COSC 310 Software Engineering

MILESTONE 3
Formal Analysis & Architecture

Due March 11, 2024

gitGoblins (L01)
Eric Harrison
Ian Steyn
Joshua Ward
Chase Winslow

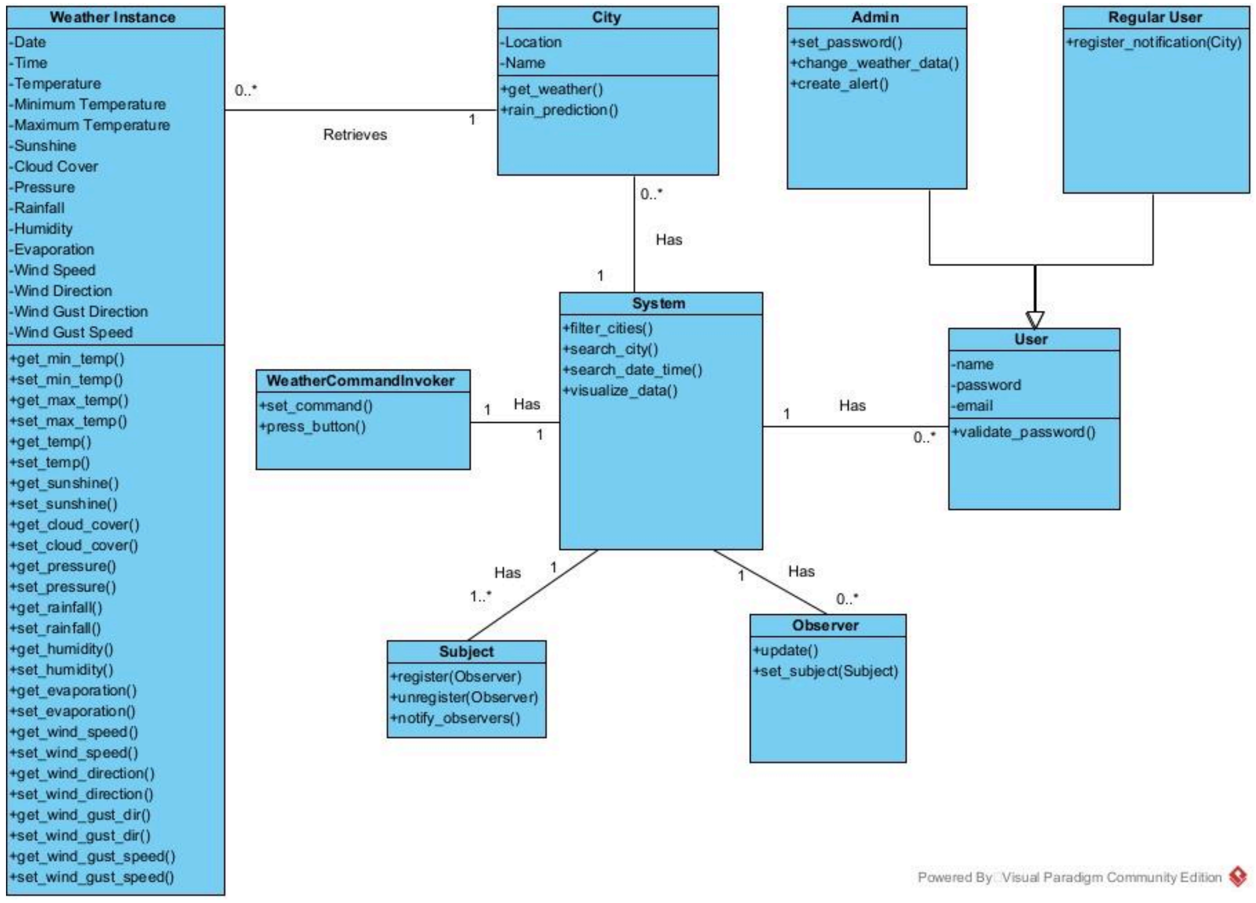
Links:
GitHub Repo
GitHub Dashboard

Overview

By this point in our project, we have spent time formally analyzing and planning our system design at a more detailed level. In this document, you will find the following:

A. **System Class Diagram** - A UML diagram describing the major components of the system
B. **Sequence Diagrams** - Four UML diagrams that describe the how the system components interact over time for some of our major uses cases
C. **Data Flow Diagrams** - Two diagrams that describe the flow of data inside and outside the system
D. **Design Patterns** - Descriptions of two of the design patterns we are using in the development of the system
E. **System Architecture** - Outline of how the application is actually put together with development tools
F. **Test Plan** - A basic outline of how we will test our system as we move into actually coding and implementing features

