Development

Techniques & Complexities Used

- Encapsulation
- Abstraction
- Inheritance
- Array Lists
- Sub-procedures
- Searching
- Recursion
- Exceptions
- Serialisation
- Database
- Usage of external libraries

Encapsulation

As I learned from IB CS Textbook, encapsulation improves data security and simplifies data management. The main application consists of three packages: Config, Database, and View (See Figure 1). All the classes required for the GUI are included in the view package, while the database package's database classes are referenced in the view package. Classes that keep the MySQL database connection and store all the necessary methods for each class in the views package are located in the database package. The Main class, executed when the program opens, is also in the config package, serving as a repository for the classes typically needed in the view and database packages.

Additionally, the Mail class—required for confirmation procedures—and the most often used global variables are stored in the config package. It wouldn't be easy to modify current account user credentials without such an organisation.

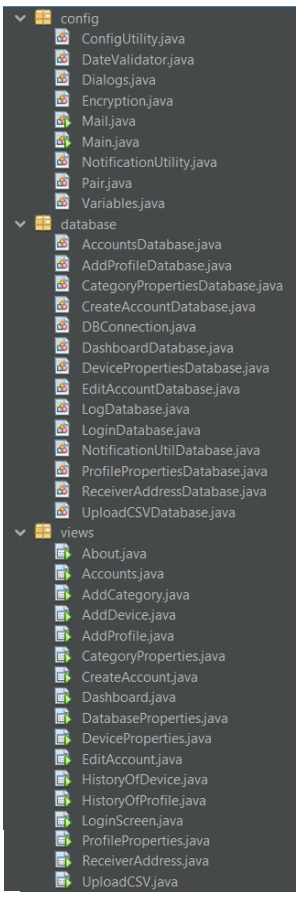

Figure 1: Package Structure

Abstraction

As an OOP method, the application provides an abstraction to hide the detailed implementations inside the program, enhancing intelligibility and simplicity, that I learned from websites[1].

```java
public class Variables {
    /* Global variables that is used throughout the program. */
    public static String id = null; // ID of the current user
    public static String uname = null; // Username of the current user
    public static String pword = null; // Password of the current user
    public static String perm = null; // Permission of the current user
    public static String dateofcreation = null; // Creation date of the current user
    public static String mail = "eystest@outlook.com"; // E-mail address for the verification processes
}
```

Figure 2: Global Variables Used Throughout The Program As Public Variables

```java
/*Sub-procedure that returns all the categories created and stored in categories database as an array list of String arrays */
public List<String[]> getCategoriesFull() {
    List<String[]> values = new ArrayList<>();
    try {
        st = cn.createStatement();
        /* A private ArrayList variable that cannnot be accessed out of this method. */
        List<String> categories = new ArrayList<>();
        rs = st.executeQuery("SELECT * FROM application.inventory");

        /* The following process adds each item's category stored in inventory database into the
        dynamic array list to calculate the total registered device of a category in the next process */
        while(rs.next()){
            categories.add(rs.getString("category"));
        }

        /* query variable used several times in this class, but they do not mix in favour of being private variables */
        String query="SELECT * FROM application.categories";
        rs = st.executeQuery(query);

        /* The following process adds each cateogry's information stored in categories database
        and the frequency of registered devices of a category into a dynamic array list */
        while (rs.next()) {
            /* The following variables are initialised as private, preventing any access from outside of this method */
            String ID = rs.getString("categoryID");
            String name = rs.getString("name");
            String prprtynumber = rs.getString("propertyNumber");
            String regdevicenum = String.valueOf(Collections.frequency(categories, name));
            values.add(new String[] {ID,name,prprtynumber,regdevicenum});

        }
    } catch (SQLException e) {
        System.out.println("Verileri okuma sırasında bir hata oluştu:" + e);
    }
    return values;
}
```

Figure 3: Private Variables Used while Fetching All Categories, Limiting the Access and Preventing Any Confusion Between the Variables

Inheritance

Inheritance is used to pass the Object class attributes into the Pair class. As I learned the concepts of object-oriented programming from CS lessons and websites[2], I implemented inheritance to my program as Pair is required to carry multiple data types together as key and value. Pair is generally used to return two data types from a method at once. The pair class (Figure 4) and the example used are shown (Figure 5).

---

[1] https://www.w3schools.com/java/java_oop.asp

[2] Biniasz, Kyle. "What Are OOP Concepts in Java? 4 Primary Concepts." *Stackify*. 24 Nov. 2021. Web. 10 May 2022.

```java
/**
 * @param <K>
 * @param <V>
 *
 * A class to represent key-value pairs including any Object. Implements Serializable class.
 */
public class Pair<K extends Object, V extends Object> implements Serializable {

    @Retention(value = RetentionPolicy.RUNTIME) // Retention annotation for value
    @Target(value = {ElementType.PARAMETER}) // Target annotation for value

    public @interface NamedArg {

        public String value();

        public String defaultValue() default "";
    }

    private K key;
    private V value;

    // Returning the key of stored Pair
    public K getKey() {
        return key;
    }

    // Returning the value of stored Pair
    public V getValue() {
        return value;
    }

    // Constructor for the class, storing the Pair of different Objects.
    public Pair(@NamedArg(value = "key") K arg0, @NamedArg(value = "value") V arg1) {
        key = arg0;
        value = arg1;
    }
}
```

