**Smart Light Controller Simulation**

**by**

**Collin Davis, crdavis2**

**Report submitted on 6 April 2019**

**EE 333 – Engineering Programming Using Objects**

**Department of Electrical and Computer Engineering**

**The University of Alabama at Birmingham**

**ABSTRACT**

TBD

**Table of Contents**

**Introduction………………………………………………………………………………………1**

**Project Definition………………………………………………………………………………...1**

Constraints………………………………………………………………………………………...2

Goals…………………………………………………………………………………………….2-3

Features…………………………………………………………………………………………....3

Applicable Standards……………………………………………………………………………...3

**Design……………………………………………………………………………………………..4**

Initial Design Process…………………………………………………………………………...4-5

Appropriate for Object Oriented Approach……………………………………………………….5

Design Decisions………………………………………………………………………………..5-6

Object Oriented Design…………………………………….…………………………………...6-9

**Discovery and Use of Online Information…………………………………………………….10**

**Debug……………………………………………………………………………………………10**

**Results…………………………………………………………………………………………...10**

**Discussion……………………………………………………………………………………….10**

**Conclusions……………………………………………………………………………………...10**

**References……………………………………………………………………………………….10**

**Appendix………………………………………………………………………………………...10**

**INTRODUCTION**

This project’s main purpose is to allow students to show their ability to design and implement a program using object orientation. This type of programming allows for the creation and definition of objects based on their characteristics. For example, a class that models cars could be used to create subclasses that are specific car types or models. With object-oriented programming though, a fair amount of time must be spent in designing the program in order to achieve the desired result with the program. The object-oriented design approach involves careful consideration of what the program will and will not do as well as creation of diagrams (these diagrams have design standards) in order to better layout the plans for the program before the coding begins.

For this project, the object-oriented language Java is used. The project is focused around the creation of a program that models a smart light controller like those made by companies such as Philips. Provided with a simple definition of the program, careful analysis of the problem is essential to the creation of an efficient and effective program. To achieve this, the problem must be broken down into its key components to better understand the requirements and how these requirements can be designed into an object-oriented program.

**PROJECT DEFINITION**

The program created for this project is a smart light controller that allows for control of light bulbs within a space such as a house, apartment, or business. To achieve this, the Java language must be used to produce a program with solid object-oriented design. The program is to be modelled after some examples of smart light controllers that can be found in the real-world today. Examples of smart light controllers in the real-world allow the user to control various states of the light bulbs such as on/off status, color, and brightness of individual bulbs as well as groups of bulbs defined by the user. The following sets of constraints, goals, features, and standards have to be taken into consideration while the program is being designed.

***Constraints***

* CO-01: The controller shall be programmed in the Java language using NetBeans.
* CO-02: The controller shall be designed using an object-oriented design approach.
* CO-03: Modelling classes shall not contain I/O methods.
* CO-04: The controller shall be modelled after a real-world example.

***Goals***

* GO-01: The controller should demonstrate all features required by the assignment.
* GO-02: The controller should be in accordance with design standards.
* GO-03: The controller should be user-friendly.

***Features (in MoSCoW list format)***

* Must-01: Allow the user to add/remove a bulb to the system.
* Must-02: Allow the user to enter a color by name for a bulb.
* Must-03: Allow the user to set groups of bulbs.
* Must-04: Allow the user to control bulbs individually or as a group.
* Must-05: Allow the user to set the brightness of a bulb/group of bulbs.
* Should-01: Allow the user to define events that cause bulbs to perform actions defined by the user.
* Should-02: Create an ID for each bulb in case it is removed, then added back later.
* Could-01: Have an “emergency” or “alert” state that overrides current state of all bulbs to a predefined state.
* Could-02: Automatically create a “master” group that includes all bulbs as long as one bulb exists.
* Could-03: Allow the user to create timers for groups of lights/allow bulbs to turn on/off at specific times of day.
* Could-04: Allow the user to create “mood” presets that can apply to any set of bulbs.
* Could-05: Allow the user to name individual bulbs.
* Won’t-01: Allow the user to enter incompatible values for color, brightness, etc.
* Won’t-02: Leave new bulbs’ characteristics blank until set by user.

***Applicable Standards***

* Standard-01: All methods and classes will comply will Javadoc documentation standards.
* Standard-02: All code should comply with documentation, formatting, and structure standards for EE333.
* Standard-03: The program will be continuously tested and debugged to guarantee completion of the program to its specified degrees.