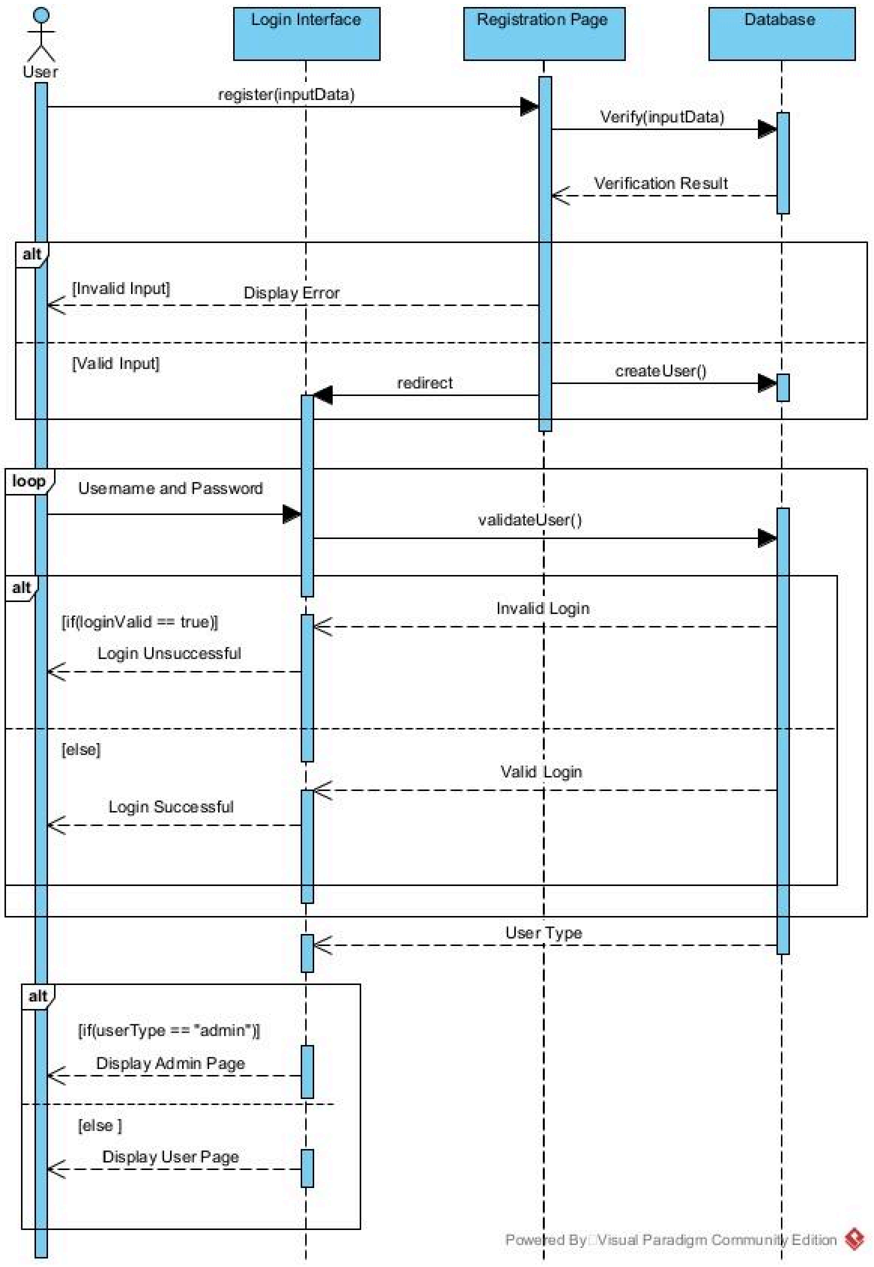
A. SYSTEM CLASS DIAGRAM



- **Overview**
    - The UML Class diagram shows an overall look at the system. It details the specialization of the User class into two distinct subclasses, an Admin User and a Regular User, each with different roles, powers, and views into the system. The diagram also includes a general look at the Command and Observer design patterns that are expanded upon in section D with their own respective diagrams. Each city can retrieve a specific instance of its weather data from the database based on time and date, which gets sent back to the application to be shown through the user interface.

