AUTOHOME-REPORT 2-FULL

Group 15: Mahmoud Bachir (Leader), Aaron Christie, Smruthi Srikumar, Gunbir Singh, Brendan Li, Kyle Ross, Jimmy Wen, Karthikey Thapliyal

AUTOHOME
https://github.com/mahmoudbachir/homeautomation
https://ruautohome.weebly.com/

Contents

Contributions Table 2
Interaction Diagrams 3
Class Diagram 7
Data Types and Operation Signatures 8
Traceability Matrix 15
Architectural Styles 17
Identifying Subsystems 18
Mapping Subsystems to Hardware 18
Persistent Data Storage 19
Network Protocol 19
Global Control Flow 20
Hardware Requirements 20
User Interface Design 21
Design of Tests 23
Project Management 43
References 43

Contributions Table

Part	Mahmoud	Aaron	Smruthi	Jimmy	Brendan	Kyle	Gunbir	Karthikey
UML Diagrams	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%
Class Diagram				100%				
Signatures			100%					
Arch Styles					100%			
Pckg diag								100%
Map hardware	100%							
Database	100%							
Traceability Matrix						100%		
Glob Cont Flow + Hard Req's							100%	
Network Protocol		100%						
UI Design		50%		50%				
UI Descrip					100%			
Testing Design	17%	21.2%	29.57%	17%	9.9%		5.6%	
Doc Merge	100%							
Proj coord	50%			50%				
Plan of wrk				100%				

Table 1: Individual Contributions Table

Interaction Diagrams

UC-1: Unlock

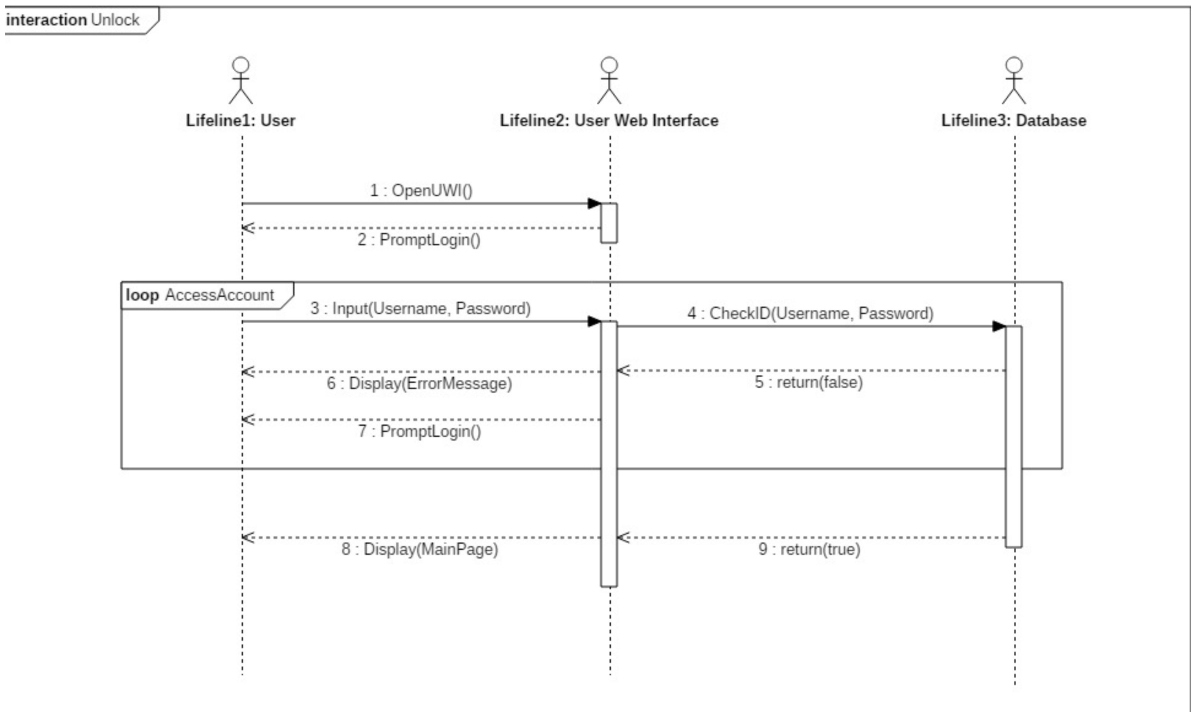


Figure 1: UC-1 Interaction Diagram

The purpose of Use-case 1 is for the user to be able to log into his/her account where all their settings are saved. This design allows for multiple profiles for different people around the house. The use-case allows communication with the server and then the database which stores the user's login information, without the user ever having to interact with the database, increasing security. The general flow of the use case will be that the user will input their username and password into the UI and system will check if the credentials are correct. If they are, the interface will display the user's list of devices they can control. If the entered credentials are incorrect, the interface will display an error message and prompt the user to enter their credentials again. This error loop will continue until the user enters the correct credentials.

UC-3 Add User

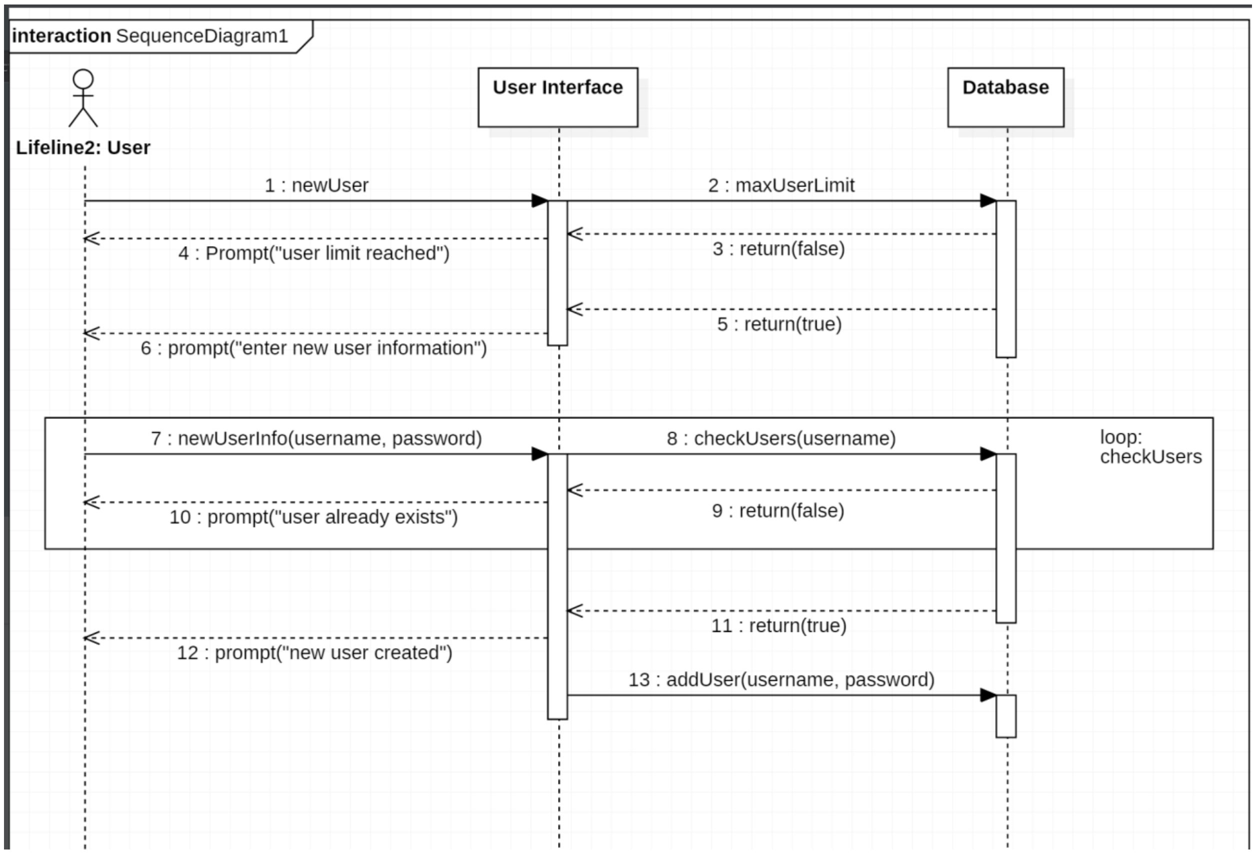

interaction SequenceDiagram1
Lifeline2: User
User Interface
Database
1 : newUser
2 : maxUserLimit
3 : return(false)
4 : Prompt("user limit reached")
5 : return(true)
6 : prompt("enter new user information")
7 : newUserInfo(username, password)
8 : checkUsers(username)
loop: checkUsers
9 : return(false)
10 : prompt("user already exists")
11 : return(true)
12 : prompt("new user created")
13 : addUser(username, password)

Figure 2: UC-3 Interaction Diagram

The purpose of Use-Case 3 is to be able to add other users to the application. This will allow ease of access when there are multiple people living in the same house because they will all have equal control over the autohome devices. This is also a method to keep the app secure because every user will require a password to gain access to the application and the devices that are controlled by the autohome.This use-case allows a new user to create a username and password so they can easily log in to the app whenever they need to. After this is done, the user information is sent to and stored in the database. Any two people may not have the same username. After the information is successfully stored in the database, a success message is returned to the user and the user can begin to use their account regularly.

