2021

Technical Report

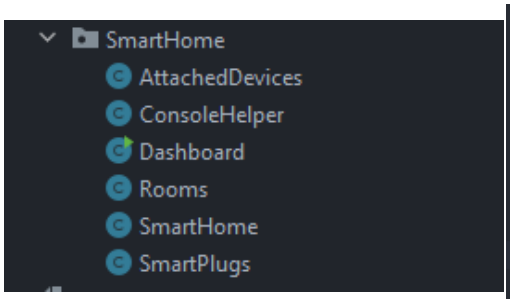
ALIKOTI SMART HOME

SAMUEL THOMPSON – UP2040452

CONSTRAINTS

Infrastructure

The program consisted of frontend and backend objects and classes. The frontend contained the ConsoleHelper object, which managed console interaction and contained wrapper functions to keep the dashboard tidy, and the dashboard class. To separate responsibilities and reduce code complexity, I decided to use the two optional backend objects, which were the room and attached device objects. These objects contained methods that would get and set their ID or name. There was also a SmartHome backend object, which contained the SmartPlugs, rooms, and devices arrays, and contained most methods. The SmartPlugs backend object had get and set methods and was also responsible for creating new room and device objects.



1 The list of java classes

```
public SmartPlugs(int roomNumber, int deviceNumber, int roomID, int deviceID) {
    this.roomNumber = roomNumber;
    this.deviceNumber = deviceNumber;
    this.status = false;
    this.roomID = roomID;
    this.deviceID = deviceID;
    setRoomNumber(roomNumber);
    setDeviceNumber(deviceNumber);
}

public void setRoomNumber(int roomNumber) { room = new Rooms(roomNumber, roomID); }
```

2 Some code from the SmartPlugs object

```
public class Rooms {
    private int roomNumber;
    private int roomID;

    public Rooms(int roomNumber, int roomID) {
        this.roomNumber = roomNumber;
        this.roomID = roomID;
    }

    public int getRoomName() { return roomNumber; }

    public void setRoom(int newRoomNumber) {this.roomNumber = newRoomNumber;}

    public int getRoomID() { return roomID; }

    public int getRoomNumber() { return roomNumber; }


}
```

3 All of the code in the rooms object

```
public class SmartHome {

    private SmartPlugs[] plugs;
    public String[] roomNames;
    public String[] deviceList;
    private int currentIndex;
    public int roomCount;
    public int plugCount;
    private int counter;
    public int deviceCount = 5;
    public int roomID;
    public int deviceID;
```

4 The list of variables and arrays within the SmartHome object

```java
public class AttachedDevices {
    private int deviceNumber;



    public AttachedDevices(int deviceNumber, int deviceID) {
        this.deviceNumber = deviceNumber;
    }

    public int getDeviceName() { return this.deviceNumber; }

    public void setDevice(int newDeviceNumber) { this.deviceNumber = newDeviceNumber; }

}
```

5 All of the code in the AttachedDevices object

Room & Plug Combinations

You can have zero, one or many smart plugs in a single room, but you cannot have one smart plug in several rooms. When the user enters the names for each room, the room is given an ID. The user uses this ID to determine where each smart plug will go. This allows us to have multiple plugs with the same room ID, as well as rooms with no smart plugs in. The program uses a loop which runs as many times as the specified number of smart plugs. This means one smart plug cannot be in several locations.

```java
    public void populateHome(SmartHome home) {
        for (int i = 0; i < home.roomCount(); i++) { //Loop runs from 0 to the number of rooms
            String roomName = getString( instructions: "Please provide a name for room number " + (i + 1) + ": ");
            home.newRoom(roomName); // Add the new room to the smart home
        }
        for (int i = 0; i < home.plugCount(); i++) { //Loop runs from 0 to the number of plugs
            populateRoom(home); // Call the function which asks for the room and device
        }
    }
```

```
---------------DASHBOARD--------------
ROOM: 1
SmartPlug | attached to: Lamp | room: Kitchen | ID: 1 | status: Off
SmartPlug | attached to: TV | room: Kitchen | ID: 2 | status: Off

ROOM: 2
SmartPlug | attached to: Lamp | room: Office | ID: 3 | status: Off
SmartPlug | attached to: TV | room: Office | ID: 4 | status: Off

ROOM: 3
SmartPlug | attached to: Computer | room: Bedroom | ID: 5 | status: Off
```

User Interface

Almost all user input is integer based. The user will input the number of rooms and plugs, as well as using digits to select a room and an attached object for each plug. The only string-based inputs are for room names and adding custom devices.

```
How many rooms are there in this property?
3
How many plugs do you want to place in this property?
5
Please provide a name for room number 1:
Kitchen
Please provide a name for room number 2:
Office
Please provide a name for room number 3:
Bedroom
ENTER PLUG INFORMATION BELOW
```