**DESIGN**

***Initial Design Process***

In order to effectively design a program, the problem/task needs to be understood to its fullest degree. In this case, there were real-life examples of what the program should accomplish. This allowed for a more open-ended set of requirements for the project. Of course, this also allowed for a variety of features that could be implemented, or not, based on the importance of the feature as determined by the designer. However, the general approach for causing an event within the smart light controller all started from the same place, like the real-world examples show. Since each event that happens in the system is caused by the user, it was most important to allow the user to be able to perform actions in a simple manner to accomplish the user’s desires. The process by which an event happens in the controller can be viewed as a narrative, like many object-oriented programs can be. This general process is as follows: user inputs a command, controller passes the command to the hub, hub processes the command and does the action on the bulb(s) involved.

Based on this general sequence of events that the program must follow, some objects that are required are evident. It is apparent that controller, hub, and bulb objects must be created as objects for the program. Based on how the real-world models work, a controller could model a cell phone that communicates with the hub or even the user’s voice which is heard by the hub. While the controller is what starts the narrative sequence of events, the bulb is the main object in the system. Bulbs were created with the standard color of white, maximum brightness, are turned on, and are not assigned to a group in order to incorporate the main features of a bulb when it is added to the system. The bulb was designed in a way that allowed for expansion of features if desired. The controller and hub were designed so that they would be able to send messages to all the other objects in the program. Each bulb will have its own address which will allow the hub to communicate actions to it specifically. This allows for easier flow of communication within the system.

These three objects were made into classes. The controller acted as an input device for the user to perform commands within the program. All three of the classes were coded with the ability to set and get the values for the various features of the bulbs. While this may seem redundant, this allowed the controller to send a message to the hub that then processed and relayed the message to the specified bulb/bulbs with a command. Thus, each object (class) could “see” the status of the various characteristics of each bulb. Although the three classes are very similar, the hub and controller had methods that were unique to them. For example, if the user wanted to add a bulb to a group, this message would be passed from the controller to the hub, where the hub would check to see if the specified group exists, create it if not, then add the group tag to the appropriate bulb.

Rest of Design TBD After Implementation

***Appropriate for Object Oriented Approach***

For a problem to be appropriate for an object-oriented approach, the problem must be able to be pieced into various, individual parts that could be seen as objects in the solution. These parts must be able to have states that can be changed and manipulated based on changes to the system at hand. For this project, it was easy to see that the problem could be broken down into three main objects: the bulb, the hub, and the controller. Each of these components was able to hold states as well as change states in some form, just like the real-world examples. The bulbs were the main component whose states were changed frequently, but the hub and controller were constantly update as the state of the bulb(s) changed. While the system was relatively simple to understand, it did lend well to the object-oriented design approach.

***Design Decisions***

AL-01 How should the user assign color to bulbs?

AL-01A User inputs RGB values for the bulb

AL-01B User puts the name of a color which gives the bulb that color’s preset RGB values

AL001C User could input either alternative A or B

ALD-01 Alternative A was chosen. For user-friendliness, the user should not have to enter three numbers in order to set the color. This would also require at leat minor understanding of how the RGB values function.

AL-02 What happens after the “alert” state ends?

AL-02A All bulbs are returned to pre-alert setting

AL-02B All bulbs are set to on with 70% brightness

ALD-02 TBD

AL-03 What happens if a bulb is removed from the system then added back?

AL-03A The bulb is treated like a new bulb

AL-03B The hub “remembers” the bulb and assigns its former values to it.

ALD-03 Alternative A was chosen here. It seemed difficult to assess if a new bulb with the same name of a previously added, then removed, bulb was desired to have the same setting as the previous, or if it is supposed to be a new bulb. Also, a record of any bulbs that have ever been added to the system would have to be remembered, at least for a specified period of time, which could potentially cause memory issues with larger systems. Thus, the user would simply have to reconfigure the bulb to achieve the settings it had prior to being removed.

AL-04 What happens if a desired state (on/off) is the same as the current state?

AL-04A Hub checks the bulb and returns a statement such as “Bulb already on” (or off)

AL-04B The hub still passes the command to the bulb, where nothing will change

ALD-04 Alternative A was chosen so that if the user attempts to turn a bulb on that is known to be off, yet the hub has the bulb set to “on”, the user would realize that the bulb itself may be broken.