B. SEQUENCE DIAGRAMS

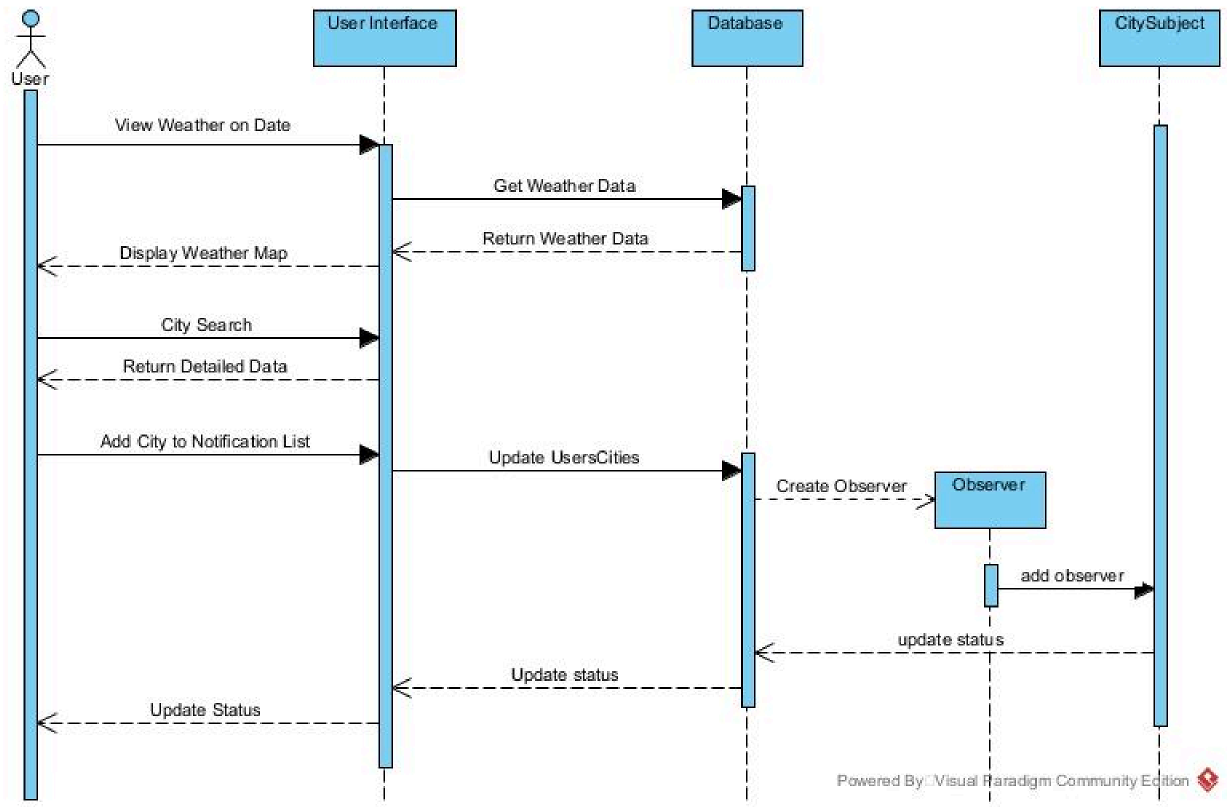
User Login



Overview

- The user login sequence diagram describes the main use case for getting into the application. A user can register for an account and afterwards will be redirected to the login page. The application will check for a valid login and check the user type - whether the user that is logging in is a regular user or an admin - and give them access to the appropriate view for their respective user type.

