Milestone 3 Report

Engicoders
H. Bird, B. Karacelik, J. Peters, J. Ropotar, A. Rybka
Department of Computer Science, University of British Columbia
COSC310: Software Engineering
Dr. Gema Rodriguez-Perez
March 11, 2024

Sequence Diagrams

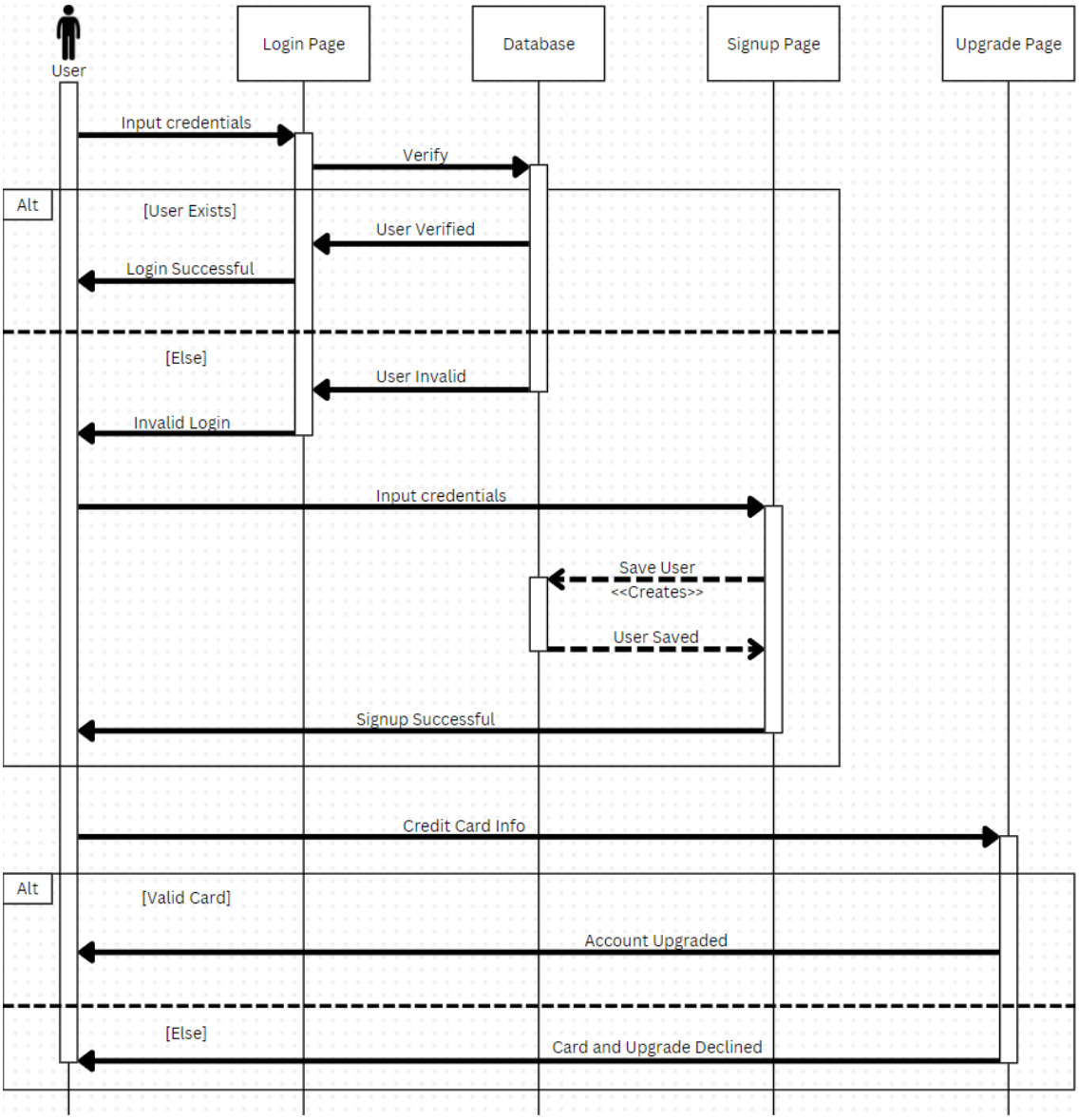

Figure 1. User Login/Signup/Upgrade Account Sequence Diagram

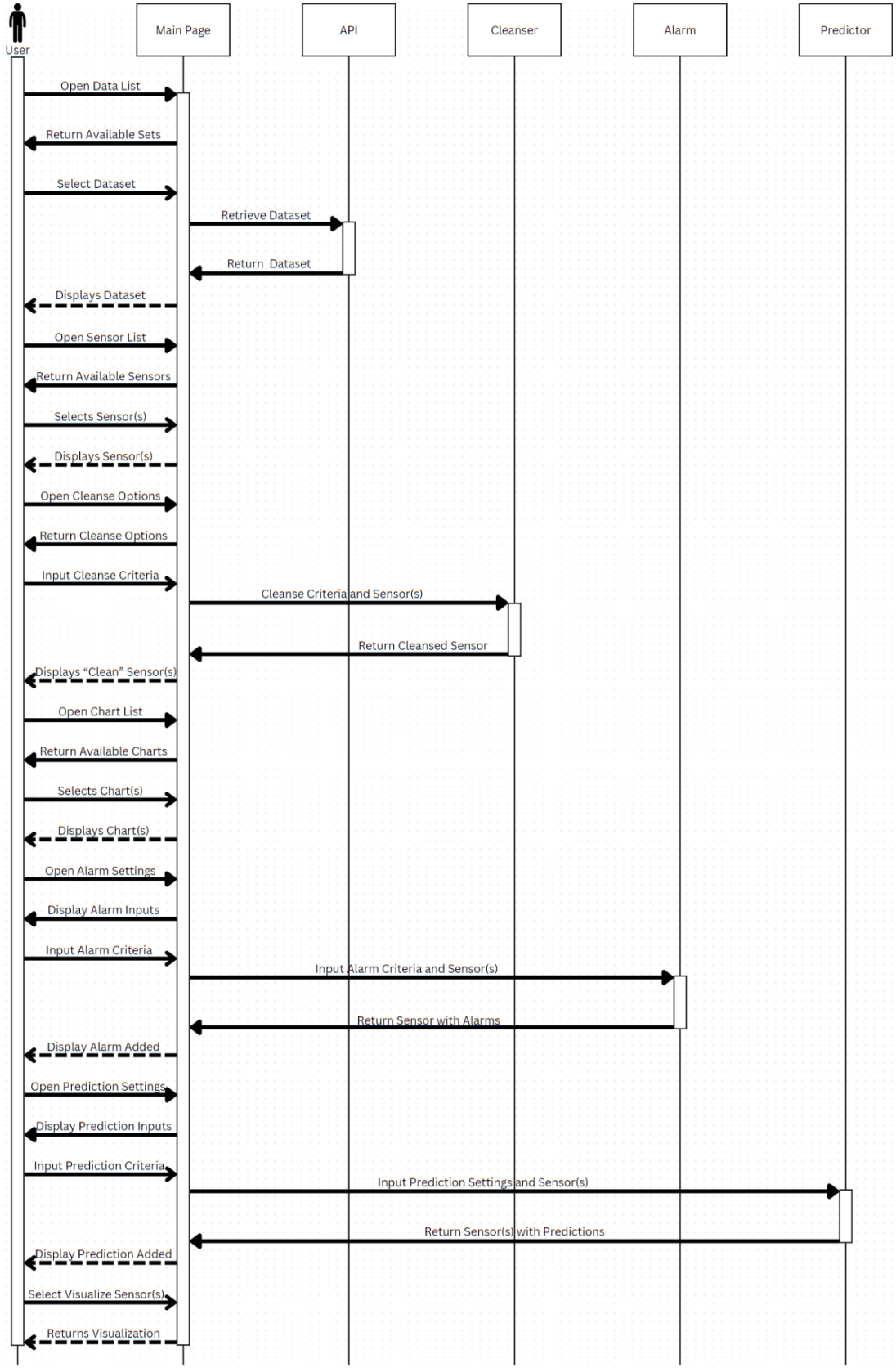

Figure 2. User Main Page Sequence Diagram

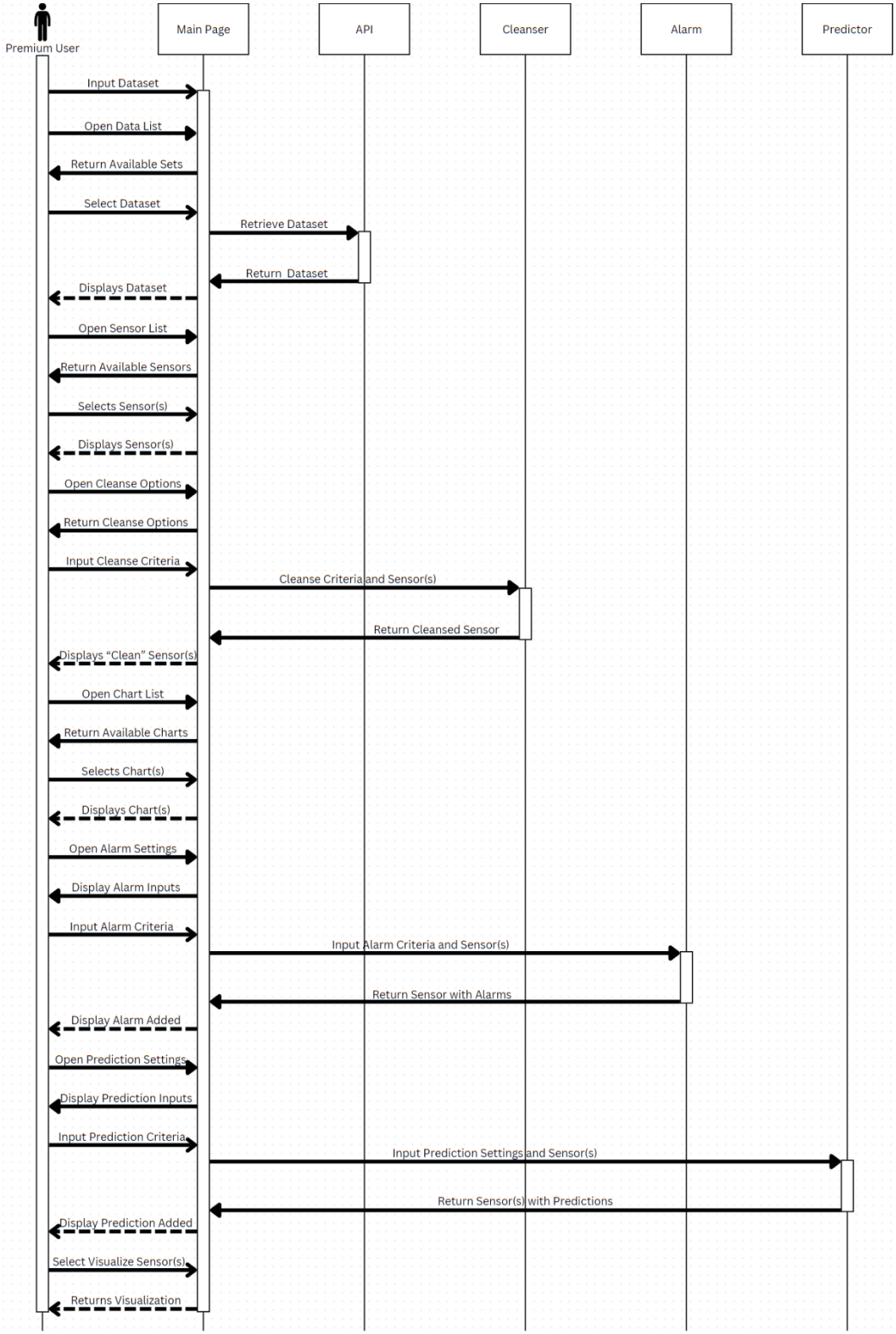

Figure 3. Premium User Main Page Sequence Diagram

Shown above in Figure 1-3 are the sequence diagrams of our project. To start, Figure 1 is the typical login page sequence. The user must input their username and password which will be verified against the database. If the user exists, they will gain access to the main page. If the user does not exist, they will be prompted to input their username and password into the sign-up screen which will then save their credentials into the database for future use. The new (or existing) user can then choose to upgrade the account, which will prompt them to input their credit card info. If the credit card is valid, the account will be upgraded, if not the account will remain unchanged.

Figure 2 illustrates the typical main page sequence of a common user. Once on the main page (after logging in), the user will open a drop-down list of available datasets and once open, they will select one to use. This will then get the dataset from the api and show the user it has been selected. Next, the user will open a list of sensors within the dataset and select the ones they want to use. If the user wants to cleanse the sensor(s), they will open the cleanse option and input their requirements which will send the sensor and requirements to the cleanser and return a message with the new sensor object. The user will then open up the available charts and select which one they want to use. If the user wants alarms, they will open up the alarm settings, input the specifications they want for the alarms, and send the data. The sensor will then be returned with the added alarm. If users want to add a prediction to the alarm, they can open the prediction tab and input their prediction requirements. Once input, the data will be sent to the predictor and the sensor with the predictions will be returned. Finally, the user can select to visualize the data they manipulated.

Figure 3 shows the typical sequence diagram for premium users on the main page. This diagram is the same as Figure 2 for the common user and the main page, but it allows the premium user to input their own dataset before selecting datasets. This allows the premium user to use the software on their data along with the data from the API.