***Object Oriented Design***

The UML class diagram for the project is seen in Figure 1 below. As described in the design portion of the project report, the main three object classes are very similar to one another. They all have access to the same set of basic commands that could be performed by the user, as indicated by the Commands superclass. This allows for standardization of these features of the program as well as simpler writing of the Bulb, Hub, and Controller classes. This also eliminates the writing of lots of redundant code between the classes. Since the “emergency” feature overrides the current values of all the bulbs, the controller, hub, and bulb versions of the method are very different from one another, which is why they are not inherited from the superclass.

The two interaction diagrams, Figure 2 and Figure 3, demonstrate two features of the program. Figure 2 depicts how the emergency setting is applied to the bulbs. This process involves an input from the user to either turn the setting on or off, which the hub checks against the current state (whether an emergency is already occurring or if the bulbs are in a normal state) and communicates the appropriate message to all the bulbs. Figure 3 depicts the process by which a bulb is added to a group. The hub allows the user to input a group that may not already exist, at which point the hub creates the group, then adds the specified bulb to the group.

Figure 4 is a state diagram that shows the process by which a bulb is turned on or off. As with all the other methods/processes, the controller, hub, and bulb are involved. If the current state of the bulb is the same as the desired state the user input, then the bulb’s state will not change, but how this is handled was included as a design issue.