UC-8 Add-Device

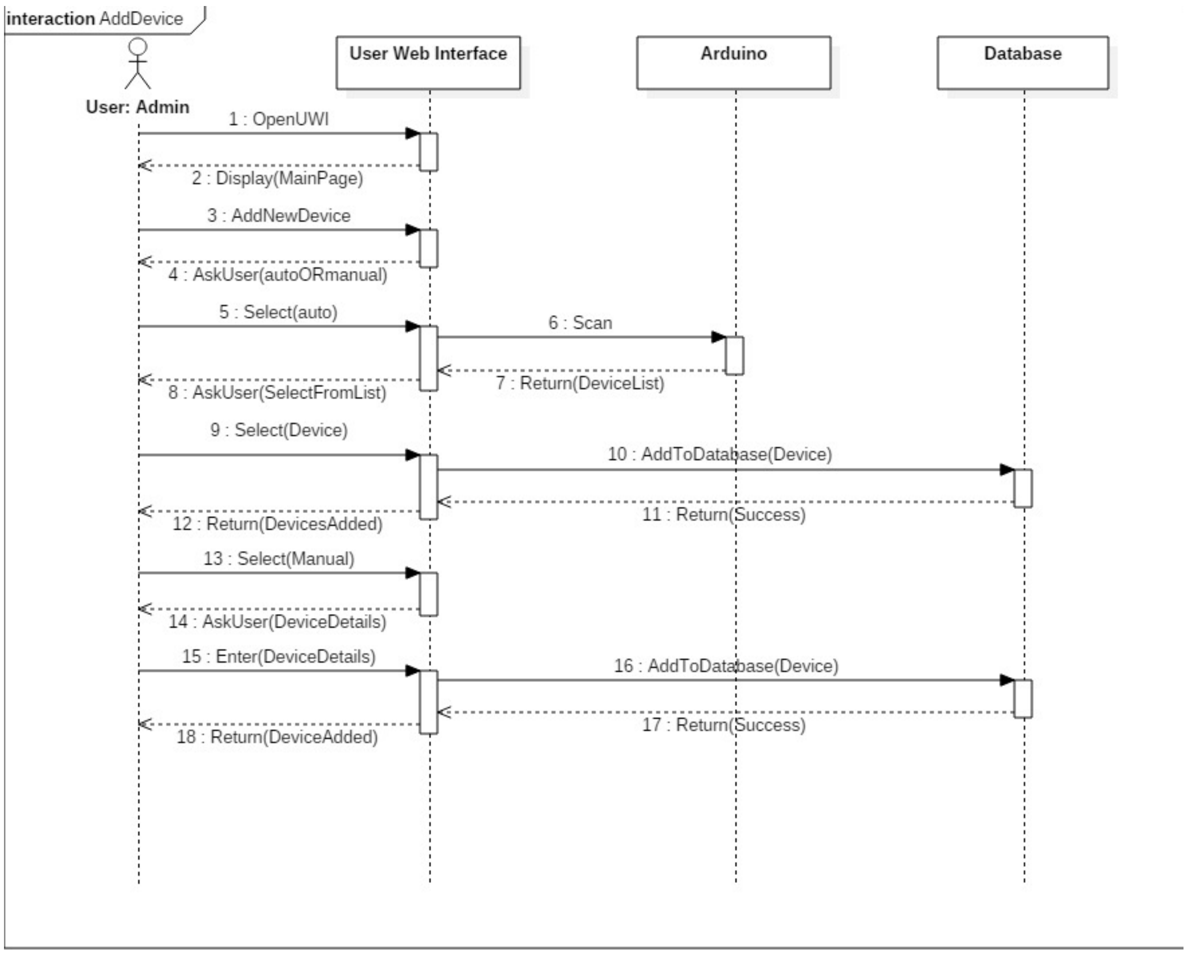


Figure 3: UC-8 Interaction Diagram

The purpose of Use-Case 8 is to be able to add in another device from the list of available autoHome devices. This will allow all the nearby available autoHome devices to be managed by the user through the user web interface. The use case allows the admin to configure the devices he wish to add. The admin prompts the user web interface to add new device which returns with an option to add the device manually or automatically. The admin thus selects the way it wants the device to configured and added. Choosing automatically makes the user web interface request the Arduino to detect all the devices near it. Thus, Arduino sends all possible connectable autoHome Devices to User Web Interface and then the User Web Interface displays all connectable devices to admin. The admin then selects the device and then confirms it. The device is then displayed to the Web user Interface and the User Web Interface adds device to database. Choosing manually just makes the Web user interface stored the configuration directly to the database. Finally, User Web Interface displays connected devices as well as actions for those devices.

UC-9 Device-Status

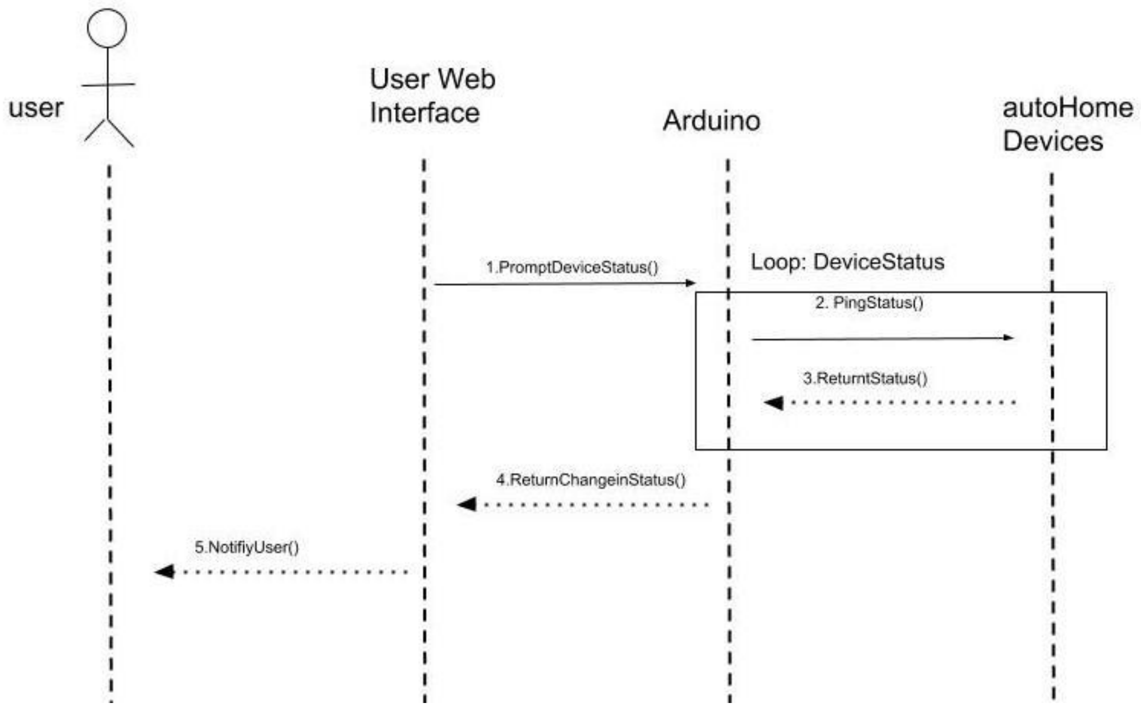


Figure 4: UC-9 Interaction Diagram

The purpose of UC-9 is for the User to be able to view the status of all of his devices at any given time. This will allow the user to be in complete control whenever/wherever he is. The way this use case is designed to work is the the user will login to the app,and then he will select the page labeled "Manage Devices". Once he picks that page, the web interface asks for the devices statuses, the Arduino will ask the devices for their statuses, and then this will be returned to the Web Interface. At this point the web interface refreshes with the updated status of each device. This page stays updated because it is constantly being updated after a certain interval of time.

