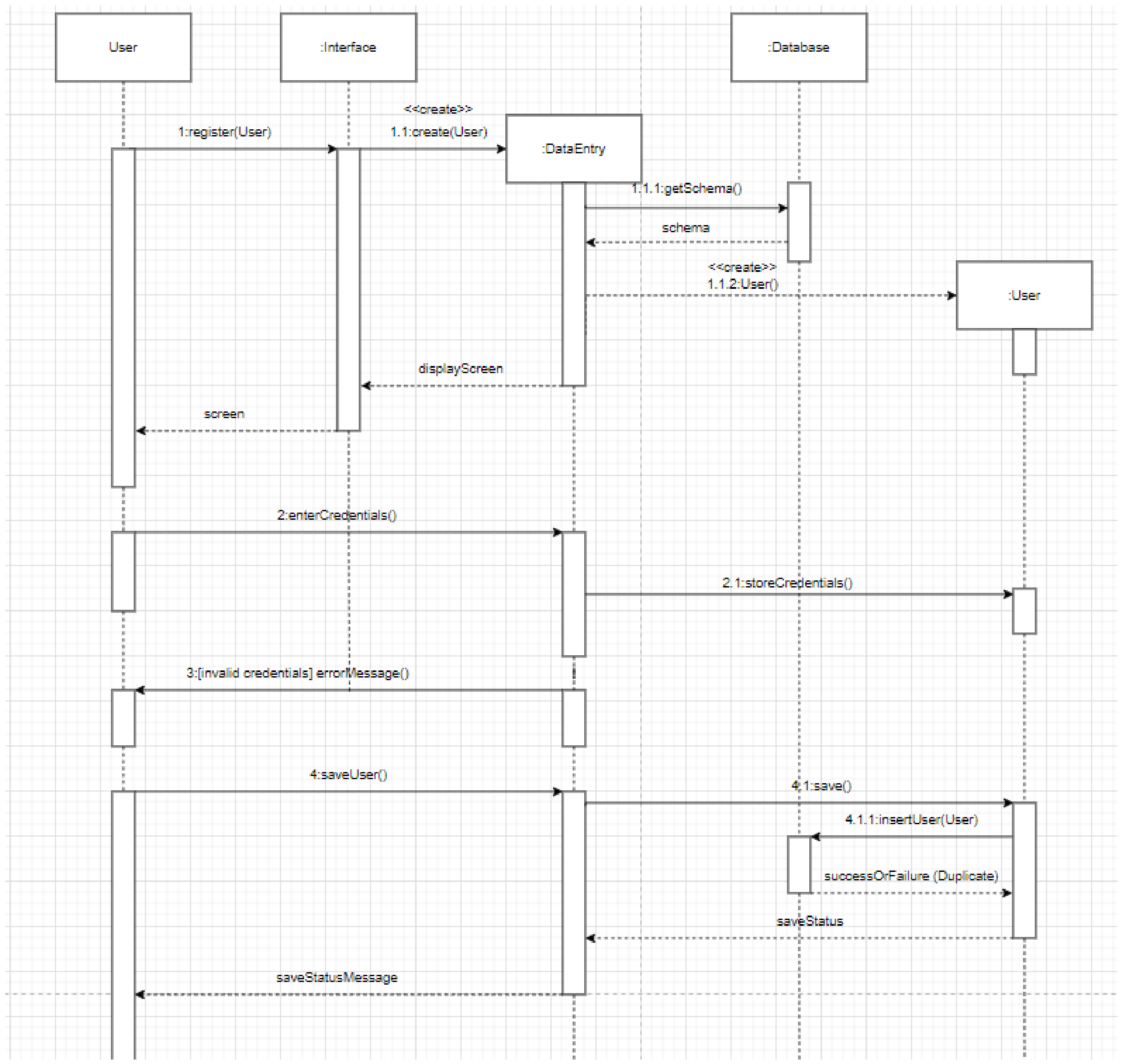