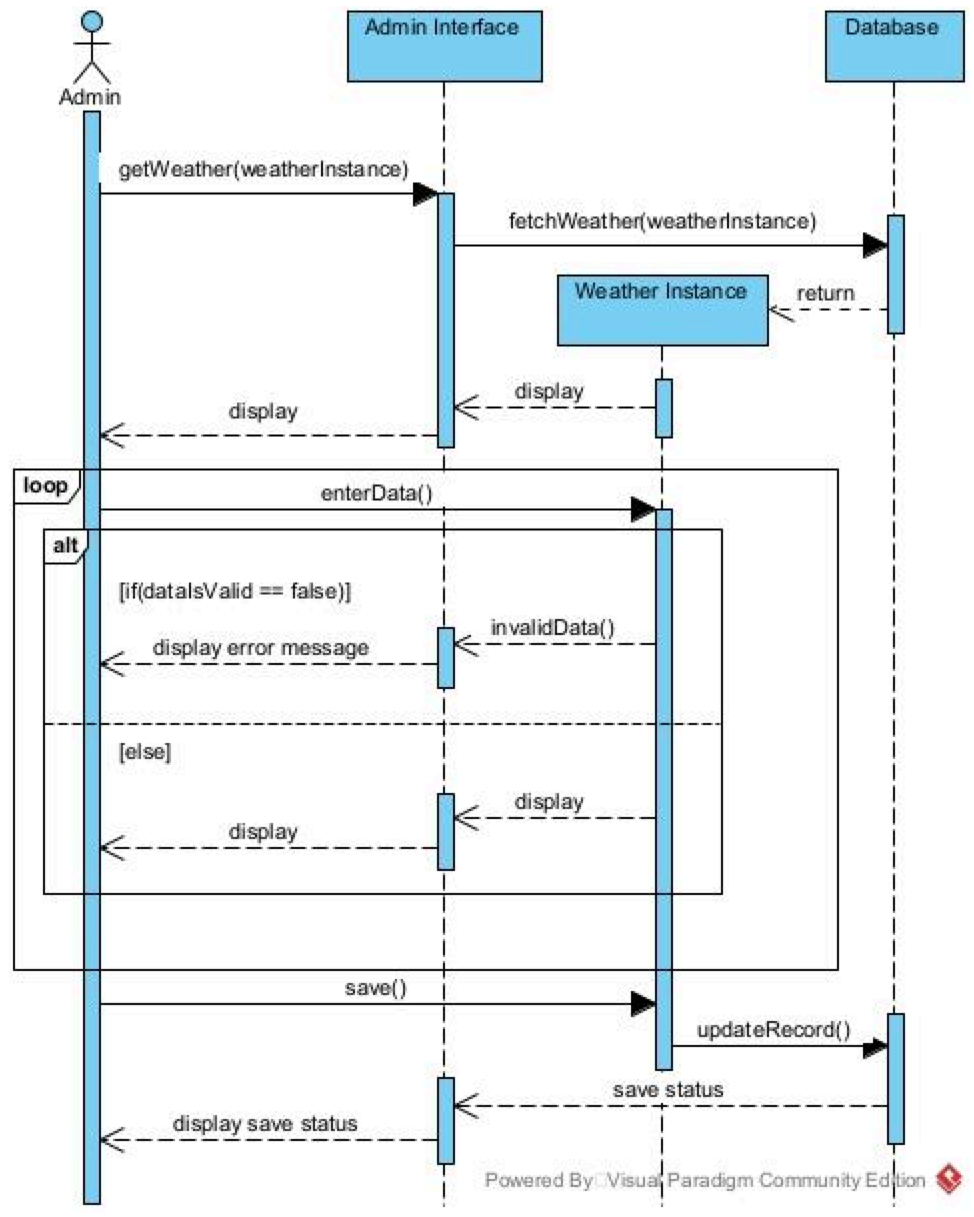
User Use Case



Overview

- The User sequence diagram shows the main use case for the regular user. The user interacts with the user interface to view weather data, and search for both the date and the city that they want to view the weather data for. They are also able to add cities that they want to get updates for; cities that they will get notified about if any alerts are made.