Figure 4: Pair class created to carry different data types at once, which uses Inheritance and Abstraction concepts of OOP

```java
/** This method shows an option dialog with a customisable combo box and properties passed through parameters
 * and returns the index & the value of the item selected by user.
 * @param choices This is the array of the values that will be shown in the combo box.
 * @param titleBar This is the title that will be showed in the title bar.
 * @param frame This is the frame that the dialog will be shown.
 * @return the index and the value of selected item in combo box as a Pair.
 */
public static Pair<Integer, Object> comboBox(Component frame, String[] choices, String titleBar)
{
    Object[] options = {"Seç"}; // This is the ok button name.
    JComboBox<String> combo = new JComboBox<>(choices); // The values are used as the elements of a new combo box created
    int choice = JOptionPane.showOptionDialog(frame, combo, titleBar, 0, JOptionPane.QUESTION_MESSAGE, null, options, options[0]);
    return new Pair<>(choice, combo.getSelectedItem());
}
```

Figure 5: Example use of Pair in Dialogs class, used to pass the Integer result and Onject selected item from the diologBox

Array Lists

Array lists, that I learned from a website[3], carry multiple data when dynamic data, such as inventory items, is pulled from the database.

In the class "DashboardDatabase," the use of array lists is crucial since the ResultSet returned from MySQL consists of a nested structure and requires dynamic structure to access all of the data. Most of the sub-procedures in the database package consist of Array Lists, and such use can be seen in Figure 6 that the returned ResultSet is going through a while loop until there are no elements left. Each element is added to the ArrayList as a String array. Figure 7 is another example, meeting the success criterion 8. Using array lists was vital as it provided flexibility in more complex variables and ease of use.

```java
/*Sub-procedure that returns all the profiles created and stored in database as an array list of String arrays*/
public List<String[]> getProfilesFull() {
    List<String[]> values = new ArrayList<>();
    try {
        st = cn.createStatement();

        List<String> c = new ArrayList<>();
        rs = st.executeQuery("SELECT * FROM application.inventory");

        /*The following process adds each item's assignedProfileID stored in inventory database into the
        dynamic array list to calculate the total assigned devices of a profile in the next process*/
        while(rs.next()){
            c.add(rs.getString("assignedProfileID"));
        }

        String query="SELECT * FROM application.profiles";
        rs = st.executeQuery(query);

        /*The following process adds each profile's information stored in profiles database
        and the frequency of assigned devices of a profile into a dynamic array list */
        while (rs.next()) {
            String ID = rs.getString("profileID");
            String name = rs.getString("name") + " " + rs.getString("surname");
            String status = rs.getString("status");
            String devicenumber = String.valueOf(Collections.frequency(c, ID));
            values.add(new String[] {ID,name,devicenumber,status});

        }
    } catch (SQLException e) {
        System.out.println("Verileri okuma sırasında bir hata oluştu:" + e);
    }
    return values;
}
```

Figure 6: The Example Use of Array Lists in methods - getProfilesFull

```java
/*Another sub-procedure that returns all the items stored in database as an array list of String arrays*/
public List<String[]> getInventoryFull() {
    List<String[]> values = new ArrayList<>();
    try {
        st = cn.createStatement();
        String query="SELECT * FROM application.inventory";

        /*Getting the result of the query search in database and it returns each item found in database.*/
        rs = st.executeQuery(query);

        /*The following process adds each item stored in database into the dynamic array list initialized before*/
        while (rs.next()) {
            String ID = rs.getString("deviceID");
            String category = rs.getString("category");
            String device = rs.getString("deviceName");
            String status = rs.getString("status");
            String assignedProfile = rs.getString("assignedProfile");
            String warranty = rs.getDate("warranty").toString();
            Period duration = Period.between(LocalDate.now(), rs.getDate("warranty").toLocalDate());
            String remainingWarranty = String.valueOf(duration.getDays()
                    + duration.getMonths() * 30 + duration.getYears() * 12 * 30);
            values.add(new String[]{ID, category, device, status, assignedProfile, warranty, remainingWarranty});

        }
    } catch (SQLException e) {
        System.out.println("Verileri okuma sırasında bir hata oluştu:" + e);
    }
    return values;
}
```

Figure 7: A Method from DashboardDatabase containing Array List - getInventoryFull

[3] https://www.w3schools.com/java/java_arraylist.asp

Sub-procedures

While developing this program, the concept of thinking procedurally is mainly used. Learning thinking procedurally at CS Higher-level allowed modifying multiple functions concurrently. It takes a massive role in the program as various tasks need to be handled and work independently from others. Thus, simple management is provided. Additionally, using sub-procedures removes the unnecessary procedures to repeat with its reusability.