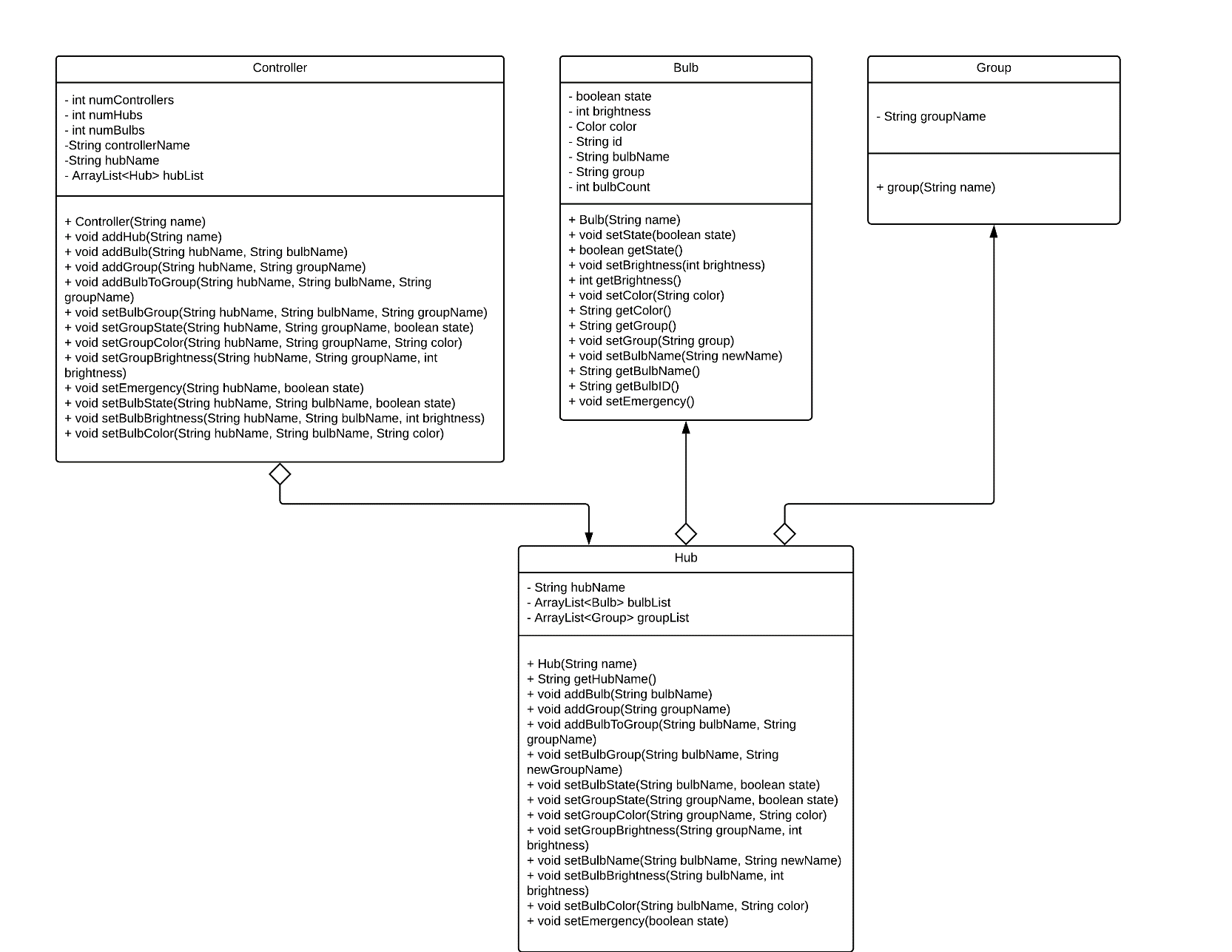


Fig 1: Smart Light Controller UML Class Diagram

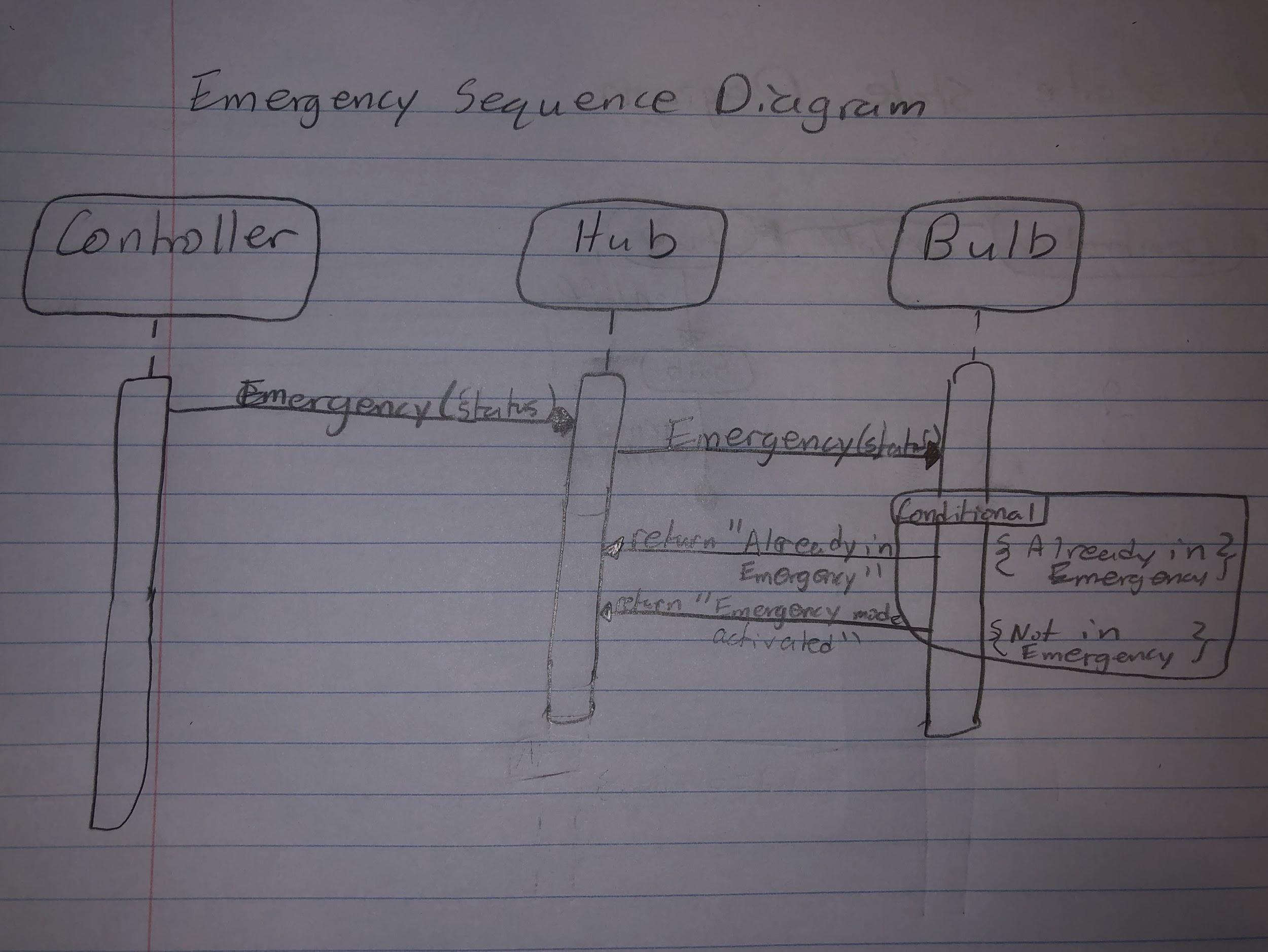


Fig 2: Emergency Sequence Diagram

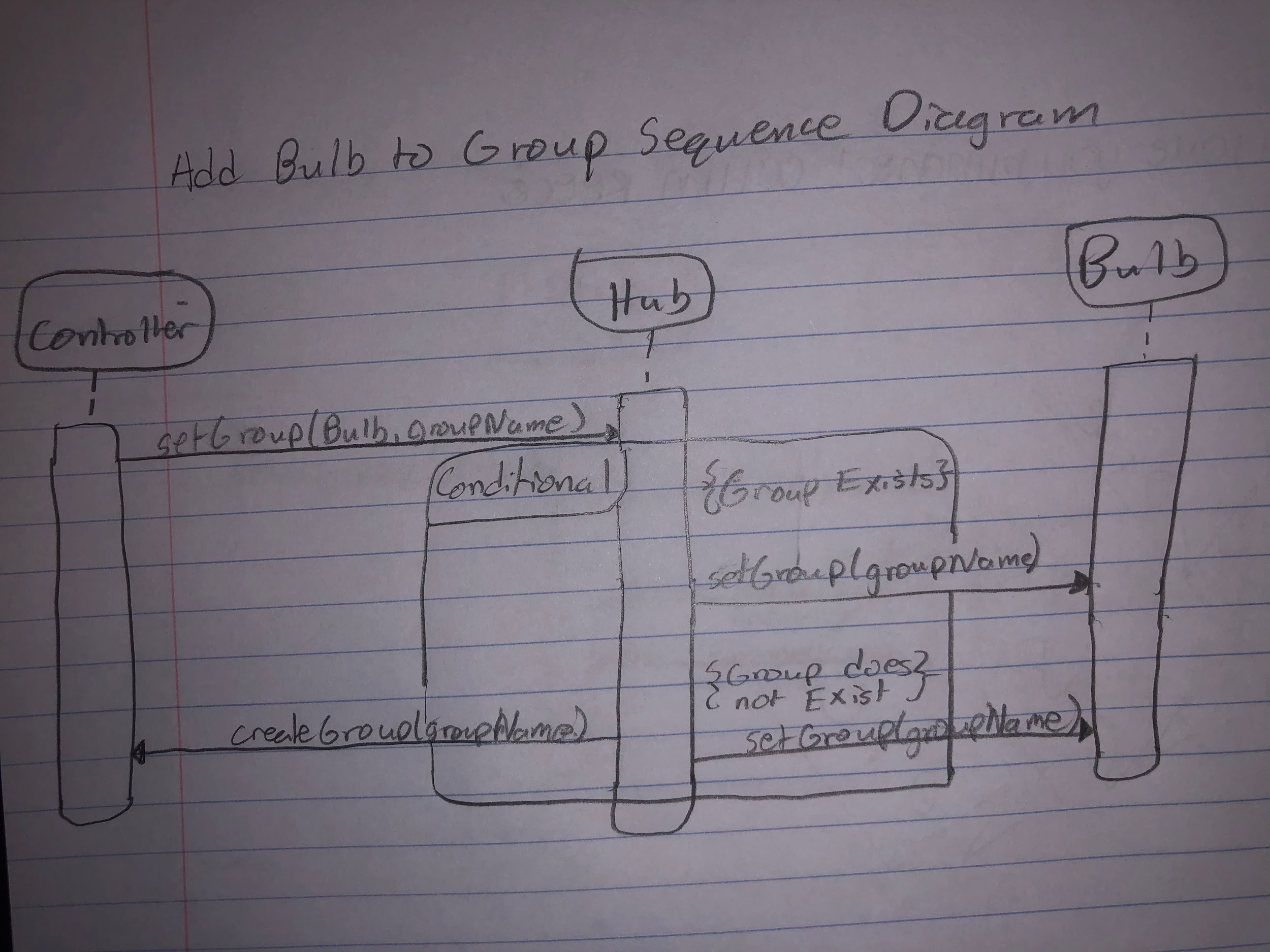


Fig 3: Add Bulb to Group Sequence Diagram

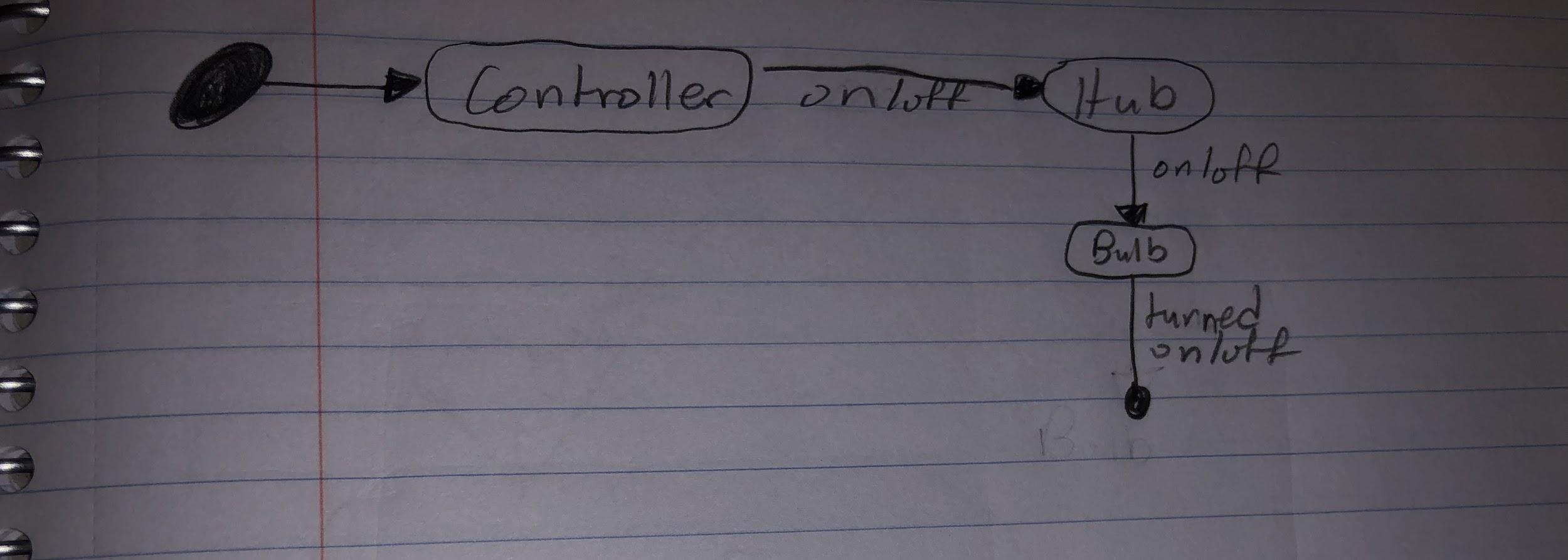


Fig 3: Bulb State State Diagram

**DISCOVERY AND USE OF ONLINE INFORMATION**

There was no use of online code, packages, or the like. However, the LucidChart website was used in order to create the class diagrams for this project.

**DEBUG**

While there will not be evidence of the errors this caused, one major issue arose when attempting to pass commands between the controller, hub, and bulbs. The problems came with CoolBeans suggesting that certain methods (like setBulbColor for the controller and hub) should be made static, yet CoolBeans was not capable of suggesting more than that. I attempted to implement static variables and methods where CoolBeans suggested, but this caused counting errors and improper passing of commands between the objects. This was resolved by rewriting the means by which the controller “told” a hub what to do. The use of ArrayLists was very important here, since the hubs and bulbs needed were stored as objects in ArrayLists, at which point they could be called with the format “bulbList.get(j).setState(state)” for example. Here, the hub finds the correct bulb from the list, then calls its setState method to set it to what the state argument holds.

**RESULTS**

TBD

**DISCUSSION**

TBD

**CONCLUSIONS**

TBD

**REFERENCES**

TBD

**APPENDIX**

TBD