OOP - Object-Oriented Plants

Kayden Adams, Ann Marie Mahon, John Henry Fitzgerald

Final State of the System

Were Implemented:

- PlantPots are simulated with their own individual water sensors and lighting system.
- Code for sensors for humidity level and temperature. These sensors interface with a hose and an AC/heating system respectively. The water sensor will automatically water the plant until the humidity reaches the desired level. The water sensor also simulates water evaporating over time. The temperature sensor will begin heating or cooling the room if the temperature is below or above a certain threshold, and then stop when it gets to the median temperature between the two ranges. It has the ability to simulate external temperature changes when the AC and heater are off (such as a cold winter day cooling down the room)
- Lights are hooked up to individual plants. These lights are set to be on for a certain number of hours a day and will turn on and off as hours pass.
- Water reservoirs are connected to multiple hoses that go to plants. The water sensors in those plants determine when the hoses should draw from the reservoirs when their plant needs water. The reservoirs will alert the user when they are low on water and need to be refilled.
- There is a basic database to store information when the program is not running. When the program is started, it will inform the user of what happened while they were away by simulating a certain number of hours.
- There are individual users, each with their own rooms, and those rooms have their own plants and reservoirs. Users log in upon entering the app and then can access their information and see the state of their system. These users can create, change, or remove rooms, plants, and reservoirs. Multiple users can also own the same room.
- There is a mocked out connection to an API for plant information. Currently the connection is just simulated by connecting to a CSV file, but could be extended to communicate with an API in the future. When the user creates a plant, if the app has data on the plant it will auto-fill those fields. Otherwise, it will prompt the user for data such as how many hours of light the plant needs. This will also provide a warning if the user is putting the plant in a room with a non-optimal temperature (for example, trying to put a tropical plant into a room that stays between 50 and 60 degrees)
- A comprehensive text-based UI to allow the user to do all of the tasks required to maintain their garden, including adding, deleting, and renaming rooms, plants and reservoirs, as well as refilling reservoirs. The UI is also in charge of interfacing with the database, API, and user.

Were Not Implemented:

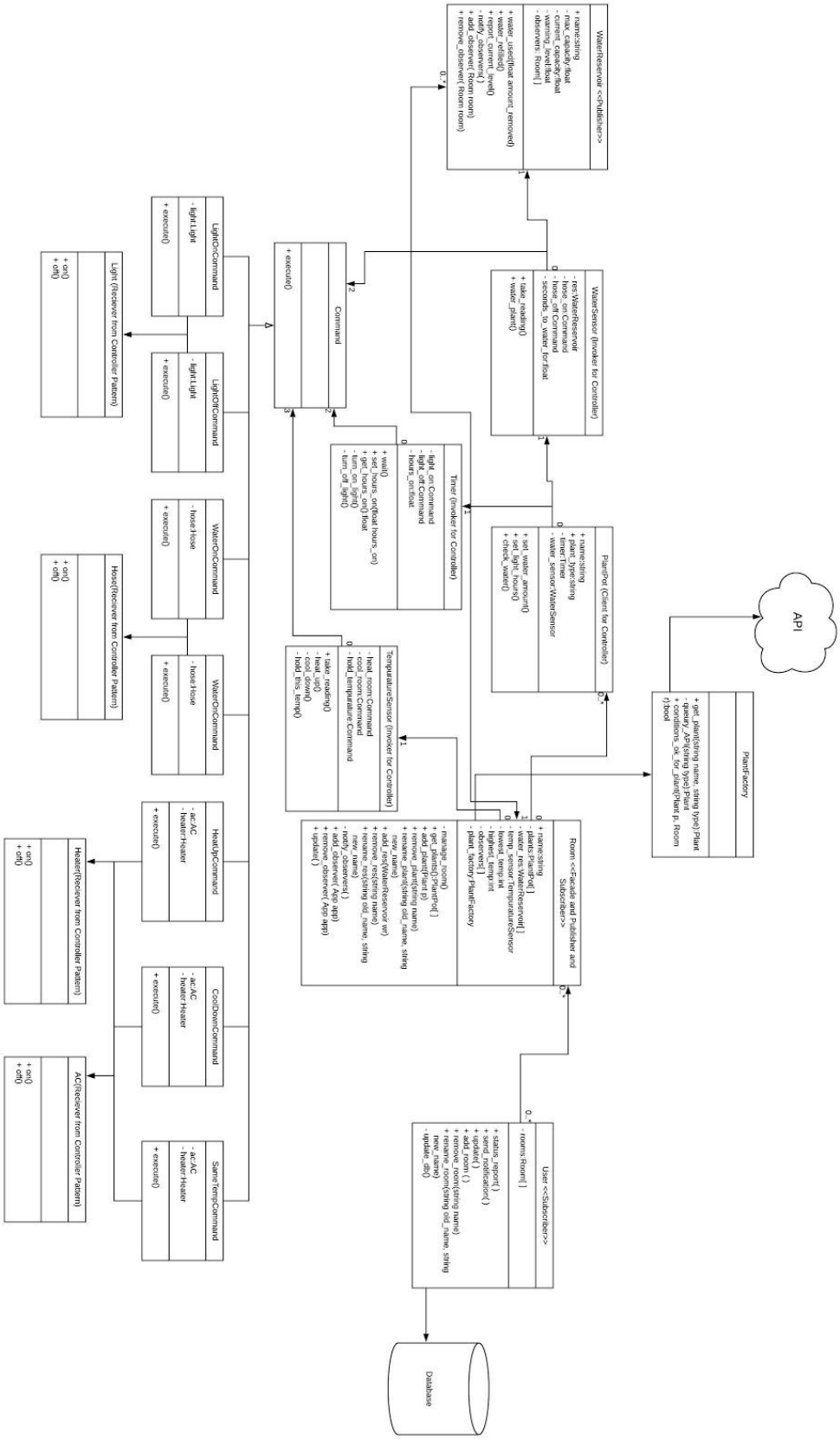
- A GUI. We determined that our time would be much better spent on functionality and gaining a better understanding of OOD principles and patterns, rather than trying to create a time-consuming GUI.