When a table is shown or an update/search button is pressed, the table associated with the action needs to be updated, meeting the success criteria 8, 10, and 15. Since the program includes many tables, it also implies different sub-procedures in various locations to update a table (See Figure 8).

```java
private void assignButtonActionPerformed(java.awt.event.ActionEvent evt) {
    if(chooseDevicetoAssign.getSelectedItem() == null) return; // Return if the user didn't select any device to assign
    if(assignedProfileID==null||assignedProfileID.equals("")) return; // Return if the profile ID is null or ""

    /* Initialise deviceidname as the string value of selected item to assign, then seperate the ID and the name
    in deviceidname as a String array and assign the ID to deviceID*/
    String deviceidname = chooseDevicetoAssign.getSelectedItem().toString();
    String[] deviceIDName = deviceidname.split(" - ");
    String deviceID = deviceIDName[0];

    /* Call the sub-procedure that assigns the device to the profile from the parameters passed. */
    ppd.assign(deviceID, deviceidname, assignedProfileID, assignedProfileName);

    /* Initialise the list value returned from the sub-procedure that returns the assigned devices
    associated with profileID passed as a parameter to fill the table in GUI*/
    assignedDevices = ppd.getAssignedDevices(assignedProfileID);
    DefaultTableModel model=(DefaultTableModel) assignedDevicesTable.getModel();
    model.setRowCount(0);
    for (String[] eachRow : assignedDevices) {
        model.addRow(eachRow); // Adding each element in the assignedDevices containing the returned list from the sub-procedure
    }

    /* Initialise a second list value returned from the sub-procedure that returns the assigned devices
    associated with profileID passed as a parameter in order to fill the combobox in GUI*/
    assignedDevices = ppd.getAssignedDevices(assignedProfileID);
    DefaultComboBoxModel model1 = (DefaultComboBoxModel) chooseDevicetoRemoveAssignment.getModel();
    model1.removeAllElements();
    for (String[] eachRow : assignedDevices) {
        String text = eachRow[0] + " - " + eachRow[1];
        model1.addElement(text); // Adding each element in the assignedDevices containing the returned list from the sub-procedure
    }

    /* Initialise a third list value returned from the sub-procedure that returns the all available
    devices to fill the other combobox in GUI*/
    List<String[]> values = ppd.getAllAvailableDevices();
    DefaultComboBoxModel device = (DefaultComboBoxModel) chooseDevicetoAssign.getModel();
    device.removeAllElements();
    for (String[] eachRow : values) {
        String text = eachRow[0] + " - " + eachRow[1];
        if(device.getIndexOf(text) == -1){
            device.addElement(text); // Adding each element in the values containing the returned list from the sub-procedure
        }
    }

    DefaultTableModel model2 = (DefaultTableModel) assignedDevicesTable.getModel();
    if(model2.getRowCount() != 0)
        removeAllAssignments.setEnabled(true);
}
```

Figure 8: Use of Sub-procedures in Profiles Properties class

Also, the program involves a method that users export a CSV File from inventory, categories, or profile tables. If the program couldn't include sub-procedures, it had to imply the same methods multiple times, occupying space and damaging the simplicity. Initial part of the task is shown in Figure 9.

```java
private void exportButtonActionPerformed(java.awt.event.ActionEvent evt) {

    /* The following lines of codes constructs the JFileChooser element on the filechooser object and set
    the default file name with extension, then the numerical value of approve and cancel selections are initialised.*/
    JFileChooser filechooser = new JFileChooser();
    filechooser.setFileSelectionMode(0);
    filechooser.setMultiSelectionEnabled(false);
    filechooser.setSelectedFile(new File("untitled.csv"));
    filechooser.addChoosableFileFilter(new FileNameExtensionFilter("CSV (Virgülle ayrılmış) (*.csv)", "csv"));
    filechooser.setAcceptAllFileFilterUsed(true);
    int cancel = JFileChooser.CANCEL_OPTION;
    int approve = JFileChooser.APPROVE_OPTION;
    switch (currentTab) { /* Switching between the tab names to call the sub-procedure to export table data
                            with the corresponding table object of current tab. */
        case "inventory":
            {
                exportTable(inventoryTable, filechooser, approve, cancel, currentTab);
                break;
            }
        case "profiles":
            {
                exportTable(profilesTable, filechooser, approve, cancel, currentTab);
                break;
            }
        case "categories":
            {
                exportTable(categoriesTable, filechooser, approve, cancel, currentTab);
                break;
            }
        default:
            break;
    }
}
```