Class Diagram

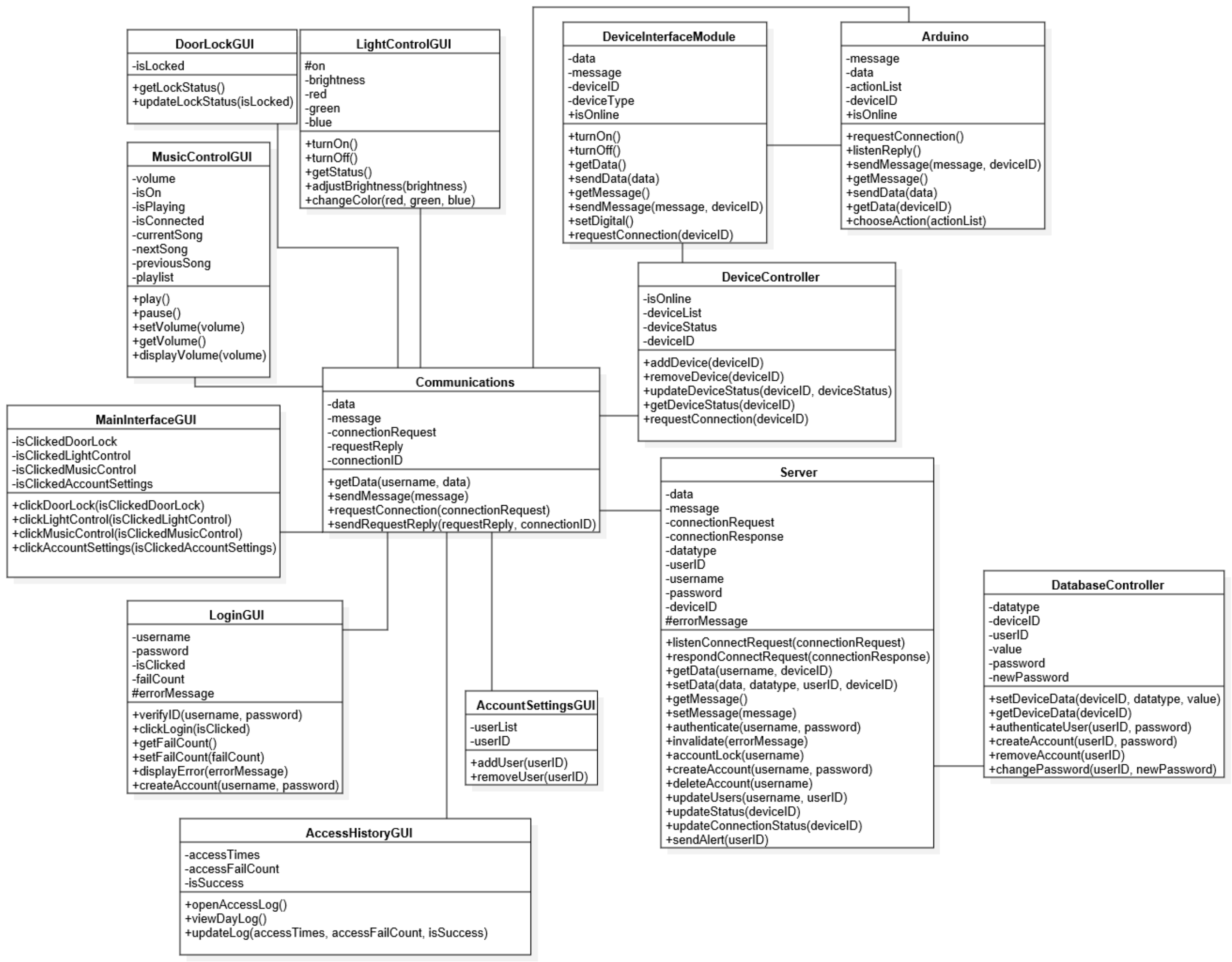


Figure 5: Class Diagram

Data Types and Operation Signatures

Communications:
Attributes:
data: int
message: string
connectionRequest: boolean - TRUE if request is successfully sent, FALSE otherwise.
requestReply: boolean - TRUE if connection request is approved, FALSE otherwise.
connectionID: int - a unique integer specifying the class with which the connection is established.

Methods:
getData(username, data): int - gets username and data
sendMessage(message): void - sends message
requestConnection(connectionRequest): boolean - uses connection request value to request a connection
sendRequestReply(requestReply, connectionID): void - sends request reply value and connection id

Server

Attributes:
data: int – information from Device to update status (color of bulb, door locked/open, etc.)
message: string – message that is sent to server
connectionRequest: oolean – true or false that says whether or not it is requested
connectionResponse: oolean – true or false the connection goes through or doesn't
datatype: (void*) – type of data (different for each device, int vs Boolean, etc.)
userID: string – ID given to user
username: string – string that user types to login
password: string – string that user types to login
deviceID: int – number used to identify different device
errorMessage: string – message sent to user in case of an error

Methods:
listenConnectRequest(connectionRequest): oolean – listens for a connection request. TRUE if received and FALSE if not.

<u>respondConnectRequest(connectionResponse):</u> oolean – sends a response to a connection request
<u>getData(username, deviceID):</u> (void*) – gets data associated with an account and its devices from the database
<u>setData(data, datatype, userID, deviceID):</u> void – sets device data associated with a user's account
<u>getMessage():</u> string – gets a message from a class
<u>setMessage(message):</u> void – sets a return message to send back to a class
<u>authenticate(username, password):</u> oolean – checks the username and password pair for correct matching.
<u>Invalidate(errorMessage):</u> int – invalidates attempt to access account when username and password do not match. Returns an error message to notify user that the attempt failed. Returns an integer to add one to the number of consecutive failed attempts.
<u>accountLock(username):</u> oolean – Locks or unlocks a user's account. First checks the current status, then flips the status.
<u>createAccount(username, password):</u> int – creates an account with the associated username and password. Returns 1 if successful, 0 if failure due to an existing account with the same username and -1 otherwise.
<u>deleteAccount(username):</u> int – deletes an account in the database. Returns 1 if successful, 0 if unsuccessful because the username does not exist in the database and -1 for any other error.
<u>updateUsers(username, userID):</u> int – updates users associated with an account in the database. Returns 1 if successful, 0 if unsuccessful because the username does not exist in the database and -1 otherwise.
<u>updateStatus(deviceID):</u> void – periodically updates the database's record of a device's members (e.g. light brightness, light RGB values, music volume, etc.)
<u>updateconnectionStatus(deviceID):</u> void – periodically updates a device's connection status
<u>sendAlert(userID):</u> void – sends an alert to the user's account (e.g. too many failed login attempts)

<u>DeviceInterfaceModule</u>

Attributes:
<u>data</u>: int – data associated with a device
<u>message:</u> string – a string that holds a set of instructions for or from a specific device
<u>deviceID:</u> int – a device identification number unique within the user's account
<u>deviceType:</u> string – contains information about the kind of device this is. This is sent as data
<u>isOnline</u>: oolean – indicates if a device is connected and online

Methods:

turnOn(): oolean – turns a device on. Returns TRUE if successful. Returns FALSE otherwise
oolean(): oolean – turns a device off. Returns TRUE if successful. Returns FALSE otherwise
getData(): int – gets data from a device
sendData(data): void – sends data to Arduino
getMessage(): string – gets a message sent from a device.
sendMessage(message, deviceID): void – sends a message to a specific device.
setDigital(): void – sets a digital value.
requestConnection(deviceID): oolean – returns TRUE if connection request was approved. Returns false otherwise.

Arduino

Attributes:
message: string – a string that holds a set of instructions for a specific device
data: int – data associated with a device
actionList: int array – an array that holds all the valid actions for a device
deviceID: int – a device identification number unique within the user's account
isOnline: oolean – indicates whether or not Arduino is connected and online

Methods:
requestConnection(): void – sends a connection request to server via communications
listenReply(): oolean – listens for a reply to connection request. Returns the reply as TRUE or FALSE.
sendMessage(message, deviceID): void – sends a message to communications
getMessage(): string – gets a message from communications
sendData(data): int – sends data to communications
getData(deviceID): int array – gets data associated with a specific device
chooseAction(actionList): int – chooses an action from a specific action list associated with a device. Actions are indexed and this method returns the index of the associated action.

DatabaseController

Attributes:
datatype: string – a string that will be matched to a datatype
deviceID: int – a unique number within a user's account that identifies a specific device associated with a user
userID: string – a unique string to identify a user

value: string – a string used to set a device's data. The string will be converted to the proper datatype.
Password: string – user's current password
newPassword: string – holds new password to replace old password

Methods:
setDeviceData(deviceID, datatype, value): void – sets device's data in database
getDeviceData(deviceID): object – gets device's data from database
authenticateUser(userID, password):
createAccount(userID, password):
removeAccount(userID): void – removes account in the database
changePassword(userID, newPassword): void – changes user's password in the database

LoginGUI

Attributes:
username: string – user enters username
password: string – user enters password to authenticate
isClicked: oolean – checks if button was clicked
failCount: int – records number of failed login attempts
errorMessage: string – error message to display to user

Methods:
verifyID(username, password): checks user input with database key-value pair
clickLogin(isClicked): function for when user clicks login button
getFailCount(): gets number of failed login attempts from database
setFailCount(failcount): sets number of failed login attempts in database
displayError(errorMessage): shows error message
createAccount(username, password): allows user to create a new account

MainInterfaceGUI

Attributes:
isClickedDoorLock: oolean – was DoorLock button clicked
isClickedLightControl: oolean – was LightControl button clicked
isClickedMusicControl: oolean – was MusicControl button clicked
isClickedAccountSettings: oolean – was AccountSettings button clicked

Methods:
clickDoorLock(isClickedDoorLock): reads oolean value and if true, door lock control
clickLightControl(isClickedLightControl): reads oolean value and if true, takes user to light control
clickMusicControl(isClickedMusicControl): reads oolean value and if true, takes user to music control
clickAccountSettings(isClickedAccountSettings): reads oolean value and if true, takes user to account settings

MusicControlGUI

Attributes:
volume: int – indicates volume level
isOn: oolean – indicates whether music is on or off
isPlaying: oolean – indicates whether music is currently playing
isConnected: oolean – indicates if music device is connected
currentSong: FILE- file name of current song
nextSong: FILE- file name of next song
previousSong: FILE – file name of previous song
playlist: FILE array – array of all songs

Methods:
play(): plays music on user's command
pause(): pauses music
setVolume(volume): uses volume int and sets volume to specified int
getVolume(): gets volume int
displayVolume(volume): displays volume level to the user

DoorLockGUI

Attributes:
isLocked: oolean – is the door locked

Methods:
getLockStatus(): gets lock status from the physical lock via communications from Arduino
updateLockStatus(isLocked): updates status after checking the isLocked value

LightControlGUI

Attributes:
status: Boolean – if device is on (1), if off (0)
brightness: int – how bright is the light
red: int – intensity of color red
green: int – intensity of color green
blue: int – intensity of color blue

Methods:
turnOn(): turns light on
oolean(): turns light off
getStatus(): gets status of light
adjustBrightness(brightness): uses brightness int to adjust to that specific brightness
changecolor(red, green, blue): change color based on intensity of each of the given variable values

AccessHistoryGUI

Attributes:
accessTimes: int- number of times user tries to gain access
accessFailCount: int – number of times user failed to gain access
isSuccess: oolean – TRUE if login attempt is a success. FALSE otherwise.