- Multiple users at once. This would require some sort of hosting rather than just a local system, and a significantly more complicated system for data storage than what we have currently.
- A fully connected API. While a connected API would be really cool and helpful, it would have also been much more difficult to connect than we had originally thought. We simply ran out of time for the complications and additional workload that getting, converting, checking, and storing data from an API would have created.
- An external database software. Everything is set up as it would be in a database such as postgreSQL, with separate tables, primary keys and foriegn keys, and joins tables. We just didn't get around to switching it to an external database.

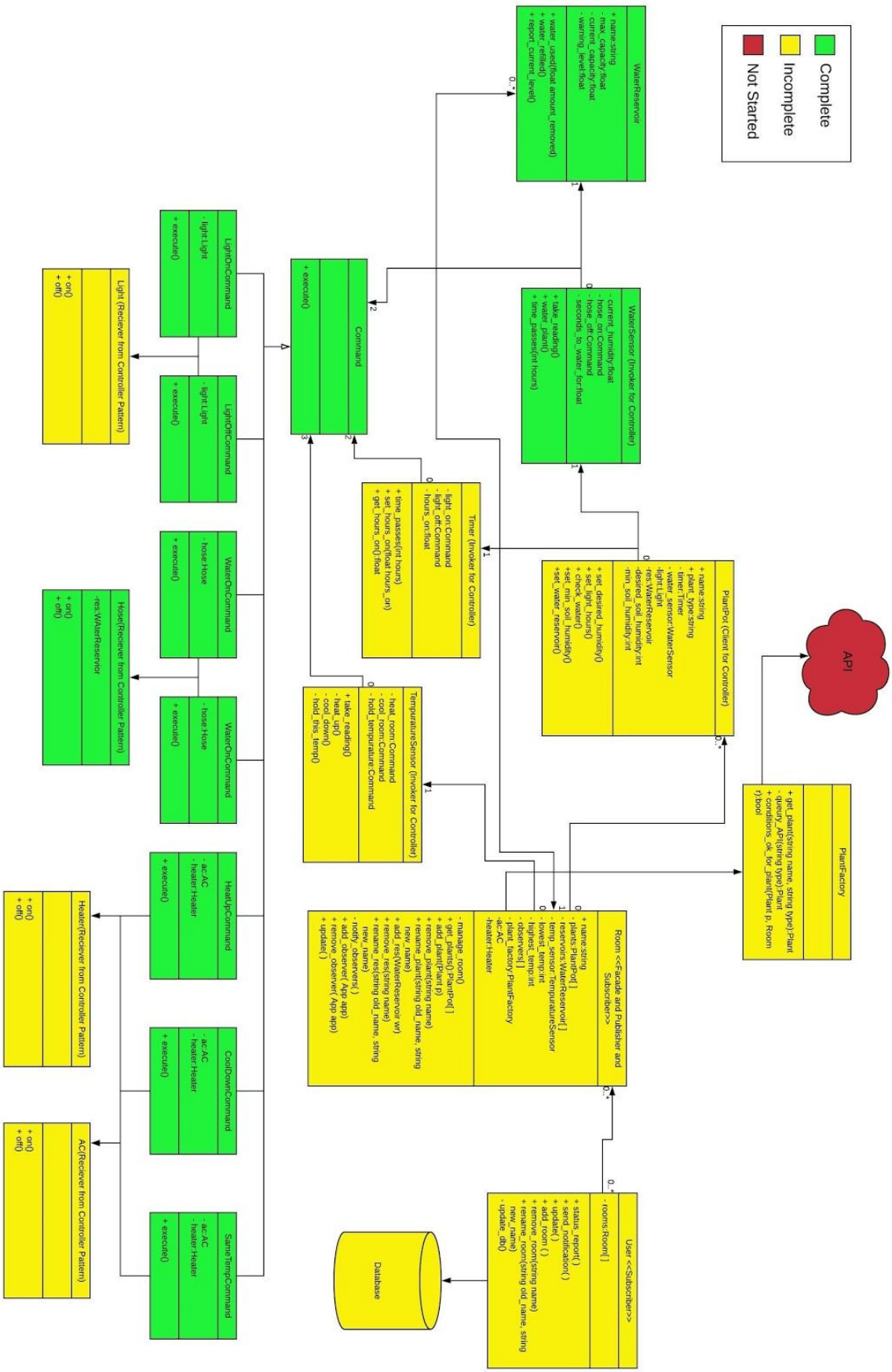
Class Diagrams

As per usual, the class diagrams have links to the Lucidchart file they're in since they are so large as to be basically unreadable at this size. In the final class diagram, blue represents a class that is part of a OO pattern.