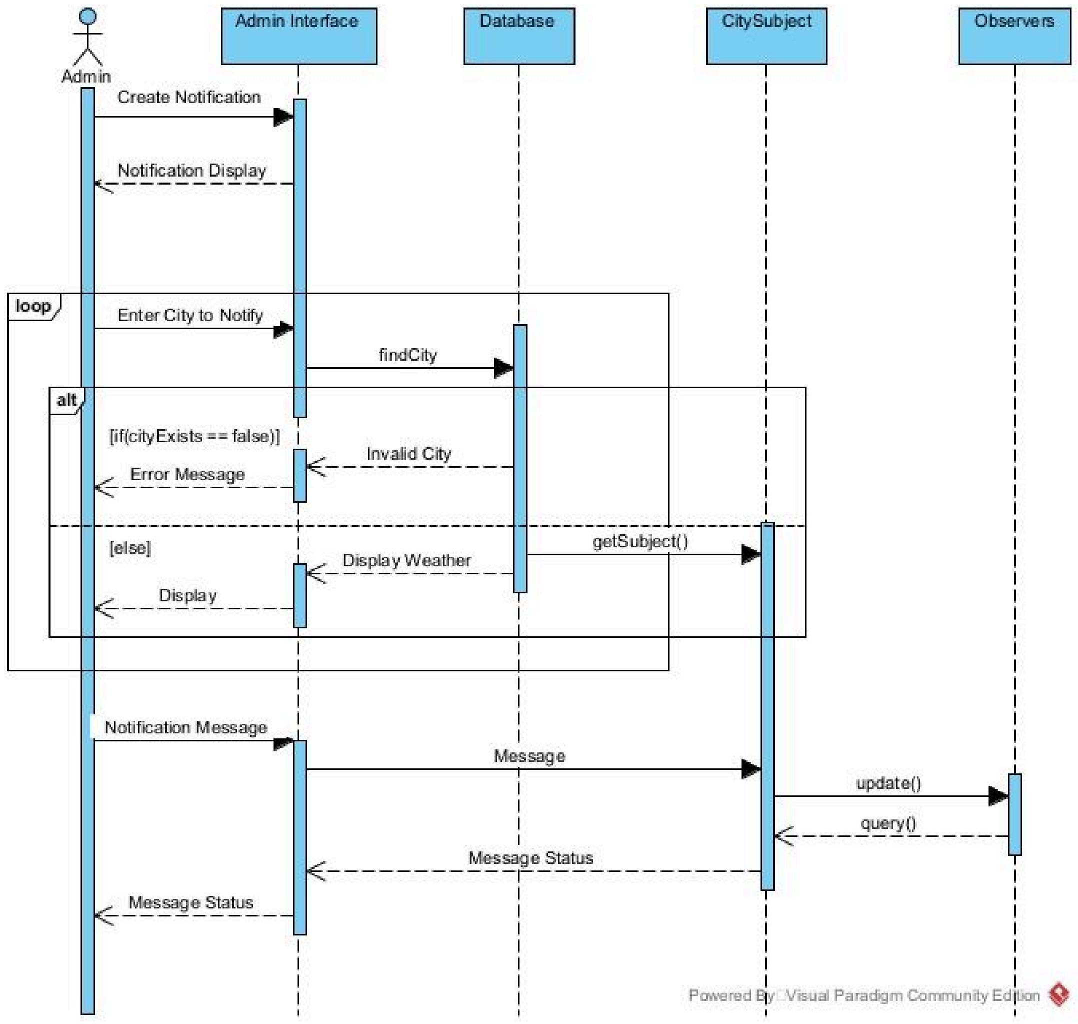
Admin, Update Weather



Overview

- The Update Weather sequence diagram details the use case for an admin updating specific weather data for a city. The admin requests the data for a specific city and time, and the database fetches it and returns the information. The admin can update the data by changing any of the fields, which is checked against the restraints on the field being updated. The admin then saves the data, which updates the changed fields to the database, and displays that status of the update.