Methods:
openAccessLog(): shows access log
viewDayLog(): shows access log for current day
updateLog(accessTimes, accessFailCount, isSuccess): change/update the access log

AccountSettingsGUI

Attributes:
userList: list of users that are registered on the account

Methods:
addUser(userID): add a user to the account
removeUser(userID): remove a user from the account

DeviceController

Attributes:

isOnline: oolean – is device online?

deviceList: array – list of devices registered

deviceStatus: oolean – is device on or off?

deviceID: int – unique number (within user's account) used for device identification

Methods:

addDevice(deviceID): add a new device under users/application

removeDevice(deviceID): remove a current device

updateDeviceStatus(deviceID, deviceStatus): change device status from on to off or off to on

getDeviceStatus(deviceID): get the device status

requestConnection(deviceID): request to connect

Traceability Matrix

Classes/ Domain	Communication	Server	Device Interface Module	Arduino	Database Controller	Login GUI	Music Control GUI	Door Lock GUI	Light Control GUI	Access History GUI	Account Settings GUI	Device Controller	Main Interface GUI
R1	x	x				x			x				
R2	x	x				x			x				
R3	x	x			x								
R4		x			x								
R5	x	x			x						x	x	x
R6	x		x	x								x	x
R7	x		x	x			x						
R8	x		x	x			x						
R9	x		x	x			x					x	x
R10	x		x	x			x		x			x	x
R11	x		x	x			x		x				
R12	x		x	x			x		x			x	x
R13	x		x	x								x	x
R14	x		x	x								x	x
R15	x	x			x		x						
R16	x	x					x						x
R17	x	x			x								
R18													x

R19	x	x			x								
R20		x											
R21		x											
R22		x											
R23	x	x			x							x	
R24	x	x			x							x	
R25	x	x			x							x	x
R26													x
R27			x									x	
R28	x	x			x								
R29	x							x				x	
R30	x							x				x	

Table 2: Traceability Matrix (Classes to Domain)

Architectural Styles

Because autoHome has multiple functions and capabilities, each function is built upon different system architectures.

- The client-server model for our user interface and support for multiple users
- The database model for data collection and monitoring
- The event-driven model for triggering actions

The client-server model allows our system to have multiple users. The user's clients can all communicate with the central server. Each client will have a unique key that is used to authenticate itself with the server, granting them access to the client's hardware.

The database model allows to autoHome to collect and log data from the devices connected to the system. This data can be later access from a user account that has administrator privileges.

The event-driven model applies to autoHome's monitoring system. With an event-driven model, the specific actions that the sensors generate are not what trigger the system. The sensors first generate event notifications which are then sent to the server for processing. When the server receives an event notification, the server will process the notification, then take appropriate actions.

Identifying Subsystems

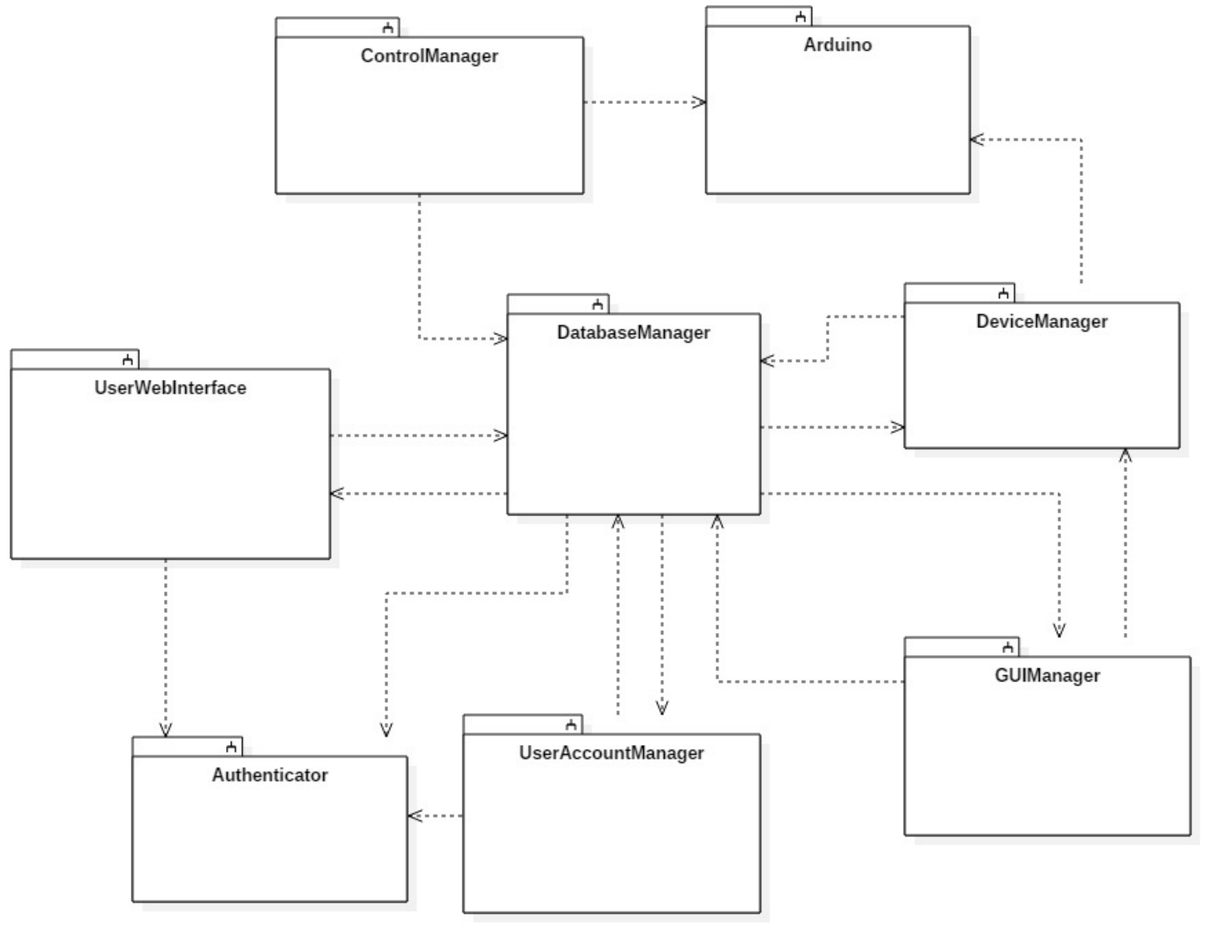


Figure 6: Package Diagram

Mapping Subsystems to Hardware

This system DOES need to run on multiple computers. The main application will be available to run as a web application (client), this includes subsystems: UserWebInterface, Authenticator, and UserAccountManager. A server will also be necessary to remotely execute commands for the user, this includes subsystems: DeviceManager, Arduino, and ControlManager. The server will also interact with the database accordingly and store all user data, login history, device statuses and whatnot, this is subsystem: DatabaseManager. We will also need a Arduino to be able to actually execute commands and effectively automate any persons home, thereby the Arduino needs to be connected with devices, hence why it is connected to subsystems: ControlManager, and DeviceManager.

Persistent Data Storage

Our system does need to save data that will outlive a single execution of the system. The main persistent data objects will be user account info. The user account info will contain everything from a user's email address, to his first and last name, his username and password, etc. Out software will also be keeping track of login attempts for security purposes, thereby a login log will be kept for every individual user. Every user will also have a set of recovery questions and answers to those questions, to be able to recover an account if the username/password is forgotten in some circumstances. Paired to every users account will be a set of devices that are paired to their home, this pairing information will need to be stored so that it is easy to reconnect and command devices in the future. Also, each device has its status readily available so that a user can remotely monitor them if needed. This will all be represented in the database schema below.