The dashboard is shown below:

```
---------------DASHBOARD--------------
ROOM: 1
SmartPlug | attached to: TV | room: Bedroom | ID: 1 | status: Off

ROOM: 2
SmartPlug | attached to: Computer | room: Office | ID: 2 | status: Off
SmartPlug | attached to: Heater | room: Office | ID: 3 | status: Off


-------------MENU OPTIONS-------------
1 - House level options
2 - Room level options
3 - Plug level options
4 - System options
Please choose an option from the above list (integer input only):
```

Exception Handling

It is stated that correct input is assumed, and as such there is no exception handling in the program. This is because exceptions do not need to be caught if they will not occur.

Libraries

The only Java library I used was **java.util.Scanner**. This was used in the ConsoleHelper to scan inputs from the console. Arraylists and array copy functions were not permitted, so I created deep copies of arrays when they needed to be resized. This was necessary in the system options section, since the user can choose to add more plugs, rooms, or devices to their respective arrays.

```java
public String[] duplicateArray(String[] originalArray, int size) {
    String[] arrayItems = new String[size]; //create new array with increased size
    for (int i = 0; i < originalArray.length; i++) {
        arrayItems[i] = originalArray[i]; //copy each item from the original array to the new array
    }
    return arrayItems; //return the new array
}
```

Complexity

All methods and functions are given clear names so that their meanings are easily discernible, and comments are scattered throughout the code to improve readability. Variable and object names are also meaningful to prevent confusion (except for variable counters in loops, where **i** is used). The code is not overly complex, and only does what is required.

Extra Functionality

As outlined in the document, no additional functionality is requested. As such, the program contains all requested functionality and nothing more.

CORE DEVELOPMENT TASKS

Key Coding Decisions

The frontend of the project consisted of a ConsoleHelper object and the Dashboard class. The ConsoleHelper object is created in the dashboard so its methods can be used to get initial user input. When the user first runs the program, they would be asked to enter the number of rooms in the home, and the number of plugs in the home. These numbers would then be used to build the SmartHome object.

```
public class Dashboard {

    public static void main(String[] args) {
        ConsoleHelper console = new ConsoleHelper();
        int numRooms = console.getInt( instructions: "How many rooms are there in this property?");
        int numPlugs = console.getInt( instructions: "How many plugs do you want to place in this property?");
        SmartHome home = new SmartHome(numPlugs, numRooms);
        console.populateHome(home);
```

On the backend, the SmartHome object was where most methods were contained. I decided to store the SmartPlugs, rooms and devices in arrays within this object. The SmartPlugs were stored in array of SmartPlug objects, whereas the rooms and devices were in string arrays. The sizes of these arrays were determined by the user at the start of the program, excluding the devices array which was predetermined.

```
public class SmartHome {

    private SmartPlugs[] plugs;
    public String[] roomNames;
    public String[] deviceList;
```

```
    public SmartHome(int numPlugs, int numRooms) {
        plugCount = numPlugs;
        plugs = new SmartPlugs[plugCount];
        roomCount = numRooms;
        roomNames = new String[numRooms];
        deviceList = new String[deviceCount];
        populateDeviceArray();
    }
```

I also decided to utilise the optional backend objects. These were the rooms object and the attached devices object. These objects contained methods which would get and set their ID and name. Each SmartPlug object would create its own room and device object, meaning the plug object was responsible for storing information relating to the plug.

```
public class SmartPlugs {
    private Rooms room;
    private AttachedDevices devices;
```

```
public SmartPlugs(int roomNumber, int deviceNumber, int roomID, int deviceID) {
    this.roomNumber = roomNumber;
    this.deviceNumber = deviceNumber;
    this.status = false;
    this.roomID = roomID;
    this.deviceID = deviceID;
    setRoomNumber(roomNumber);
    setDeviceNumber(deviceNumber);
}
```

```
public void setRoomNumber(int roomNumber) { room = new Rooms(roomNumber, roomID); }

public int getRoomName() { return room.getRoomName(); }

public int getRoomNumber() { return room.getRoomNumber();}

public void setDeviceNumber(int deviceNumber) {devices = new AttachedDevices(deviceNumber, deviceID); }

public void changeDevice(int newDeviceNumber) {devices.setDevice(newDeviceNumber);}

public void changeRoom(int roomNumber) {room.setRoom(roomNumber);}

public int getDeviceName() { return devices.getDeviceName(); }
```