Admin, Create Weather Notification

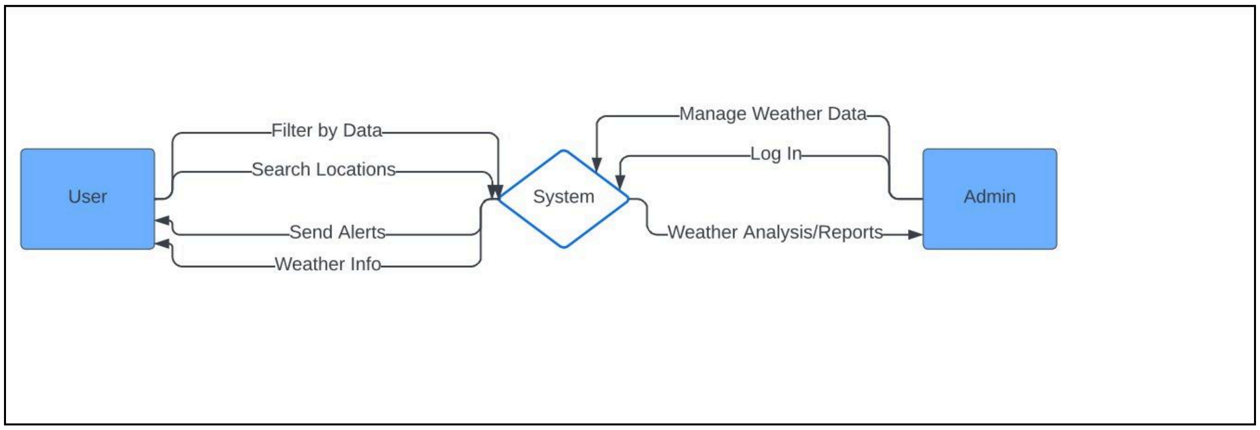


Overview

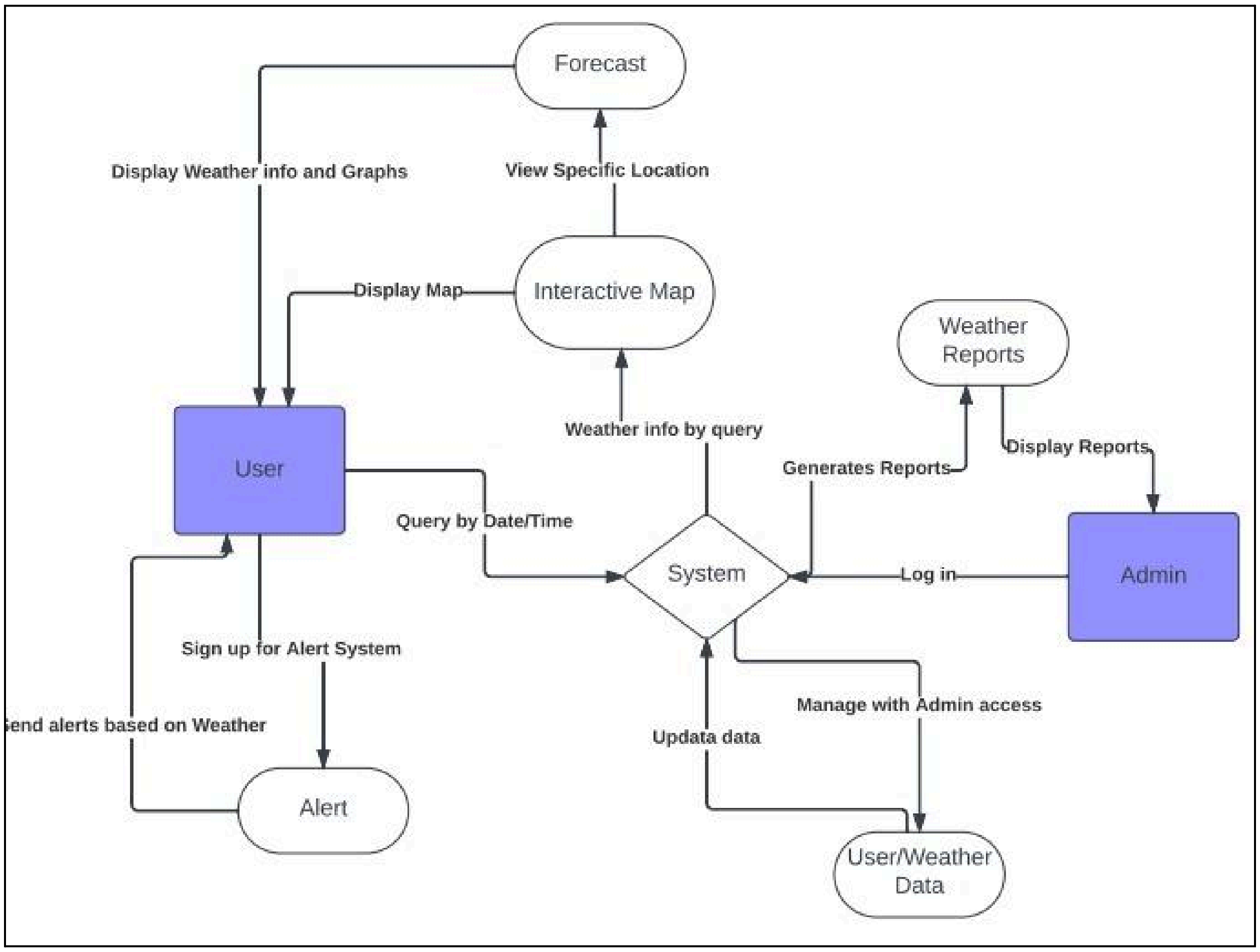
- The Create Notification sequence diagram details the use case for an admin creating a weather alert that gets sent out to anyone subscribed for weather alerts to the respective city. The admin inputs that they want to send an alert, and enters the city that they want to create an alert for. The application retrieves the data for the city that the admin requests, and gets the Subject related to the city. The admin enters the alert message, and sends it to the Subject to update its respective Observers. The admin is updated on the status of their message.