Class Diagrams

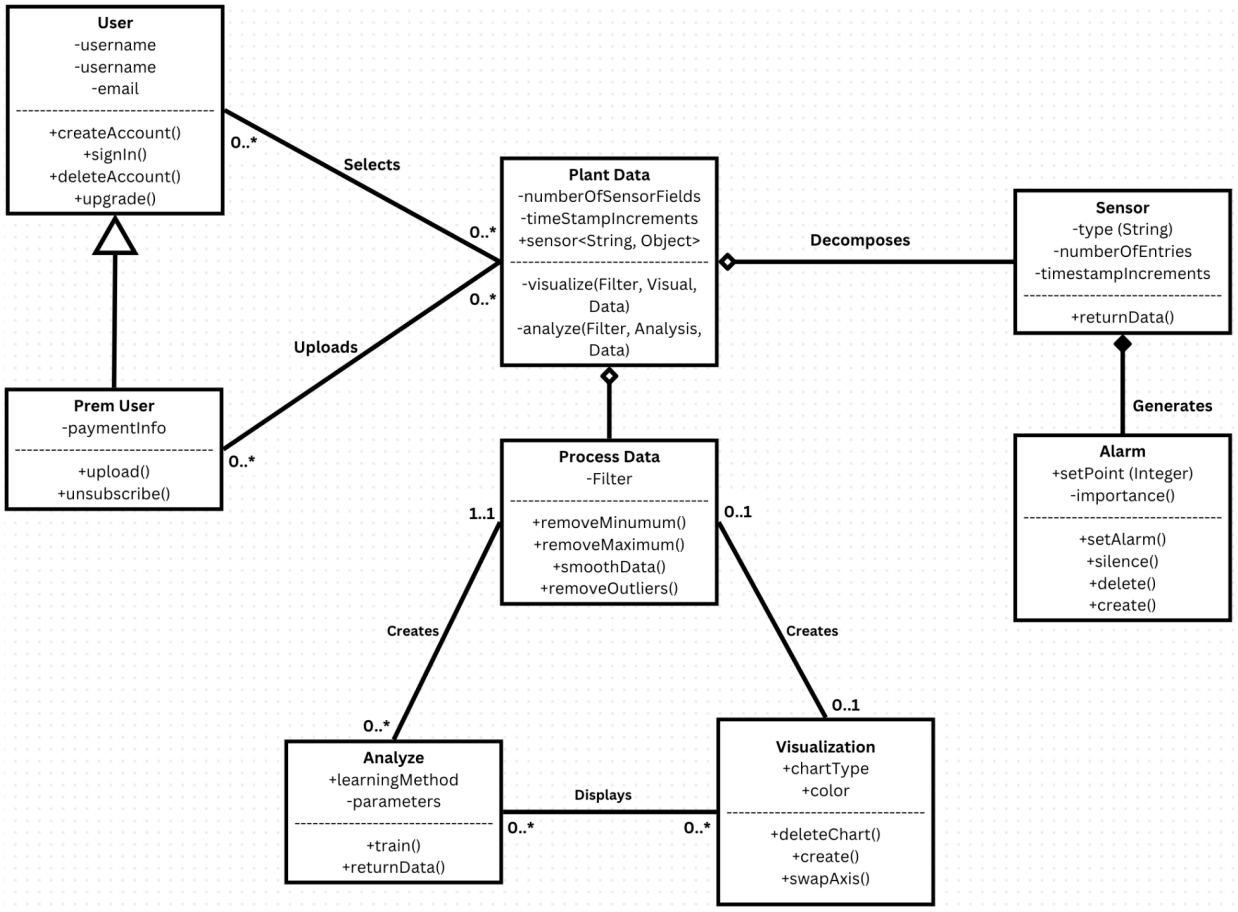

Figure 4. Class Diagram

Our class diagram, shown above in Figure 4, illustrates how the different entities of our project will associate with each other as well as their general attributes and methods. To start, the main entity in our project will be the two types of users, common and premium users. The premium user is an extension of the regular user, with the primary difference being that the premium user has to have payment information, and gains the ability to upload data. Users can select from many plant data sets, and each plant data is not necessarily fixed to any one user. The plant data has a set of sensors, specific to each specific plant data instance. Each sensor can have alarms, which are triggered by specific inputs. Plant data is also associated with process data, which takes in plant data to filter. It contains the filtered set of data, which can be passed to visualization or analyze. Each set of processed data can be associated with many different analysis methods, as well as a specific visualization of the data. The analysis can be visualized as well, with each analysis returning multiple potential sets to visualize.

Test Plan

For our project, we have implemented a white box testing strategy. This strategy is further broken into three procedures to ensure our code works. To start, we have a continuous integration pipeline which goes through all of the code in pull requests before they are merged into the main branch. This pipeline checks and ensures there are no simple errors, merge conflicts, or problematic code before the code can be merged. Secondly, whenever code is written, the creator must implement unit tests into the tests folder and ensure they pass before merging their code into the main branch. These tests must have complete branch coverage, meaning they test all possible pathways in which the program can run. This ensures that all methods and classes work properly before merging into main. This also helps ensure no added code creates bugs in our existing code so we will always have a working product in main. Finally, before any merge, the code has to be reviewed and approved by two members of the group so we can ensure the code is properly working. The testers will run the added code, and test from a user perspective, to make sure the code is fully functional. This also helps the group understand what others on the team are creating. If we find that these three testing steps are not adequate, we will look for new ways to test our project.

Design Patterns

Observer:

For our project, we have chosen to implement the observer pattern. The observer pattern is useful for the IoT web application because it allows for dynamic updating and communication between objects in the GUI and event-driven systems. This is particularly useful, not only as sensor values change, but as sensors are swapped. For this application, there will be an abstract Subject class where a sensor object is the subject. For each subject, the abstract Observer class will maintain a list of observers. These observers will include Alarm, Visualization, and Prediction classes. When the subject, or sensor, updates or is replaced, the observers will have an update() method that will be called. An illustration of this pattern is shown below in Figure 5.