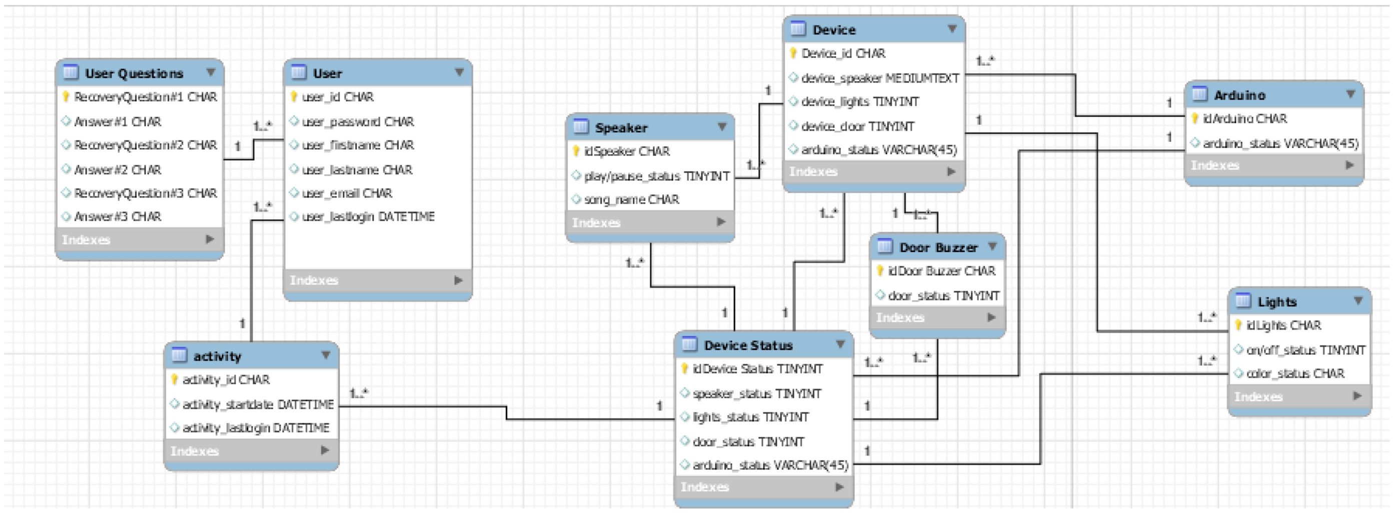


Figure 7: Database Schema

Network Protocol

For our project, we have 3 main components: The website/mobile app interface for user communications and input, an Arduino to communicate with hardware devices and servo motors, and a web server/ database to relay data. To be able to work with all of them we will need to use HTTP protocol to send and receive messages between the Arduino and our server. Furthermore, we will need to use Java RMI to communicate between our Arduino and other hardware i.e. door lock.

Global Control Flow

The autoHome system is an event driven system where actions are based from cause and effect type actions such as "turn on a light in this room" then the light in the room turns on. Also, an event doesn't just have to come because of a user-based action, an event can come from a timer such as when the door opens the buzzer beeps.
System time will be maintained by the computer clock which is also synchronized with the national atomic government time once a week. Accuracy of time is not crucial because the timing of events that rely on time such as turning off lights at night really do not need to be more accurate than to the nearest minute. Time is also backed up through use of a simple clock battery that can last around fifteen years.

autoHome is a real-time system however multiple actions can occur at any given time and the system will be able to prioritize by importance of the task and it can also process tasks in a queue like system. We will use thread algorithm for the server. The algorithm serves the non-functional requirement that the server should be able to communicate with multiple clients at the same time. The server keeps search for the connection requirement. If there is a requirement, the server will start a new thread to communicate with the client while the main thread is still searching for the requirement.

Hardware Requirements

Our server hardware does not need to be too sophisticated given that it will only be processing PHP, MySQL, AJAX, jQuery, and C++. To enhance system performance, the following specification is recommended:
Processor: Intel Pentium 4 2.0 GHz or better
Memory: 512MB or more
Storage: 50GB or larger
Broadband Internet Connection ≥ DSL 1.2 MBits/s or equivalent
Client will access the system interface via Android devices. Our Android software requires the following hardware specification:
OS: Google Android 2.1 or later
Processor: Qualcomm MSM8250 768MHz or better
Memory: 512MB or more
Storage 1GB or more
Display: HVGA 480 X 320 16-bit color or better

User Interface Design

Our initial user interface design, although appealing, lacked ease-of-use and was not entirely intuitive. Below is a side-by-side picture of our old design compared to our new design

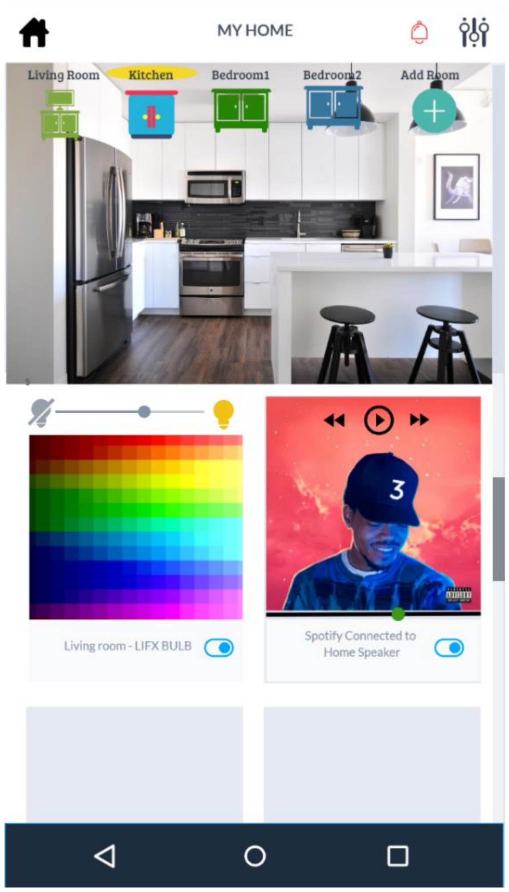


Figure 8: Old UI Design

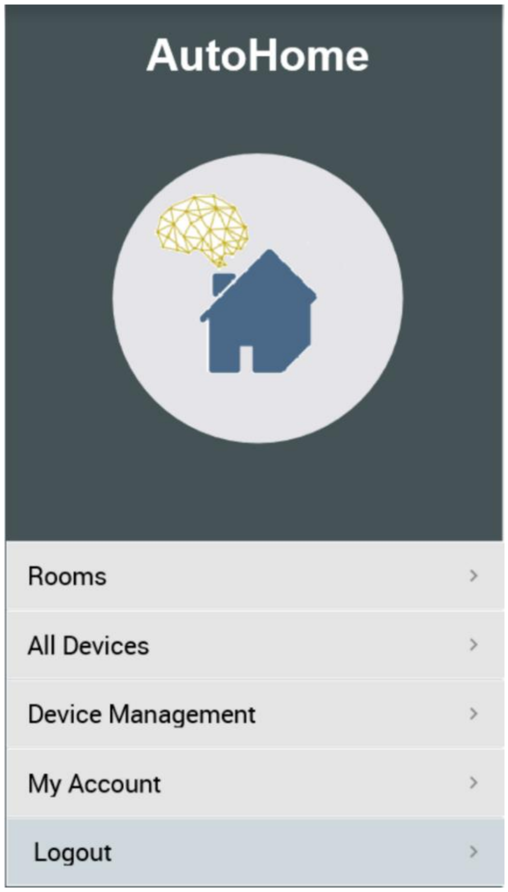
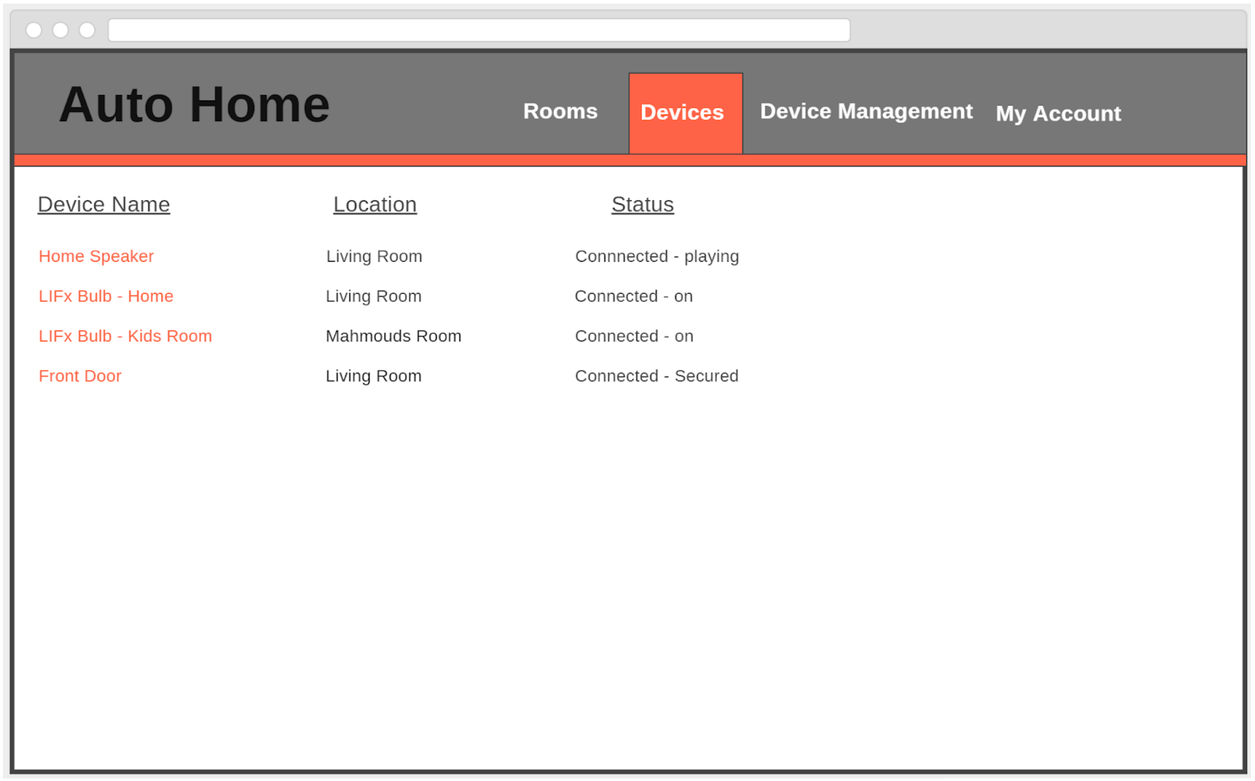