C. DATA FLOW DIAGRAMS

Level 0



Level 1



D. DESIGN PATTERNS

1. Observer

Quick Overview:

- This is a behavioural design pattern that lets subscribers receive notifications from a central source.
- The Observer Pattern Explained and Implemented in Java - Geekific

Reasoning:

- Extreme weather events such as heat waves or heavy rainfall are critical information for users.
- Subscribers interested in extreme weather events need to be notified when these events occur.

How will we implement it into the system?

Subject (WeatherStation):

- Maintains a list of observers interested in weather updates.
- Includes methods to subscribe, unsubscribe, and notify

ConcreteSubject (AustralianWeatherStation):

- Extends the ConcreteSubject to include a mechanism to identify extreme weather events.
- Notifies observers with details of the extreme event when it occurs.

Observer (WeatherComponent):

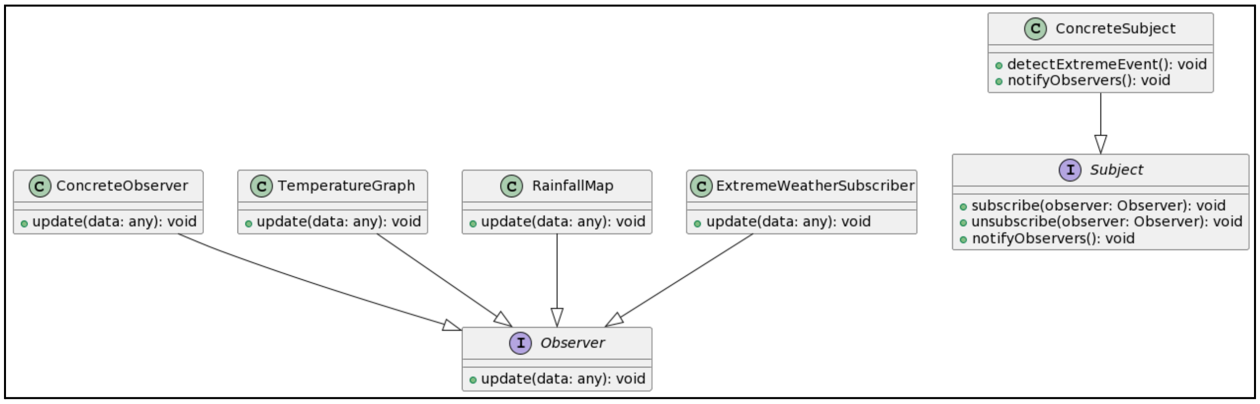
- Enhances the update method to include information about extreme weather events.

ConcreteObserver (TemperatureGraph, RainfallMap, ExtremeWeatherSubscriber, etc.):

- Implements the updated update method to react to changes in weather data.
- If an extreme weather event is detected, the observer can take appropriate action (e.g., display a notification).

ExtremeWeatherSubscriber:

- A specialized observer that specifically subscribes to extreme weather events.
- Implements the update method to react specifically to extreme weather events.



2. Command

Quick Overview:

- This is a behavioural design pattern that force requests to be their own encapsulated object
- [The Command Pattern Explained and Implemented in Java - Geekific]

Reasoning:

- The web application likely involves various user actions related to weather, such as refreshing data, changing the location, or switching temperature units.
- Commands encapsulate these actions, allowing for parameterization, queuing, and logging.

How will we implement it into the system?

Command Interface (WeatherCommand):

- Declares an execute method to perform the weather-related action.

ConcreteCommand Classes (RefreshDataCommand, ChangeLocationCommand, SwitchTemperatureUnitCommand):