Figure 9: Another use of sub-procedures in Dashboard while exporting CSV File

As seen in Figure 9, this method calls a sub-procedure multiple times. The sub-procedure "exportTable" contains many processes (see Figure 10). Otherwise, the task had to be repeated numerous times, causing an unnecessary flock in the code.

```java
/**
 * This method exports the specified table's information as a CSV File, and it provides
 * a file chooser window for user to select the target location for the file will be created.
 * @param table This is the table that the information will be gathered.
 * @param filechooser This is the JFileChooser constructor to show file chooser window.
 * @param approve This parameter contains the value of approve choice from the file chooser.
 * @param cancel This parameter contains the value of cancel choice from the file chooser.
 * @param currentTab This parameter contains the name of current tab open which will be
 * used to decide the type of data will be exported.
 */
private void exportTable(JTable table, JFileChooser filechooser, int approve, int cancel, String currentTab){
    /* The following initialisation creates a int array for the selected rows in the table */
    int[] rows = table.getSelectedRows();
    int answer = filechooser.showSaveDialog(null);
    if (answer == cancel) { //User commanded to cancel the process if answer equals to cancel variable
        return;
    }

    /* Passing the location of target folder and name into path variable, then getting the extension written */
    String path = filechooser.getSelectedFile().getAbsolutePath();
    String[] extensions = ((FileNameExtensionFilter) filechooser.getFileFilter()).getExtensions();
    String extension = "."+extensions[0];

    /* Looping until the user enters an extension and clicks the approve button,
    Showing a warning about the extension error to user if the condition is not met.*/
    while (answer == approve && !path.endsWith(extension)) { //
        if (answer == cancel) {
            return;
        }
        JOptionPane.showMessageDialog(null, "Lütfen "+extension+" uzantısı ekleyiniz!", "Hata", 0);
        answer = filechooser.showSaveDialog(filechooser);
        path = filechooser.getSelectedFile().getAbsolutePath();
    }
    /* Looping until the user enters an existing file name and clicks the approve button*/
    while (answer == approve && filechooser.getSelectedFile().exists()) {
        if (answer == cancel) {
            return;
        }
        // Showing a choice dialog if the user wants to overwrite the file if the condition is not met.
        int n = JOptionPane.showOptionDialog(this, "Bu isime sahip bir dosya zaten bulunuyor. Bu dosyanın üzerine yazmak"
                + " istediğinize emin misiniz?.", "Uyarı", 0, 0, null, new String[]{"Evet", "Hayır"}, "Hayır");
        if (n == 1) { // Showing the filechooser window again, if the answer is not 0 `yes`.
            answer = filechooser.showSaveDialog(filechooser);
        } else if (n == 0) {
                        break;
        }
    }
        if (rows.length < 1) {
            if (answer == approve) { /* Switching between the tab names and exporting all the elements in the table of the
                                        current tab, if user selects approve and the selected row count is 0 */
                switch (currentTab) {
                    case "inventory":
                        sdb.exportInventory(path); // Exporting the all data from inventory table
                        break;
                    case "profiles":
                        sdb.exportProfiles(path); // Exporting the all data from profiles table
                        break;
                    case "categories":
                        sdb.exportCategories(path); // Exporting the all data from categories table
                        break;
                    default:
                        break;
                }
            }
        } else {
            /* Finding the index of the column `ID` and then storing all the values in that column into a List of String values,
            if the selected row count is 1 or more*/
            int column = 0;
            for (int columnIndex = 0; columnIndex < table.getColumnCount(); columnIndex++) {
                if (table.getColumnName(columnIndex).equals("ID")) {
                    column = columnIndex;
                }
            }
            List<String> IDs = new ArrayList<>();
            for (int row : rows) {
                String ID = (String) table.getValueAt(row, column);
                IDs.add(ID);
            }
            if (answer == approve) { /* Switching between the tab names and exporting all the elements
                                        in the table of the current tab, if user selects approve */
                switch (currentTab) {
                    case "inventory":
                        sdb.exportDevices(path, IDs); // Exporting the selected rows' data from inventory table
                        break;
                    case "profiles":
                        sdb.exportSomeProfiles(path, IDs); // Exporting the selected rows' data from profiles table
                        break;
                    case "categories":
                        sdb.exportSomeCategories(path, IDs); // Exporting the selected rows' data from categories table
                        break;
                    default:
                        break;
                }
            }
        }
}
```

Figure 10: A method that contains various processes for exporting CSV File

Searching

Searching is implemented throughout the program for the users to look for specific item(s). The database queries expand the search field with filters and parameters.

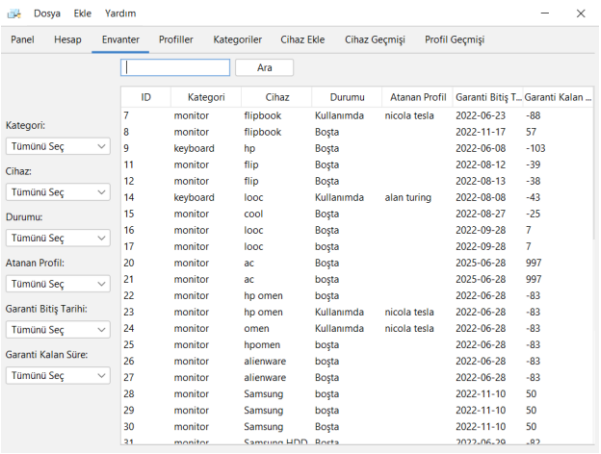

Figure 11: Inventory Search Screen

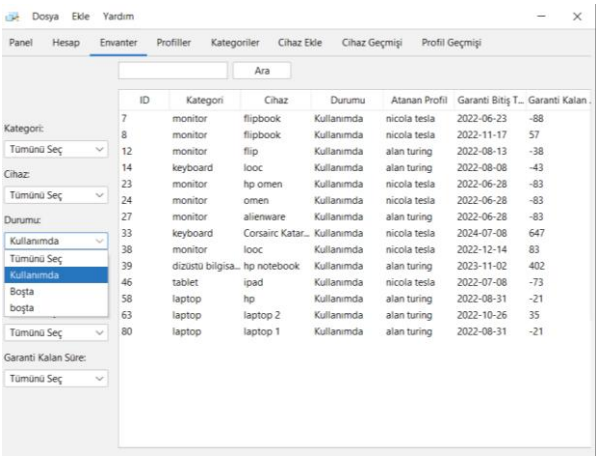

Figure 12: Filtering the devices as "In Use"/"Kullanımda"