Figure 9: New UI Design

As you can tell from the new mockup of the home screen, it is greatly redone. Initially, our mockup was very flashy with lots of colors which may throw users off and confuse them with various different toggle bars and buttons on the page. The new design avoids all the flashy extra's which makes the interface a whole lot more user-friendly. The new design minimizes the amount of buttons on the screen which minimizes confusion. Although it may take more clicks to do the same thing on the first design, for example, changing lights color now requires users to do the following: All Devices >> LIFx Bulb >> change color, this new design is intuitive and doesn't require a lot of thinking to figure out where certain features are in the app as it is laid out in simple means.

In the new design, we still get all the same functionality as the previous design. Instead of having tabs for different rooms in the house, the new GUI home page has a button for Rooms where one can see the different rooms in the house and see which devices are connected and adjust the status of devices, i.e changing the song on the living room speaker. Additionally, the user can view all the devices connected to the network by clicking All Devices. This page allows for a second means of adjusting the status of devices. Device management is where users can add or remove devices as well as edit current ones. My Account holds the tools for editing user information and finally, Logout is used to log the user out of the system.

The webpage has a similar design to the app in that it is simplistic, yet user friendly and easy to use. The layout is similar with buttons that correspond to the buttons used in the App. Below is a mock up for our webpage.



Device Name	Location	Status
Home Speaker	Living Room	Connnected - playing
LIFx Bulb - Home	Living Room	Connected - on
LIFx Bulb - Kids Room	Mahmouds Room	Connected - on
Front Door	Living Room	Connected - Secured

Figure 10: Web Client Mockup UI

For the second iteration of our application interface, we decided to hide the more complicated functions from the user. For example, the control of the RGB LEDs had the color changer next to the brightness adjustment and on/off switch. After some deliberation, we decided that the user will not change the color of the LEDs that often and moved it into a separate sub-menu to reduce clutter. So even though it takes more actions to get to the sub-menu, the overall feel of the app is much improved. We took this design approach of moving less used function into sub-menus to reduce clutter of the main pages.

The design philosophy we took while designing our app was to reduce clutter and the number of buttons on the screen at one time while maintaining the core functionality. We moved options and less used functions into drop down menus. We decided it was okay it was okay to increase the amount of button presses to get to some functions.

Design of Tests

The home automation system that we are building has a few central components with several supplementary features. The central software components include the code for the WiFi module, the user-web interface, the database, the server and the communication between all components of the system. These parts are essential because they are interdependent. The failure of any one of these features causes the whole system to fail. Supplementary features include light control, music control, monitoring of door locks, and system control using a mobile application. These features are supplementary because they can fail without affecting other parts of the home automation system. It is important to keep in mind that even though we call these features supplementary, this does not mean these features are unimportant.

Following this distinction, the way we design the acceptance tests for our home automation system follows two phases of “Bottom Up” integration testing. In the first phase, we perform unit tests on the central components. Once the system passes the unit tests, individual components are put into small groups and tested together. We continue combining groups until we test each central component as a whole and finally perform integration testing as we test the components together. After the central components have passed the integration test phase, we begin the second phase. This phase begins with unit tests on the supplementary features. Following the same procedure that was used to test the central components, these supplementary features are tested from the bottom upwards. After each supplementary feature has passed its own “Bottom Up” testing, it is tested with the system of central components in integration testing. Although it was previously stated that the second phase of testing starts after the first phase, we can perform the unit tests for both phases concurrently.

The section below shows plans for unit tests of each class shown in the class diagram. We start by isolating individual functions in a class for testing.

DeviceController

addDevice(deviceID); removeDevice(deviceID)

Success:

- The user sends a request to add or remove a device to his or her account, a device ID unique within the user's account is passed to the function and the function sends the request to the server via communications. In the database, the device is added to or removed from the account.

Failure:

- The user sends a request to add or remove a device from his account and the request is not received by the server.
- The user sends a request to add or remove a device from his account and the device ID generated is not unique within the user's account.
- The user sends a request to add or remove a device from his account, server receives the request, but the device ID is not stored in the database.

updateDeviceStatus(deviceID, deviceStatus)

Success:

- The device controller sends a message to the server via communications. This message asks the database to update the status of the specific user's device.

Failure:

- Some or all of the message sent is lost during communications.
- The message is sent and received, but the wrong user account is updated.
- The message is sent and received, but the wrong device is updated.
- The message is not sent.

getDeviceStatus(deviceID)

Success:

- The device controller gets the status of a device from the Arduino via data from communications.

Failure:

- The device controller gets the correct device, but the wrong status.
- The device controller gets the status of the wrong device.
- The device controller gets a null status for a device that exists in the user's account.
- The device controller gets a status for a device that does not exist in the user's account.

requestConnection(deviceID)

Success:

- The device controller receives a connection request from the Arduino via the device interface module. The controller then sends the request to server via communications and waits for a boolean reply. The function returns the boolean reply.

Failure:

- The controller receives a connection request, but does not send it.
- The controller sends a connection request when no request was received.
- The controller receives and sends a connection request to server, but never receives a reply and is stuck waiting.
- The reply is not boolean.

DeviceInterfaceModule

turnOn()

Success:

- The user presses a button on the device interface to turn on a specific device. The button changes appearance to indicate to the user its current status. The user’s input is sent to the Arduino and the device corresponding to the button pressed turns on.

Failure:

- The user presses a button to turn on a specific device and the signal is not sent to the Arduino.
- The user presses a button to turn on a specific device and the signal is sent to the Arduino, but the device does not turn on.
- The user presses a button to turn on a specific device and the signal is sent to the Arduino, but the wrong device turns on.
- The button does not change appearance after being pressed.

turnOff()

Success:

- The user presses a button on the device interface to turn off a specific device. The button changes appearance to indicate to the user its current status. The user’s input is sent to the Arduino and the device corresponding to the button pressed turns on.

Failure:

- The user presses a button to turn off a specific device and the signal is not sent to the Arduino.
- The user presses a button to turn off a specific device and the signal is sent to the Arduino, but the device does not turn off.
- The user presses a button to turn off a specific device and the signal is sent to the Arduino, but the wrong device turns off.
- The button does not change appearance after being pressed.

<u>getData()</u>

Success:

- The device interface module retrieves data from the Arduino.

Failure:

- The Arduino sends data, but the device interface module cannot or does not retrieve the data.
- The Arduino does not send data, but the device interface module retrieves data that is not null from the Arduino.
- The Arduino has data and the device interface module can retrieve data from the Arduino, but the data is corrupted.

<u>sendData(data)</u>

Success:

- The device interface module sends data to the Arduino.

Failure:

- The device interface module sends data, but the Arduino cannot or does not receive the data.
- The device interface module does not send data, but the Arduino receives data.
- The device interface module sends data and the Arduino receives data, but this data gets corrupted.

<u>getMessage()</u>

Success:

- The device interface module gets a message from a queue of messages sent by the Arduino. This message may be code, a signal or an error message.

Failure:

- The message retrieved is corrupted.
- The device interface module gets a message with content when the message queue is empty.
- The device interface module does not get a message when there are messages in the message queue.

<u>sendMessage(message, deviceID)</u>

Success:

- The device interface module sends a message to a queue of messages for the Arduino. This message should only be code.

Failure:

- The message sent is not code.
- The message is not queued.
- The message is not sent.

<u>setDigital()</u>

Success:

- The device interface module sets a digital value as a parameter for the Arduino. This digital value is used to change a device's output.

Failure:

- The device interface module does not pass the digital value to the Arduino.
- The device interface module passes the digital value to the Arduino, but the device's output does not change.
- The device interface module passes the digital value to the Arduino and the device's output changes, but not in the correct manner.
- The device interface module passes anything that is not a digital value to the Arduino.

<u>requestConnection(deviceID)</u>

Success:

- The device interface module passes a connection request from the Arduino to the device controller.

Failure:

- The request from the device interface module is not sent to the device controller even though a request was sent by the Arduino.
- The request is sent, but not received.
- The request is sent with the wrong device ID.
- The request is sent to the Arduino.
- A request is sent to the device controller when no request was sent from the Arduino.

