Technical Report

Dashboard

This class is where all of the information and the interface is displayed, and where the user enters all of the information through the console. The frontend of the program. The entered information is later stored in the backend classes such as the “SmartHome” and “SmartPlug”. I decided to do it that way because it is in line with the constraint that says the frontend of the program should only take inputs and output to the console and the backend is where that data should be stored. Throughout the whole “Dashboard” the program takes inputs from the user such as, for e.g., the number of rooms in the house or which smart plug to turn on or off, and stores them in the “SmartHome” and “SmartPlug” classes respectively.

In addition, there is one more decision in this class which needs explaining. It is my use of switch statements through line 66 to 169 of my code. I used switch statements for choosing different options when there are more than two conditions to compare and when the actual condition is simple and obvious like comparing an input to an option number. I did this because it makes the syntax cleaner and is more efficient than an if statement. Also at one point throughout development, when I used an if statement for selecting options, the option number would need to be input multiple times for it to be registered depending on the number.

```
helper.printStr( prompt: "-------------MENU OPTIONS-------------\n" +
        "---------please select option:--------\n" +
        "1 - house level options\n" +
        "2 - room level options\n" +
        "3 - plug level options\n" +
        "4 - system options\n");

switch(helper.intInput()) {
    case 1:
        helper.printStr( prompt: "\n" +
                "HOUSE LEVEL OPTIONS\n" +
                "1 - Switch all plugs off\n" +
                "2 - Switch all plugs on\n" +
                "Select an option\n");
        smartHome.plugs.changeAllPlugsStates(helper.intInput());
        break;
    case 2:
        helper.printStr( prompt: "ROOMS AVAILABLE: ");

        for(int i = 0; i < smartHome.getSize(); i++) {
            helper.printStr( prompt: smartHome.getID(i) + " - " + smartHo
        }
```

Short part of switch statements

ConsoleHelper

This is the class that takes care of storing inputs from the user and outputting to the actual console. This was done because the frontend, “Dashboard”, is not supposed to store any data according to the given constraints. The class takes string and integer inputs, stores them and returns them. Also, it outputs strings into the console.

```
public class ConsoleHelper {
    private int lastIntInput;

    public int intInput() {
        Scanner input = new Scanner(System.in);
        lastIntInput = input.nextInt();
        return lastIntInput;
    }

    public int returnLastIntInput() { return lastIntInput; }

    public String strInput() {
        Scanner input = new Scanner (System.in);
        return input.nextLine();
    }

    public void printStr(String prompt) { System.out.print(prompt); }
}
```

ConsoleHelper class

SmartHome

```
public class SmartHome {
    private static int size;
    private String[] rooms;
    private int chosenRoom;
    private int[] roomID;
    SmartPlug plugs = new SmartPlug();

    public SmartHome(int roomSize) {
        this.rooms = new String[roomSize];
        setSize();
        setIDSize();
    }

    public void setIDSize() { this.roomID = new int[size]; }

    public void populateID(int index) {
        roomID[index] = index + 1;
    }
    public void populateRooms(int index, String roomName) {
        rooms[index] = roomName;

    }
```

SmartHome class

In this class, mainly the data in relation to the rooms as opposed to the smart plugs is managed (with the exception of one method). It takes the data about the rooms input by the user and stores it in the backend which also complies with the constraints. The class sets the number of rooms, size of the roomID (used for identifying the different rooms and comparing them in if statements throughout the “Dashboard”), populates the room string array with the names of the rooms and the roomID array with the identifiers for all of the rooms. This is done through the use of for loops in the “Dashboard” and calls methods from the “SmartHome”. For loops are used due the constraints not permitting the use of java.util.Arrays in order to populate an array. This class is also the only point of contact with the “Dashboard” and the “SmartPlug" classes. For the “Dashboard” to use the “SmartPlug” class it first refers to the object made from the “SmartHome” class in the “Dashboard” and then to the “SmartPlug” object made in the “SmartHome” class. For e.g., if the plugID in the “SmartPlug” class needs to be populated, the line of code reads: **smartHome.plugs.populatePlugID()**. I did this because it avoids unnecessary methods in the “SmartHome” class that would need to call upon methods from the “SmartPlug” class, resulting in cleaner code and less lines of code.