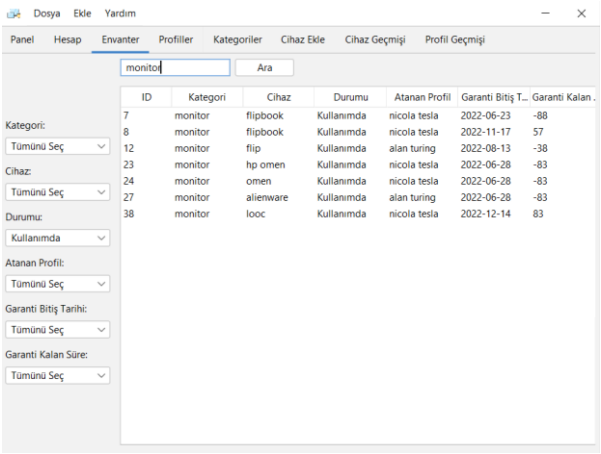

Figure 13: Searching the devices that has the term 'monitor' in their properties

```java
public List<String[]> getInventory(String searchTerm, String categoryFilter, String deviceFilter, String statusFilter,
        String assignedProfileFilter,String warrantyFilter, String remainingWarrantyFilter) {
    List<String[]> values = new ArrayList<>();
    try { // Exception Handling
        String searchterm = searchTerm != null ? "(LOCATE('"+searchTerm+"',deviceID) > 0 OR LOCATE('"+searchTerm+"',category) > 0 OR "
                + "LOCATE('"+searchTerm+"',deviceName) > 0 OR LOCATE('"+searchTerm+"',status) > 0 OR LOCATE('"+searchTerm+"',assignedProfile) > 0)" : "";
        String ctagory = (categoryFilter!= null) && (!categoryFilter.equals("Tümünü Seç")) ? " AND category='"+categoryFilter+"'" : "";
        String dvice = (deviceFilter!= null) && (!deviceFilter.equals("Tümünü Seç")) ? " AND deviceName='"+deviceFilter+"'" : "";
        String stats = (statusFilter!= null) && (!statusFilter.equals("Tümünü Seç"))? " AND status='"+statusFilter+"'" : "";
        String assignedPrfile = (assignedProfileFilter!= null) && (!assignedProfileFilter.equals("Tümünü Seç"))? " AND assignedProfile='"+assignedProfileFilter+"'" : "";
        String wrranty = (warrantyFilter!= null) && (!warrantyFilter.equals("Tümünü Seç"))? " AND warranty='"+warrantyFilter+" '" : "";
        String warrantyfromremaining = (remainingWarrantyFilter!=null) &&
                (!remainingWarrantyFilter.equals("Tümünü Seç")) ? String.valueOf(LocalDate.now().plusDays(Integer.valueOf(remainingWarrantyFilter))): "";
        String warrantyfromrmainingwarranty = (remainingWarrantyFilter!= null) &&
                (!wrranty.equals("")) && (!remainingWarrantyFilter.equals("Tümünü Seç"))? " AND warranty='"+warrantyfromremaining+"'" : "";
        /* Initialising the neccessary variables for searching and filtering the devices. */
        st = cn.createStatement(); // Using the connection created with the MYSQL databse to fetch the data according to the filters and searching.
        String query="SELECT * FROM application.inventory WHERE"+ searchterm + ctagory + dvice + stats + assignedPrfile + wrranty + warrantyfromrmainingwarranty;
        rs = st.executeQuery(query);
            // Passing each returned data to an ArrayList
        while (rs.next()) {
            String ID = rs.getString("deviceID");
            String category = rs.getString("category");
            String device = rs.getString("deviceName");
            String status = rs.getString("status");
            String assignedProfile = rs.getString("assignedProfile");
            String warranty = rs.getDate("warranty").toString();
            Period duration = Period.between(LocalDate.now(), rs.getDate("warranty").toLocalDate());
            String remainingWarranty = String.valueOf(duration.getDays() + duration.getMonths()*30 + duration.getYears()*12*30);
            values.add(new String[] {ID,category,device,status,assignedProfile,warranty, remainingWarranty});

        }
    } catch (SQLException e) {
        System.out.println("Verileri okuma sırasında bir hata oluştu:" + e);
    }
    return values;
}
```

Figure 14: Method for searching and filtering the inventory

Recursion

Since it provides a conditional iterative field, recursion is used to improve the program's usability and enhance exception handling, which I learned from CS HL Textbook[4].

```java
/* The following initialisation creates a int array for the selected rows in the table */
int[] rows = table.getSelectedRows();
int answer = filechooser.showSaveDialog(null);
if (answer == cancel) { //User commanded to cancel the process if answer equals to cancel variable
    return;
}

/* Passing the location of target folder and name into path variable, then getting the extension written */
String path = filechooser.getSelectedFile().getAbsolutePath();
String[] extensions = ((FileNameExtensionFilter) filechooser.getFileFilter()).getExtensions();
String extension = "."+extensions[0];

/* Looping until the user enters an extension and clicks the approve button,
Showing a warning about the extension error to user if the condition is not met.*/
while (answer == approve && !path.endsWith(extension)) { //
    if (answer == cancel) {
        return;
    }
    JOptionPane.showMessageDialog(null, "Lütfen "+extension+" uzantısı ekleyiniz!", "Hata", 0);
    answer = filechooser.showSaveDialog(filechooser);
    path = filechooser.getSelectedFile().getAbsolutePath();
}
/* Looping until the user enters an existing file name and clicks the approve button*/
while (answer == approve && filechooser.getSelectedFile().exists()) {
    if (answer == cancel) {
        return;
    }
    // Showing a choice dialog if the user wants to overwrite the file if the condition is not met.
    int n = JOptionPane.showOptionDialog(this, "Bu isime sahip bir dosya zaten bulunuyor. Bu dosyanın üzerine yazmak"
            + " istediğinize emin misiniz?.", "Uyarı", 0, 0, null, new String[]{"Evet", "Hayır"}, "Hayır");
    if (n == 1) { // Showing the filechooser window again, if the answer is not 0 `yes`.
        answer = filechooser.showSaveDialog(filechooser);
    } else if (n == 0) {
                break;
    }
}
```

Figure 15: Showing a file chooser in recursion with a condition

Exceptions

I learned exceptions from Oracle's Java website[5]. The development requires considering exceptions, a sub-title of thinking ahead. The developer makes exceptions, which are essential as many problems can occur while the program runs. Without exceptions, the program would give errors, and the user could face problems, decreasing the ease of use and user satisfaction. The program frequently uses exceptions since various conditioning and loops exist. Multiple dialogues are added between the steps of exporting a table because such cases to be prevented can be handled while exporting (See Figures 9 and 10).