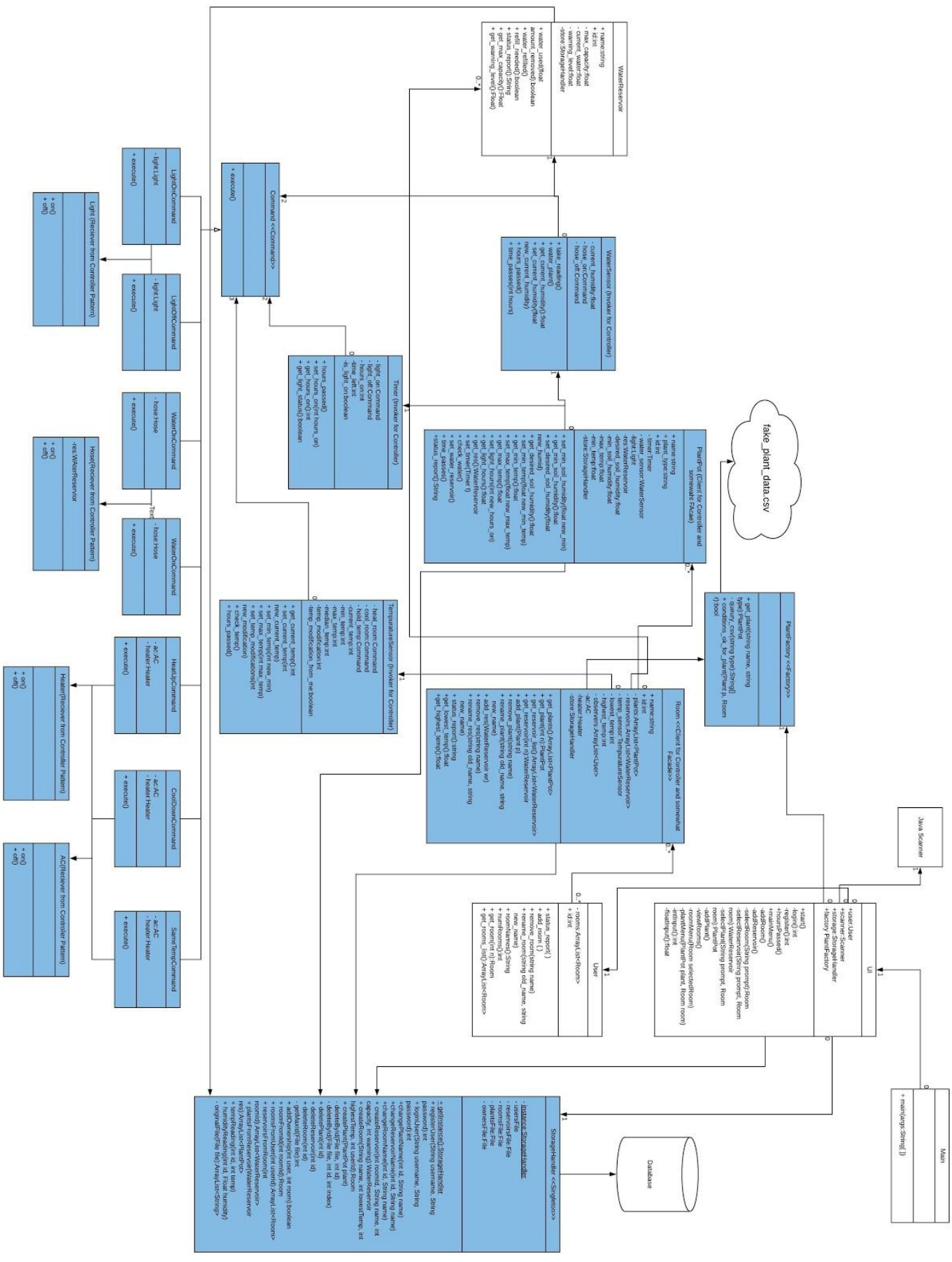
Project 4 Class Diagram



Project 5 Class Diagram



Final Class Diagram



Comparison Statement

The observer pattern for reservoir/room and room.user was removed. As we progressed, it no longer made sense in our design, and we decided it made more sense to have a “HAS-A” relationship than use the observer pattern.

We now have an object for database interactions called StorageHandler. This makes way more sense than having the user interacting with the database directly.

We have a text-based UI as a class now, whereas before we were thinking about using a GUI of some sort.

The Trefle API we were planning on using was replaced with a CSV file for a mock API, with the PlantFactory object taking care of using the mock API to create plants.

Third-Party code vs. Original code Statement

The vast majority of the code was fully original. A lot of the code we used to help us with patterns came from the lecture slides for this class. The resources we used are the following:

Code examples for using ArrayList from:
https://www.w3schools.com/java/java_arraylist.asp

Reference for java io https://www.tutorialspoint.com/java/java_files_io.htm

Singleton reference https://www.geeksforgeeks.org/singleton-class-java/

Reference for reading and writing to files:
https://tutoref.com/how-to-read-and-write-files-in-java-8/

Reference for how to take terminal input:
https://alvinalexander.com/java/edu/pj/pj010005/

Helpful for fixing terminal input:
http://chronicles.blog.ryanrampersad.com/2011/03/text-based-menu-in-java/

No code was directly taken whole-sale from another website, aside from the structure of singleton, the lines of code to initialize the scanner, and the code for reading a file which were taken from https://www.geeksforgeeks.org/singleton-class-java/, https://alvinalexander.com/java/edu/pj/pj010005/, and https://tutoref.com/how-to-read-and-write-files-in-java-8/ respectively.

Statement on the OOAD process for your overall Semester Project

1. We spent a great deal of time creating the class diagram and thinking through how all of the pieces would fit together. We tried to follow OO principles. We used encapsulation, in making many variables private. We used abstraction in having each object only responsible for its own things and only responsible for things that made sense to us. We used patterns when they made sense to us, but when we noticed we were trying to force a pattern where there didn't need to be one, we changed to something that made more sense.
2. We didn't do very much testing or refactoring during the design, up until the end. We did some informal unit testing as we went along, but the bulk of the testing came towards the end of the process. Going forward, we would start doing unit and integration tests earlier. However, waiting until later to refactor worked well. It was more important to us to get functionality and then get the code looking and flowing better.
3. Through the design process, we learned about cutting scope. We kept core features and the central concept but made implementation details easier. For example, the postgreSQL database we had planned became csv files, but the structure was the same as the database structure would have been, and the same functionality was present. We couldn't create a full API reader, but there's still a text-file to represent the API and an object in place to query it. We didn't do a GUI, but we still created a UI, just a text-based one. We feel we learned how to set more realistic scoping goals, but we also learned a lot about how to cut scope in ways that don't ruin the thing you're trying to build.