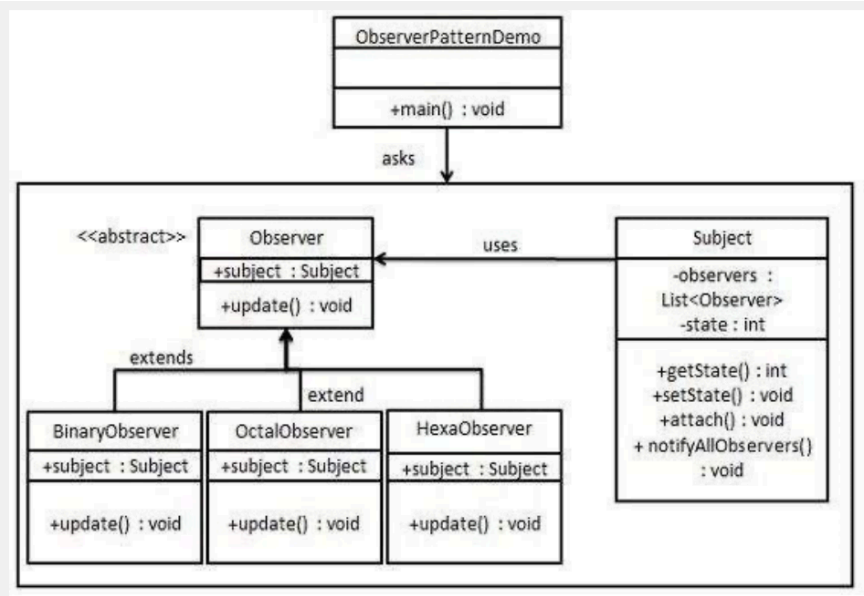

Figure 5. Observer Pattern Example

MVC Pattern:

Another design pattern our project will integrate is the Model-View-Controller pattern (MVC); which is commonly applied for web applications. In our use case, it can separate the application's data, UI, and control logic into manageable modules. This means our UI will be built using flasks which will be separate from our data. Our data will be a separate class representing the queries and GET REQUEST interactions with Thingspeak. Finally, our control module will manage the sensors, alarms, and visualizations. By using MVC these separate concerns can be developed, tested, and maintained easier. A diagram of the MVC pattern can be found below in Figure 6.

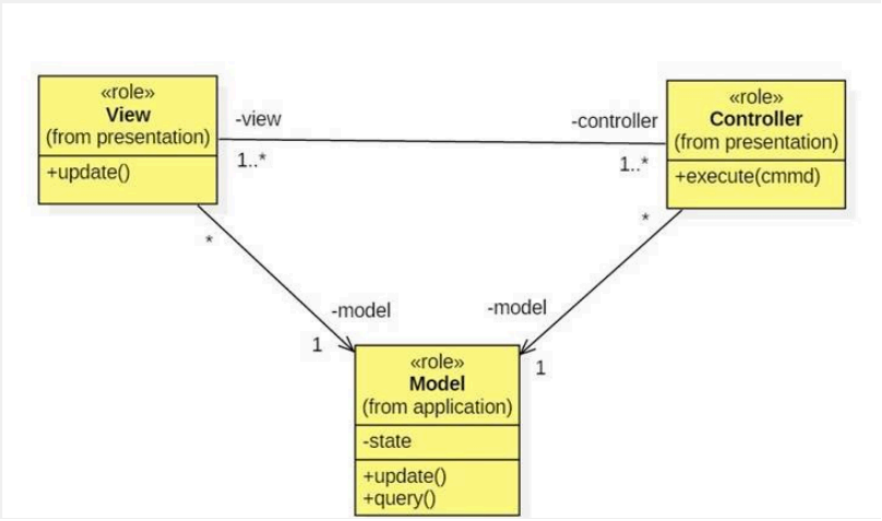

Figure 6. MVC Pattern Example