[4] Advanced Computer Science: For the IB Diploma Program (international Baccalaureate) High Level Computer Science

[5] https://docs.oracle.com/javase/tutorial/essential/exceptions/

It is essential to inform the user about these exceptions; thus, a class for specialised dialogues is created to use each in necessary instances (See Figure 16).

```java
public class Dialogs {

    /** This method shows a message dialog with the properties passed through parameters.
     * @param infoMessage This is the message that will be showed in the dialog.
     * @param titleBar This is the title that will be showed in the title bar.
     */
    public static void infoBox(String infoMessage, String titleBar)
    {
        JOptionPane.showMessageDialog(null, infoMessage, titleBar, JOptionPane.INFORMATION_MESSAGE);
    }

    /** This method shows an error dialog with the properties passed through parameters.
     * @param infoMessage This is the error message that will be showed in the dialog.
     * @param titleBar This is the title that will be showed in the title bar.
     */
    public static void errorBox(String infoMessage, String titleBar)
    {
        JOptionPane.showMessageDialog(null, infoMessage, titleBar, JOptionPane.ERROR_MESSAGE);
    }

    /** This method returns the answer of the user to the custom option dialog with the properties passed through parameters.
     * The option dialog is shown in the frame passed through parameters.
     * @param frame This is the frame that the dialog will be shown.
     * @param infoMessage This is the message that will be showed in the dialog.
     * @param titleBar This is the title that will be showed in the title bar.
     * @return 0 or 1 accordingly to the user's choice in the option dialog.
     */
    public static int questionBox(Component frame, String infoMessage, String titleBar)
    {
        Object[] options = {"Evet", "Hayır"}; // These are the button names will be shown to user
        int choice = JOptionPane.showOptionDialog(frame, infoMessage, titleBar, 0, JOptionPane.QUESTION_MESSAGE, null, options, options[1]);
        return choice;
    }

    /** This method shows an option dialog with a customisable combo box and properties passed through parameters
     * and returns the index & the value of the item selected by user.
     * @param choices This is the array of the values that will be shown in the combo box.
     * @param titleBar This is the title that will be showed in the title bar.
     * @param frame This is the frame that the dialog will be shown.
     * @return the index and the value of selected item in combo box as a Pair.
     */
    public static Pair<Integer, Object> comboBox(Component frame, String[] choices, String titleBar)
    {
        Object[] options = {"Seç"}; // This is the ok button name.
        JComboBox<String> combo = new JComboBox<>(choices); // The values are used as the elements of a new combo box created
        int choice = JOptionPane.showOptionDialog(frame, combo, titleBar, 0, JOptionPane.QUESTION_MESSAGE, null, options, options[0]);
        return new Pair<>(choice, combo.getSelectedItem());
    }
}
```