The ConsoleHelper object contained many wrapper functions that kept the frontend code tidy. In the dashboard, I pass the home object to the ConsoleHelper as a parameter, meaning it can then use the SmartHome methods.

```
SmartHome home = new SmartHome(numPlugs, numRooms);
console.populateHome(home);
```

The ConsoleHelper was then responsible for creating and updating the dashboard, as well as adding new plugs, rooms, or devices, among other things. The ConsoleHelper would manage all the looping and inputs, and then call SmartHome methods which would manipulate the arrays. This ensures the user is not directly interacting with the backend. An example is shown below.

```
public void addNewDevice(SmartHome home, int numNewDevices) {
    for (int i = 0; i < numNewDevices; i++) {
        String deviceName = getString( instructions: "Please provide a name for the new device: ");
        home.newDevice(deviceName);
    }
}
```

Another key decision was how the dashboard update options were managed. Each option level used a switch statement, and the user input determined the method that was called.

```
public void displayHouseOptions(SmartHome home) {
    print("HOUSE LEVEL OPTIONS\n" +
    "1 - Switch all plugs off\n" +
    "2 - Switch all plugs on\n" +
    "Please choose an option from the above list (integer input only): ");
    int optionLevel = getInt( instructions: "Select an option");
    switch (optionLevel) {
        case 1:
            home.allPlugsOff( wholeHouse: true, optionLevel);
            break;
        case 2:
            home.allPlugsOn( wholeHouse: true, optionLevel);
            break;
    }
    updateDashboard(home);
}
```

The options to toggle all plugs in the house or room were simple since the program can just loop through all plugs and change their status.

```
public void allPlugsOn(boolean wholeHouse, int roomID) {
    if (wholeHouse) {
        for (int i = 0; i < plugCount(); i++) {
            plugs[i].setStatus(true);
        }
```

The method is passed a Boolean parameter that will be true if all plugs in the house are to be toggled. If this is false, the program will only toggle plugs that match the provided room ID.

```
public void allPlugsOff(boolean wholeHouse, int roomID) {
    if (wholeHouse) {
        for (int i = 0; i < plugCount(); i++) {
            plugs[i].setStatus(false);
        }
    } else {
        for (int i = 0; i < plugCount(); i++) {
            if (plugs[i].getRoomNumber() == roomID) {
                plugs[i].setStatus(false);
            }
        }
    }
}
```

When the user chose to toggle a single plug, they would provide the plug's ID and the program would manipulate only that plug.

```java
public void togglePlug(int plugID, int status) {
    boolean toggle;
    if (status == 1) { toggle = true;}
    else if (status == 0) { toggle = false;}
    else {return;}
    plugs[plugID].setStatus(toggle);
}
```

The technique for changing a plugs room or device is similar. The user would be shown a list of all plugs and would select a plug ID.

```
SmartPlug | attached to: TV | room: Bedroom | ID: 1 | status: Off
SmartPlug | attached to: Computer | room: Living Room | ID: 2 | status: Off
SmartPlug | attached to: Phone Recharger | room: Living Room | ID: 3 | status: Off

Please select a plug from the list above by entering its ID:
```

They would then be shown the options for manipulating the plug.

```
Please select a plug from the list above by entering its ID:
1
PLUG LEVEL OPTIONS
1 - Switch plug off
2 - Switch plug on
3 - Change attached device
4 - Move plug to different room
Please select an option
```

If they choose option 3, a list of devices would appear, and they would select a new device. If they choose option 4, a list of rooms would appear, and they would choose a new room.

```
Please select an option
3
1 - Lamp
2 - TV
3 - Computer
4 - Phone Recharger
5 - Heater

Please select a new device to attach to smart plug (integer input only):
```

```
Please select an option
4
1 - Bedroom | 2 - Living Room |
Please select a new room to move this smart plug to (integer input only):
```

The program would then pass these IDs as parameters to methods that update that certain plug.

```java
public void updateAttachedDevice(int plugID, int newDevice) { plugs[plugID].changeDevice(newDevice); }

public void updateRoom(int plugID, int newRoom) { plugs[plugID].changeRoom(newRoom); }
```

The system level options allow the user to add more plugs, rooms, or devices. This meant their arrays would have to be resized. I did this by passing the original array into a copy method as a parameter and copying its contents to an array with the increased size.

```java
public String[] duplicateArray(String[] originalArray, int size) {
    String[] arrayItems = new String[size]; //create new array with increased size
    for (int i = 0; i < originalArray.length; i++) {
        arrayItems[i] = originalArray[i]; //copy each item from the original array to the new array
    }
    return arrayItems; //return the new array
}
```

The method would return the new array, and the remaining blank space in the array would be filled with the user inputs (i.e., the user would enter new room names, or new attachable devices).