In addition, there is one more decision that may need explaining. That is the decision of making selectRoom and returnSelectedRoom methods. They take in the “Dashboard” the input which selects a room. I decided to do this because there are multiple occurrences in the “Dashboard” where more than one input needs to be taken, and once there was a prompt for the second input, the first input would be overwritten by the second one therefore I needed a way to store the first input.

```
public void selectRoom(int chosenRoomInput) { chosenRoom = chosenRoomInput; }

public int returnSelectedRoom() { return chosenRoom; }
```

SmartHome class

SmartPlug

This final class deals with everything related to the smart plugs and the different devices that can be attached to the smart plugs. Just like the “SmartHome” class it takes inputs from the “Dashboard” through the use of the “ConsoleHelper” and stores them in the backend in the “SmartPlug” class, which is also in line with the given constraint of storing data only in the the backend of the program. This class sets the size of the multiple arrays that are in this class, it creates the array with the device types. One identifier that may need explaining is the plugToRoomID. This array stores the roomID in relation to the plugID. Throughout my code I treat the index of the array as the plugID and assign a roomID to the correct plugID. I’ve done this because I make use of it in multiple if statements in the “Dashboard” in order to print correct plugs that are located in the correct room.

```
public class SmartPlug {
    private String[] deviceTypes = {"Lamp", "TV", "Computer", "Phone Recharger", "Heater"};
    private String[] attachedDevices;
    private int[] plugToRoomID;
    private int[] plugID;
    private boolean[] plugState;
```

Arrays in the SmartPlug class

This class also populates all of the other arrays such as the attached devices (which only stores devices that are attached to a plug), different plug identifiers and the plugState array which stores whether a plug is turned on or off and the two identifiers that are in this class. The arrays that are populated outside of the for loops in the “Dashboard” class are populated using for loops in the “SmartPlug” class whereas those methods that need to be called inside of a for loop in the “Dashboard” use counters that are set to 0 when they are declared as variables at the very beginning of this class. This was done because I found that if I didn’t use counters, one of the positions in the array (either beginning or the end depending on the index) that I wanted to populate would be null.

```
public void populatePlugToRoomID(int roomID) {
    plugToRoomID[plugCounter] = roomID;
    plugCounter++;
}

public void populatePlugID() {
    for(int i = 0; i < size; i++) {
        plugID[i] = i + 1;
    }
}
```

```
public void populateAttachedDevices(int deviceID) {
    attachedDevices[deviceCounter] = deviceTypes[deviceID - 1];
    deviceCounter++;
}
```

```
public void setInitialPlugStates() {
    for(int i = 0; i < plugState.length; i++) {
        plugState[i] = false;
    }
```

Methods populating arrays in “SmartPlug”

There are multiple options that can be chosen throughout the “Dashboard” that can be chosen in order to customize all of the plugs, such as turning them on or off, changing the attached device or moving them to different rooms. This class also stores the last chosen plug that was input in the “Dashboard” for the same purpose as the “SmartHome” class.

```
public void amendPlug(int chosenDevice) {
    attachedDevices[getSelectedPlug()] = deviceTypes[chosenDevice];
}
```

```
public void changePlugToDiffRoom(int chosenRoom) {
    plugToRoomID[getSelectedPlug()] = chosenRoom;
}
```

Methods changing plug’s device and room

Lastly, there are three methods that change the states of the plugs. One that changes states of all of the plugs, second one that changes states of plugs in certain rooms and last one that changes the states of individual plugs. This was done so it was easier to identify which option they correlate with as there is an option related to each of those methods in the “Dashboard.”

```
public void changeAllPlugsStates(int choice) {
    if(choice == 1) {
        for( int i =0; i < plugState.length; i++) {
            plugState[i] = false;
        }
    }else {
        for( int i =0; i < plugState.length; i++) {
            plugState[i] = true;
        }
    }
}
public void changeRoomPlugsStates(int choice, int selectedRoom) {
    if(choice == 1) {
        for(int i = 0; i < size; i++) {
            if(selectedRoom == plugToRoomID[i]) {
                plugState[i] = false;
            }
        }
    }else if(choice == 2) {
        for(int i = 0; i < size; i++) {
            if(selectedRoom == plugToRoomID[i]) {
                plugState[i] = true;
            }
        }
    }
}
public void changeIndividualPlugState(int choice) {
    if(choice == 1) {
        plugState[chosenPlug] = false;
    }else {
        plugState[chosenPlug] = true;
    }
}
```

Methods changing states of the plugs