Figure 16: Dialogs Class

For example, to add new devices, the user provides multiple data; if they do not, the program may give an error and crash. Hence, the dialogue sub-procedures created here are used (See Figure 17). These sub-procedures are used in several classes, including the Dashboard class. Also, using try-catch prevents the program from crashing and shows the error as a dialogue, warning the user.

```java
private void createDeviceButtonActionPerformed(java.awt.event.ActionEvent evt) {

    /* The following if conditions are checking the requirements for adding devices or to prevent any errors to occur */
    if(newDeviceWarrantyDateCheckBox.isSelected() && newDeviceWarrantyDate.getDate() == null){
        // Showing user a warning dialog to enter a device warranty date if the warranty date check box is selected and the value is null
        Dialogs.infoBox("Garanti bitiş tarihini giriniz!", "Eksik Bilgi");
        return;
    }
    if(newDeviceName.getText() == null || newDeviceName.getText().equals("")){
        // Showing user another warning dialog to enter a device name which is essential to differentiate the items and prevent errors
        Dialogs.infoBox("Lütfen cihaz ismi giriniz!", "Eksik Bilgi");
        return;
    }
    if (newDeviceCategory.getSelectedItem() == null) {
        // Showing user another warning dialog to select a category if the user does not select to prevent any errors
        Dialogs.infoBox("Lütfen bir kategori seçiniz!", "Eksik Bilgi");
        return;
    }

    /* Initialising each value in the text fields and combo box to save them as device credentials. */
    String deviceName = newDeviceName.getText(), category = newDeviceCategory.getSelectedItem().toString(),
            p1 = property1.getText(), p2 = property2.getText(), type = newDeviceWarrantyType.getSelectedItem().toString(),
            p3 = property3.getText(), p4 = property4.getText(), p5 = property5.getText(), p6 = property6.getText(),
            p7 = property7.getText(), p8 = property8.getText(), p9 = property9.getText(), p10 = property10.getText(),
            snum1 = newDeviceSerialNum1.getText(), snum2 = newDeviceSerialNum2.getText(), snum3 = newDeviceSerialNum3.getText(),
            snum4 = newDeviceSerialNum4.getText(), snum5 = newDeviceSerialNum5.getText(), snum6 = newDeviceSerialNum6.getText(),
            snum7 = newDeviceSerialNum7.getText(), snum8 = newDeviceSerialNum8.getText(), snum9 = newDeviceSerialNum9.getText(),
            snum10 = newDeviceSerialNum10.getText();
    int deviceNum = (Integer) newDeviceAmount.getValue(), warranty = (Integer) newDeviceWarranty.getValue();

    List<String> serialNums = new ArrayList<>(Arrays.asList(snum1, snum2, snum3, snum4, snum5, snum6, snum7, snum8, snum9, snum10));
    List<String> properties = new ArrayList<>(Arrays.asList(p1, p2, p3, p4, p5, p6, p7, p8, p9, p10));

    if (deviceNum <= 0) { // Showing user an error dialog to warn him/her to enter a number of device to be registered
        Dialogs.infoBox("Cihaz adedi 1'den küçük olamaz!", "Eksik Bilgi");
        return;
    }

    int x =0;
    for(String serialNum : serialNums){ // Calculating the number of the serial numbers with a for loop
        if(serialNum == null || serialNum.equals(""))
            x++;
    }

    if(10-x<deviceNum){ // Showing user an error dialog to warn him/her to enter serial numbers in specified number above.
        Dialogs.infoBox("Belirtilen sayıda seri numarası girilmedi!", "Eksik Bilgi");
        return;
    }

    /* Initialising the warranty date and converting it to LocalDate which is compatible with the database. */
    Date date = newDeviceWarrantyDate.getDate();
    LocalDate warrantyDate = newDeviceWarrantyDate.getDate() == null ? null: date.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();

    /* Adding devices to database with the following sub-procedure with all the credentials for the device(s) to be stored. */
    sdb.addDevices(deviceName, category, warrantyDate, warranty, type, properties, serialNums);
}
```

Figure 17: An Example usage of dialogs created

Serialisation

The client requested that the data listed in tables to be serialised and that the program can export data to and import data from CSV Files. To generate CSV File, I used built-in functionalities FileOutputStream, OutputStreamWriter and BufferedWriter, according to my learnings from w3schools[6]. With these functions, I could focus on the algorithmic side of creating CSV Files. The main methods are kept in DashboardDatabase to access easily (See Figure 18). At the same time, the connection with the data is provided by the "exportTable" method in the Dashboard class (See Figure 10), fulfilling success criterion 16.

[6] https://www.w3schools.com/java/

```java
/** This method generates a CSV File containing the information of inventory in the target location
 * with the path passed as a parameter
 * @param path This is the path of target location that the CSV File will be generated.
 */
public void exportInventory(String path) {
    try { // Try Catch prevents any crash due to errors.
        st = cn.createStatement();
        String query = "SELECT * FROM application.inventory";
        rs = st.executeQuery(query);

        File file = new File(path); // The file is created with the path of location and name of the file
        try ( FileOutputStream fos = new FileOutputStream(file);
                OutputStreamWriter osw = new OutputStreamWriter(fos, Charset.forName("ISO-8859-9"));
                BufferedWriter writer = new BufferedWriter(osw)) { // The file is started to be read
            String line = "Cihaz ID,Kategori,İsim,Durum,Atanan Profil,Atanan Profil ID,Garanti Bitiş Tarihi,Seri Numarası";
            writer.append(line); // Appending first line as the titles of the columns
            writer.newLine(); // Starting to a new line
            while (rs.next()) { // Iterating until there isn't any elements left in ResultSet
                String id = rs.getString("deviceID");
                String category = rs.getString("category");
                String deviceName = rs.getString("deviceName");
                String status = rs.getString("status");
                String assignedProfile = rs.getString("assignedProfile");
                String assignedProfileID = rs.getString("assignedProfileID");
                String warranty = rs.getDate("warranty").toString();
                String serialNum = rs.getString("serialNum");
                line = id + "," + category + "," + deviceName + "," + status + "," + assignedProfile +
                        "," + assignedProfileID + "," + warranty + "," + serialNum;
                writer.append(line); // Appending each row created with the values returned from database
                writer.newLine(); // Starting to a new line
            }

        } catch (IOException e) {
            System.out.println(e); // Logging any error to console
        }

        // Informing the user about the success and the location of file created
        Dialogs.infoBox("Envanter tablosu CSV olarak aşağıdaki dosya yoluna kaydedildi:\n"+file.getPath(), "");

    } catch (SQLException e) {
        System.out.println(e);
    }
}
```