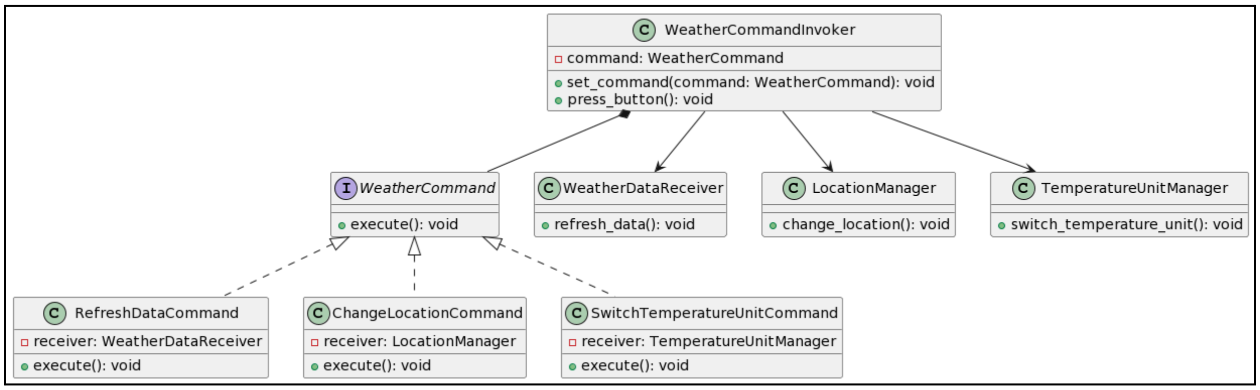
- Implement specific weather-related actions (commands).
- Hold references to receivers (e.g., data fetchers, location managers) that perform the actions.

Invoker (WeatherCommandInvoker):

- Sets and invokes commands.
- Can maintain a queue of commands for additional functionality (e.g., undo/redo).

Receiver Classes (WeatherDataReceiver, LocationManager, TemperatureUnitManager):

- Perform the actual weather-related actions.
- Receive and execute commands.



E. SYSTEM ARCHITECTURE

This project is a Python-based web application. Below are the development tools that bring it all together:

- ***Core programming language:*** **Python 3.12**
  - A versatile, weakly typed, multi-paradigm language with a focus on OOP, readability, and flexibility.
  - Support offered for 3.11.
- ***Framework:*** **Flask 3.0**
  - A microframework for Python web app development that provides methods and objects that act as code abstractions for components and processes of your web application.
  - It's the glue holding everything together. There's a lot it doesn't do, so it provides extensions that allow you to use other tools/libraries alongside it.
  - This runs the app locally in a browser.
- ***Webpage Rendering:*** **Jinja 2**
  - An HTML template rendering engine that is installed alongside Flask by default. It provides special syntax for HTML, which:
    - Allows templates to inherit elements from a base template, so you don't have to re-write HTML for similar web pages
    - Uses passed variables to perform logic operations that decide exactly what HTML will be displayed
- ***Database language:*** **SQL**
  - Specifically used for storing and processing information in our database.
- ***Database Interface:*** **sqlite3**
  - A Python library that allows us to create and manage our database using SQL without needing a separate server to host it.
- ***Hosting:*** **local**
  - We are not currently hosting either the app or the database anywhere other than our local machines. While this may change in the future, it is sufficient during development.
- ***Other:***
  - In the next phase of our project, we will be implementing several features that will make use of other libraries and plug-ins. Luckily, we don't have to do anything special to accommodate those in our system architecture design, since Flask provides support for most of them to be seamlessly integrated into the project.

F. TEST PLAN

As we continue to write code and move into the feature implementation phase of our project, we will have to test that it all works. While we haven't written many tests yet, we have started implementing a testing framework, and we've outlined the plan for future testing below.

***Front-end Testing:* Our eyes.**
Testing the front end of our system is mostly done visually. This means we just run the application and open it to see if the HTML/CSS formatting and template rendering is working as expected. Of course, we can write unit tests to check that HTML components show up on the webpage, but this is cumbersome and will be done sparingly (likely one component per page, maximum.)

***Unit Testing:* pytest.**
These are tests that we write in Python code to test our other Python code. We are using the pytest framework because it has good documentation and is supported by Flask. All unit testing stuff is contained in the `tests` folder of our project:

- *Configuration* is contained in `conftest.py`, which creates a temporary database, app, and other fixtures, i.e. code and objects that will be used across many tests.
- *Unit Tests* are contained in the several files that start with `test_`. We try to stick by the following guidelines:
    - One class/module ⇒ one test file.
    - One function/action ⇒ one unit test
    - Aim for full coverage; give tests data that covers all decision branches

***Integration Testing:* GitHub Actions.**
Our project has a continuous integration (CI) workflow set up that runs through GitHub Actions. This workflow runs automatically on every push and pull request. It installs dependencies, builds the program, and runs *all* unit tests on several virtual machines, so that we can test the program on different operating systems (latest releases of Ubuntu, Windows, MacOS), as well as a few different versions of Python (currently 3.11 and 3.12).

Essentially, the CI workflow catches little errors that we might not if we simply ran tests on our own machines, and it ensures that new code does not break old tests.

[END OF REPORT]