AccountSettingsGUI

<u>addUser(userID)</u>

Success:

- Input the userID to add a user to the home automation system. User is successfully added

Failure:

- userID is not recognized
- userID is not in string form
    - Both of these cases would not allow the method to proceed, and would lead to failure as a result
- userID is recognized but user is not showing up in the system

<u>removeUser(userID)</u>

Success:

- Input the userID to remove a user from the home automation system. User is successfully removed

Failure:

- userID is not recognized
- userID is not in string form
    - Both of these cases would not allow the method to proceed, and would lead to failure as a result
- userID is recognized but user still remains in the system

AccessHistoryGUI

<u>openAccessLog()</u>

Success:

- Shows access log when button is clicked

Failure:

- Button is clicked but access log is not shown

- Access log is shown on the screen when button hasn't been clicked

<u>viewDayLog()</u>

Success:
- Shows access log for the current day when button is clicked

Failure:
- Button is clicked but access log is not shown
- Access log is shown on the screen when button hasn't been clicked
- Access log is shown for an incorrect day

<u>updateLog(accessTimes, accessFailCount, isSuccess)</u>

Success:
- Update the log as more accesses/attempts to access occur

Failure:
- One or more of the input variables has an incorrect value, therefore the updated log is incorrect
    - This can happen if an access or a failed access is not recorded, thus the variable would not have been updated

DoorLockGUI

<u>getLockStatus():</u>

Success:
- Gets lock status from the physical lock via communications from Arduino

Failure:
- Gets incorrect lock status
- Fails to get lock status from arduino

<u>updateLockStatus(isLocked)</u>:

Success:
- updates status after checking the isLocked value

Failure:
- LockStatus is updated to an incorrect value because of value isLocked
- isLocked is read incorrectly thus updateLockStatus is incorrect
- isLocked is not read, therefore lock status is not updated

LightControlGUI

<u>turnOn()</u>:

Success:
- User clicks button to turn light on, and light is turned on

Failure:

- User clicks button and light doesn't turn on
- User doesn't click button and light turns on

turnOff():

Success:

- User clicks button to turn light off, and light turns off

Failure:

- User clicks button and light stays on
- User doesn't click button and light turns off

getStatus(): gets status of light

Success:

- Gets the current status of light and correctly shows the status to the user
  - If light is on, the user is told that the light is on
  - If light is off, user is informed that light is off

Failure:

- Displays incorrect status of the light

adjustBrightness(brightness): uses brightness int to adjust to that specific brightness

Success:

- Takes int brightness and sets the brightness to that specified number.

Failure:

- Misreads int brightness and displays incorrect brightness
- Doesn't read int brightness and doesnt change brightness

changecolor(red, green, blue): change color based on intensity of each of the given variable values

Success:

- Reads values of variables red, green, and blue and adjusts color based on the intensity of each color

Failure:

- Fails to read or incorrectly reads one or more of the input colors, thus displaying an incorrect color
- Doesnt read one or more int and doesn't display the colors
- Displays color when the color values haven't been specified

MusicControlGUI

play(): plays music on user’s command

Success:

- system reads user’s request to play music → music plays
- System plays song that user selects → correct song is played
- Song is selected from an array of files

Failure:

- System doesn’t read the user’s request to play music → music doesnt play when user requests it
- System plays a song that user doesn’t select → incorrect song is played
- System plays music when user doesn’t request it

pause(): pauses music

__Success:

- system reads user’s request to pause music → music is paused

Failure:

- System doesn’t read the user’s request to pause music → music doesnt pause when user requests it
- System pauses a song when user doesn’t request → song is paused unwantedly/unexpectedly

setVolume(volume):

__Success:

- reads the int volume and sets the volume to the specified integer

Failure:

- Receives int volume and doesn’t set volume to specified int
- Fails to read volume int, volume doesn’t change

getVolume(): gets volume int

__Success:

- Gets volume int successfully, and gets the correct value.

Failure:

- Doesn’t receive volume int
- Receives incorrect volume int
- Receives volume int when it is not sent

displayVolume(volume):

__Success:

- Uses input variable volume and displays the specified number

Failure:

- Displays incorrect number
- Doesn’t display number when inputted
- Displays a number when a number hasn’t been inputted

MainInterfaceGUI

clickDoorLock(isClickedDoorLock):

Success:

- DoorLock button clicked and signal was recognized → user is taken to door lock control

Failure:

- Button clicked but not read → user is not taken to door lock control
- Button not clicked and is incorrectly read → user is taken to door lock control as an error

clickLightControl(isClickedLightControl):

Success:

- LightControl button is clicked and signal is recognized → user taken to light control

Failure:

- Button clicked but not read → user is not taken to light control
- Button not clicked and is incorrectly read → user is taken to light control as an error

clickMusicControl(isClickedMusicControl):

Success:

- MusicControl button is clicked and signal is recognized → user is taken to music control

Failure:

- Button clicked but not read
- Button not clicked and is incorrectly read

clickAccountSettings(isClickedAccountSettings):

Success:

- AccountSettings button is clicked and signal is recognized → user is taken to account settings

Failure:

- Button clicked but not read → user is taken to account settings as an error
- Button not clicked and is incorrectly read → user is not taken to account settings

LoginGUI

<u>verifyID(username, password)</u>

Success:

- The user enters in his username and password. The LoginGUI passes this information to function where it verifies and matches every character to its corresponding table in the database, if both username and password matches, the User is able to login

Failure:

- The user enters in his username in password. The LoginGUI passes this information to the function where it verifies and matches every character to its corresponding table in the database, if just one character does not match, the user will not be able to login and it will be recorded as a fail attempt.

<u>clickLogin(isClicked)</u>

Success:

- The user clicks on the 'Login' button, the LoginGUI should recognize this, and upon verification of the ID, will allow the user to move to the next page

Failure:

- The user clicks on the 'Login' button, the LoginGUI should recognize this, if the ID is not verified successfully, the user will not be allowed to move and a fail count will be recorded
- The user clicks on the 'Login' button, the LoginGUI does not recognize this.

<u>getFailCount()</u>

Success:

- The LoginGUI will retrieve the number of failed login attempts from the database

Failure:

- The LoginGUI will attempt to retrieve the number of failed login attempts from the database, and the failcount is unable to be retrieved
- The LoginGUI cannot communicate with the database

<u>setFailCount(FailCount)</u>

Success:

- The LoginGUI will increase the fail count by one everytime the verifyID function fails

Failure:

- The LoginGUI will fail to increment the fail count by one even if the verifyID function failed.

displayError(errorMessage)

Success:

- The LoginGUI will cout/print a error message to the user depending on the type of failure

Failure:

- The LoginGUI will fail to print out a message to alert the user, and instead, nothing will be displayed

createAccount(username, password)

Success:

- The LoginGUI will send the new username and password to the database where it will be stored, then a message will be displayed to the user to relay that account creation was successful

Failure:

- The LoginGUI is unable to communicate to the database and is unable to store the new username and password
- The LoginGUI sends the new username and password to the database but the database is unable to store the new user account.

Arduino

requestConnection()

Success:

- Arduino requests connection to the main server, and accepts signals to control the connected systems after a successful connection.

Failure:

- Connection is unsuccessful due to server being offline.
- Connection is unsuccessful due to the Arduino not having internet connection

listenReply()

Success:

- Arduino gets reply from server and successful acts upon it.

Failure:

- Arduino does not get a reply due to no connection to the internet
- Arduino does not get a reply due to the listening period timing out.

sendMessage(message, deviceID)

Success:

- Arduino sends a command to the specified device and device performs the action

Failure:

- The message is invalid and the signal is meant for a different device
- The deviceID is invalid and is not a connected device

getMessage()

Success:

- Arduino receives device's message

Failure:

- Device is offline and no message is received by the Arduino

sendData(data)

Success:

- Arduino sends device data to server

Failure:

- Arduino does not have internet connection and data is not sent to the server
- Server is not online and data is not able to receive data

getData(deviceID)

Success:

- Arduino gets device status and data from the specified deviceID

Failure:

- The specified deviceID is invalid and no data is received
- Specified device is not currently connected and no data is received

chooseAction(actionList)

Success:

- Arduino receives a valid action and sends it to the device

Failure:

- Inputted actionLIst is not valid and no action is sent to the device

Communications

<u>getData(username, data)</u>

Success

- The device interface module retrieves data from the Arduino as requested by the logged in user

Failure

- The Arduino sends data, but the device interface module cannot or does not retrieve the data.
- The Arduino does not send data, but the device interface module retrieves data that is not null from the Arduino.
- The Arduino has data and the device interface module can retrieve data from the Arduino, but the data is corrupted.

<u>sendMessage(message)</u>

Success:

- The device interface module sends a message to a queue of messages for the Arduino. This message should only be code.

Failure:

- The message sent is not code.
- The message is not queued.
- The message is not sent.

<u>requestConnection(connectionRequest)</u>

Success:

- The device interface module passes a connection request from the Arduino to the device controller.

Failure:

- The request from the device interface module is not sent to the device controller even though a request was sent by the Arduino.
- The request is sent, but not received.
- The request is sent with the wrong device ID.
- The request is sent to the Arduino.