Figure 18: Main method of exporting inventory data as CSV File

```
Cihaz ID,Kategori,İsim,Durum,Atanan Profil,Atanan Profil ID,Garanti Bitiş Tarihi,Seri Numarası
7,monitör,flipbook,Kullanımda,Arif Aygün,1,2022-06-28,542523423
8,monitör,flipbook,Boşta,null,null,2022-06-28,4234235151
9,monitör,hp flipbook,Kullanımda,mehmet ,8,2022-06-16,14512341234
10,monitör,flipbook,Kullanımda,Mehmet Ki,6,2022-11-11,542523423
11,monitör,flip,Boşta,null,null,2022-06-03,4234235151
12,monitör,flip,Boşta,null,null,2022-06-03,14512341234
14,monitör,cool,Boşta,null,null,2022-08-27,021491324913
15,monitör,cool,boşta,null,null,2022-08-27,344352450898
16,monitör,looc,Boşta,null,null,2022-09-28,222223213123
17,monitör,looc,boşta,null,null,2022-09-28,021491324913
19,monitör,ac,Kullanımda,Mehmet Ki,6,2025-06-28,222223213123
20,monitör,ac,Boşta,null,null,2025-06-28,021491324913
```

Figure 19: Example CSV output of inventory table, opened in Notepad

Database

Storing and accessing data is an essential point of a management program. Since it is possible to access it online, meeting the client's needs, MYSQL is used for the program. I learned MySQL operations from a website[7].

```java
st = cn.createStatement(); // Using the connection created with MYSQL.

String query1 = "SELECT * FROM application.inventory WHERE deviceName='"+ deviceName +"'";
rs = st.executeQuery(query1); // Creating a query to get the devices with the name passed through deviceName variable and executing the query.
if(rs.next()){ // Controlling if there is already a device with the given name and informing the user, if not the loop exits.
    int answer = Dialogs.questionBox(null, "Bu isme sahip bir cihaz zaten kaydedilmiş! Devam etmek istiyor musunuz?", "Veritabanı Hatası!");
    if(answer == 1)
        return;
}

/* Generating a String text in a loop for a proper format to pass it to MYSQL as a query to insert the
properties coming with the input deviceName, then executing the query. */
String values = "";
for(String next : properties){
    if(next!=null && !next.equals("")){
        values += ", '"+next+"'";
    }else{
        values += ", NULL";
    }
}
String query3 = "INSERT INTO application.properties (deviceName, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10) VALUES ('"+deviceName+"'"+values+")";
String query2 = "SELECT * FROM application.inventory";
rs = st.executeQuery(query2);
List<String> serNums = new ArrayList<>();
while(rs.next()){
    serNums.add(rs.getString("serialNum"));
}

/* Getting all the serial numbers and looping through each of them
to generate a query String text to insert the new devices */
String p = "";
for(String next : serialNums){
    /* If one or more of the serial numbers entered is already registered,
    the user is informed about that serial number with a diologue box. */
    if(serNums.contains(next) && !next.equals("")){
        Dialogs.infoBox(next+" seri numarası zaten kaydedilmiş! Lütfen kontrol ediniz.", "Hata");
        return;
    }
    /* If the variable 'next' is not null amd not empty, the device is added to the query. */
    if(next!=null && !next.equals("")){
        p += "('"+category+"', '"+deviceName+"', 'boşta', '"+warrantyDate+"', '"+next+"'),";
    }
}
StringBuffer sb= new StringBuffer(p);
sb.deleteCharAt(sb.length()-1); // Removing the last character of p String as it has a , at the end due to loop.

st.executeUpdate(query3); // Executing the update with query of properties that was generated previously.
// Inserting the new devices into database with the correct details.
st.executeUpdate("INSERT INTO application.inventory (category, deviceName, status, warranty, serialNum) VALUES "+sb+";");
Dialogs.infoBox("Cihaz(lar) başarıyla kaydedildi!", "Başarılı"); // Informing the user that the process was successful.
rs = st.executeQuery("SELECT * FROM application.inventory WHERE deviceName='"+deviceName+"'");

/* Initialising an ArrayList for the ids of the new devices and adding each id to the ArrayList. */
List<String> deviceIDs = new ArrayList<>();
while(rs.next()){
    deviceIDs.add(String.valueOf(rs.getInt("deviceID")));
}

/* Creating log registry for each device in the deviceIDs array list with a dedicated method. */
deviceIDs.forEach(id -> {
    log.addDeviceChange(id, "Cihaz Eklendi", null, Variables.uname);
});
```

Figure 20: Connection with MYSQL and Query Execute-Update Methods

[7] LearnSQL.com

External Libraries Used

- FlatLaf-2.4
  - Learned from GitHub[8]
- Mysql-connector-java-8.0.29
- Swingx-1.6
- Jcalendar-1.4
  - Learned from GitHub[9]
- Javax.mail-1.6.2
  - Learned from Oracle[10]
- Javax.activation-1.2.0
  - Learned from Oracle[11]

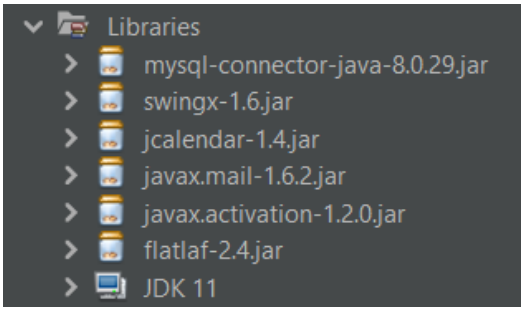

Figure 21: External Libraries Used

System Requirements[12]

[8] https://github.com/JFormDesigner/FlatLaf

[9] https://github.com/toedter/jcalendar

[10] https://docs.oracle.com/javaee/7/api/javax/mail/package-summary.html

[11] https://docs.oracle.com/javaee/7/api/javax/mail/package-summary.html

[12] OS: Windows or MacOS, Storage: 6 MB available space, Java: JDK 11, Database: MySQL