sqlite.org/lang.html.

"The Sqlite JDBC Driver." DbSchema. Accessed February 16, 2023.
https://dbschema.com/jdbc-driver/Sqlite.html.