- A request is sent to the device controller when no request was sent from the Arduino.

<u>sendRequestReply(requestReply, connectionID)</u>

Success

- Reply is send upon successful completion of the request along with the connection ID requesting the reply.

Failure

- The request is sent, but not received.
- The request is sent with the wrong device ID.

Server

<u>listenConnectRequest(connectionRequest)</u>

Success:

- Server receives request from Arduino to establish connection

Failure:

- Server receives signal, but connection

<u>respondConnectRequest(connectionResponse)</u>

Success

- After server receives message from arduino, the server sends “connectionResponse” message to the arduino
- Message successfully received by arduino.

Failure

- “connectionResponse” message sent by server but not received by arduino
- Connection unsuccessful.

<u>getData(username, deviceID)</u>

Success

- Data is sent successfully to the arduino from the server for the given username and device ID

Failure

- Username is not found in database.

- Device ID is not found in the database.

<u>setData(data, datatype, userID, deviceID)</u>

Success

- Data is sent successfully from the arduino to the server.
- Data is saved under the Device ID and userID

Failure

- Data is sent but not received by the server
- Data is being sent with wrong device ID or userID

<u>getmessage()</u>

Success

- Request sent and message successfully received.

Failure

- Request sent but message not received.

<u>setmessage(message)</u>

Success

- Message sent and set.

Failure

- Message sent but not successfully received.

<u>authenticate(username, password)</u>

Success

- Username matches with the password and user is confirmed.

Failure

- Username and password are not a match, authentication failed

<u>invalidate(errorMessage)</u>

Success

- Error message sent and received by device.
- User receives “username or password incorrect” error message

Failure

- Error message sent but not received by device.

<u>accountLock(username)</u>

Success

- Account with username specified status switched to locked

Failure

- Lock request sent but not locked
- Username specified does not exist

<u>createAccount(username, password)</u>

Success

- Request sent by the user and is received successfully by the server
- New user is created in the database with username as password as specified

Failure

- Request sent by the user but not received by the server
- Request sent by user but the username specified already exists in the server
- Request is sent by user with a valid username but password does not meet requirements.

<u>deleteAccount(username)</u>

Success

- Request to delete is sent by user with valid username
- User with username specified is deleted from server database

Failure

- Request to delete is sent by user but not received by the server
- Username specified by user does not exist

<u>updateUsers(username, userID)</u>

Success

- Request to update userID is sent by user and received.
- User with userID specified is updated to new ID also specified by user

Failure

- Request is sent to update userID but not received
- New userID already exists

<u>updateStatus(deviceID)</u>

Success

- Request sent to update status and perform an action
- Device with device ID receives the status update message

Failure

- Request sent to update status but message not received
- deviceID specified does not exist

<u>updateConnectionStatus(deviceID)</u>

Success

- Request sent by user to update connection status for given deviceID
- Device with device ID receives the connection status update message

Failure

- Request sent by user to update connection status but message not received
- deviceID specified does not exist

<u>sendAlert(userID)</u>

Success

- Request sent to user for Alert
- User with userID receives the alert message

Failure

- Request sent but message not received by user specified
- userID specified does not exist in server

DatabaseController

<u>setDeviceData(deviceID, datatype, value)</u>

Success:

- The DatabaseController is able to successfully set the data type for each individual device. For example, RGB values for lights range from 0-255 and have a datatype of int. The door buzzer would be a boolean datatype for locked/open, etc.

Failure:

- The DatabaseController is unable to set datatypes and their values for different devices
- The DatabaseController assigns incorrect datatypes and values for devices.

<u>getDeviceData(deviceID)</u>

Success:

- The DatabaseController is able to successfully retrieve the data values for each device (correct data types, etc.)

Failure:

- The DatabaseController is unable to retrieve the data values for a device
- The DatabaseController retrieves data values for a device, but is an incorrect value, or an incorrect datatype.

<u>authenticateUser(userID, password)</u>

Success:

- The LoginGUI sends over a login attempt with parameters for a user and password. First the DatabaseController will search for the username, if there is a match, it well then compare the password entered to the password stored in the database, if this is a match as well, then authentication succeeds.

Failure:

- The LoginGUI sends over a login attempt with parameters for a username and password. The DatabaseController searches for the username but is unable to find one that matches in the database
- The LoginGUI sends over a login attempt with parameters for a username and password. The DatabaseController searches for the username and there is a match, unfortunately the password doesn't match so authentication fails.

<u>createAccount(userID, password)</u>

Success:

- The LoginGUI sends over new user account data to be stored into the database, the DatabaseController takes the argument parameters and successfully inserts it into the database.

Failure:

- The LoginGUI sends over new user account data, but unfortunately the database is unable to insert it into the database
- The LoginGUI is unable to communicate with DatabaseController in order to send over new user account data

<u>removeAccount(userID)</u>

Success:

- The LoginGUI sends over a request to delete a user account (after authenticateUser succeeds). The DatabaseController then searches for this username, if found, it will delete the user account info from the database

Failure:

- The LoginGUI sends over a request to delete a user account. The DatabaseController searches for this username but is unable to find it
- The LoginGUI sends over a request to delete a user account. The DatabaseController succesfully locates this user account but is unable to delete it from the database.

<u>changePassword(userID, newPassword)</u>

Success:

- After the user is authenticated, the LoginGUI sends over a request to change the password. The DatabaseController searches for the username, if the username is found, then the password is successfully updated.

Failure:

- The user is not authenticated.
- The user is authenticated but the DatabaseController is unable to find the username in the database
- The user is authenticated and the DatabaseController is able to find the username, however it is unable to update the password.

Project Management

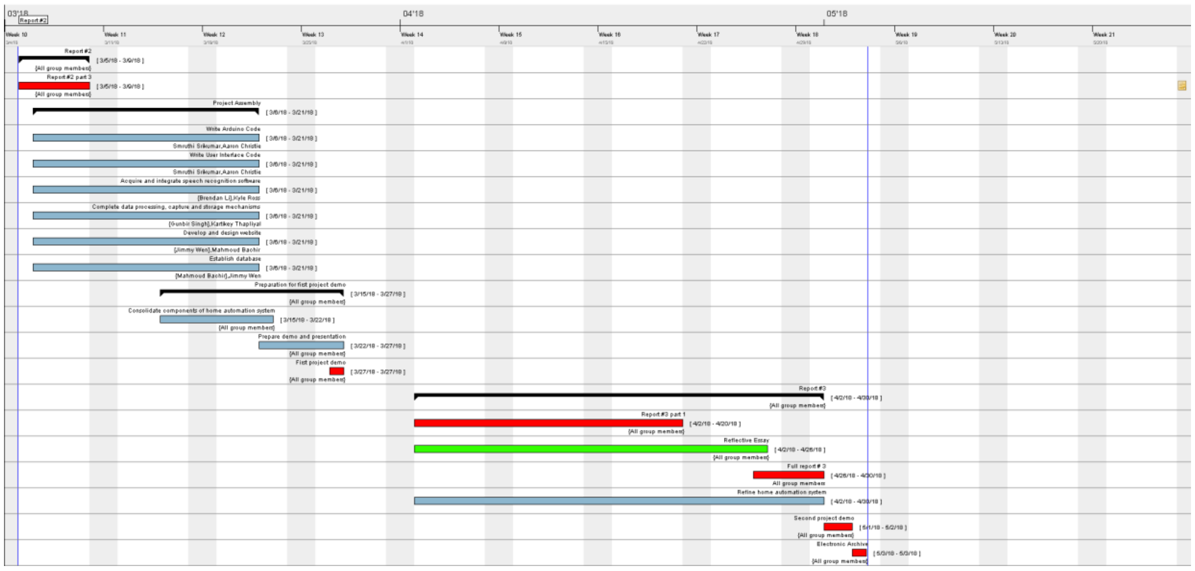


Figure 11: Gantt char for Plan of Work

Most, if not all, Arduino functions have successfully been written and are fully functional. We have successfully created a mobile app GUI but have yet to create a web client. The database has begun to be developed but it is still in its early stages as most of us have to learn SQL in greater depth.

By next week we should have the Web Client developed and close to fully functional. The database will also be worked on, and set up, we hope to have the majority of it done by next week. We also want to begin making sure that subsystems can effectively communicate with each other.

References

1] Home automation - https://www.vectorsecurity.com/UserFiles/File/PDF/blog/Smart-Home-Interactive.pdf

[2] Arduino microcontroller - https://www.arduino.cc/, https://www.elegoo.com/download/, https://www.linuxjournal.com/content/learning-program-arduino

[3] Version control system - https://github.com/

[4] Software Engineering book - http://eceweb1.rutgers.edu/~marsic/books/SE/book-SE_marsic.pdf

[5] Android development - https://medium.com/google-developers/developing-for-android-introduction-5345b451567c,http://www.ebookfrenzy.com/pdf_previews/AndroidStudioEssentialsPreview.pdf

[6] Smart home sensors - https://www.smarthomeusa.com/plan-your-own-smart-home-system/,https://www.networkworld.com/article/2925722/security0/home-security-demystified-how-to-build-a-smart-diy-system.html