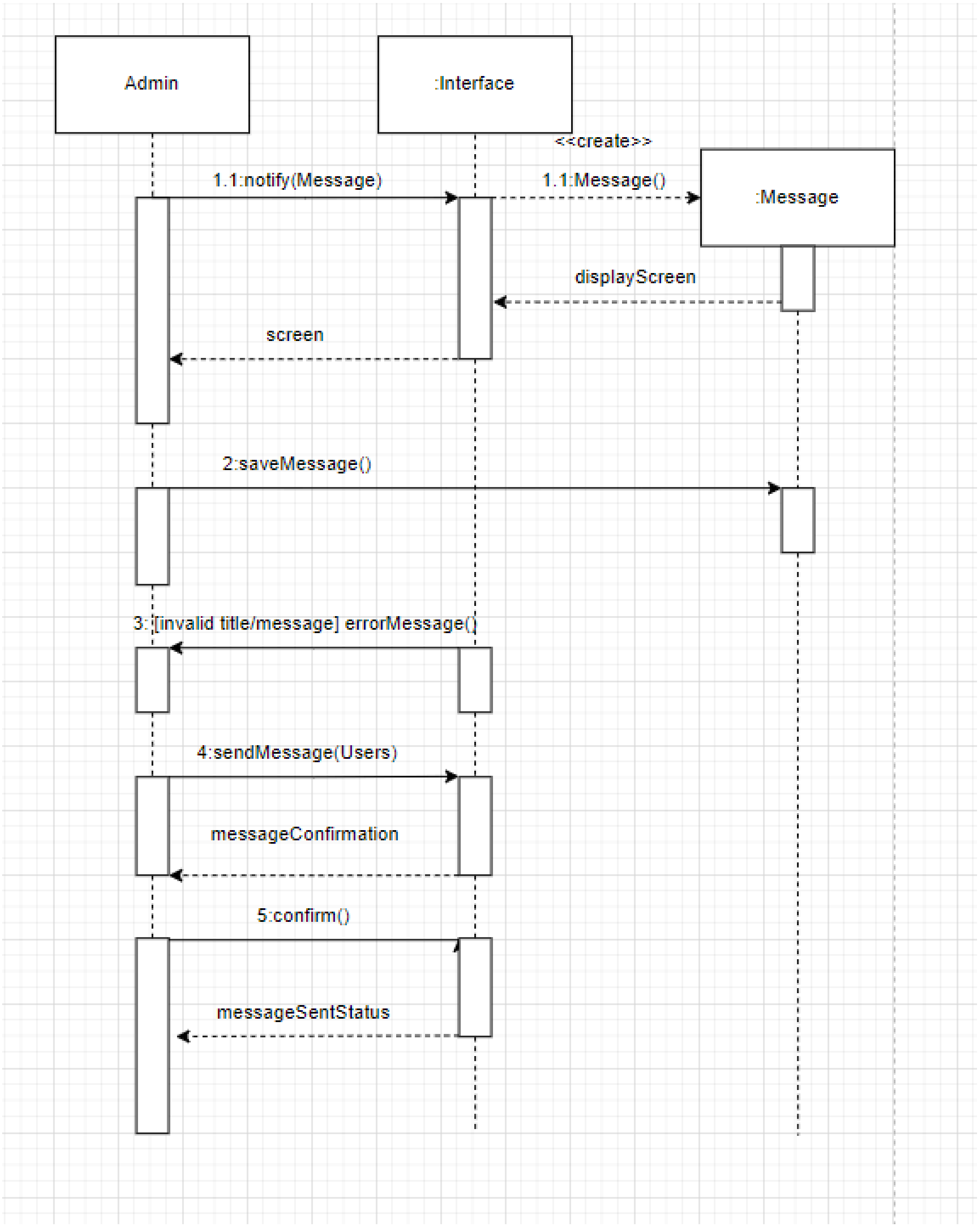
Team name: Infinite Loopers

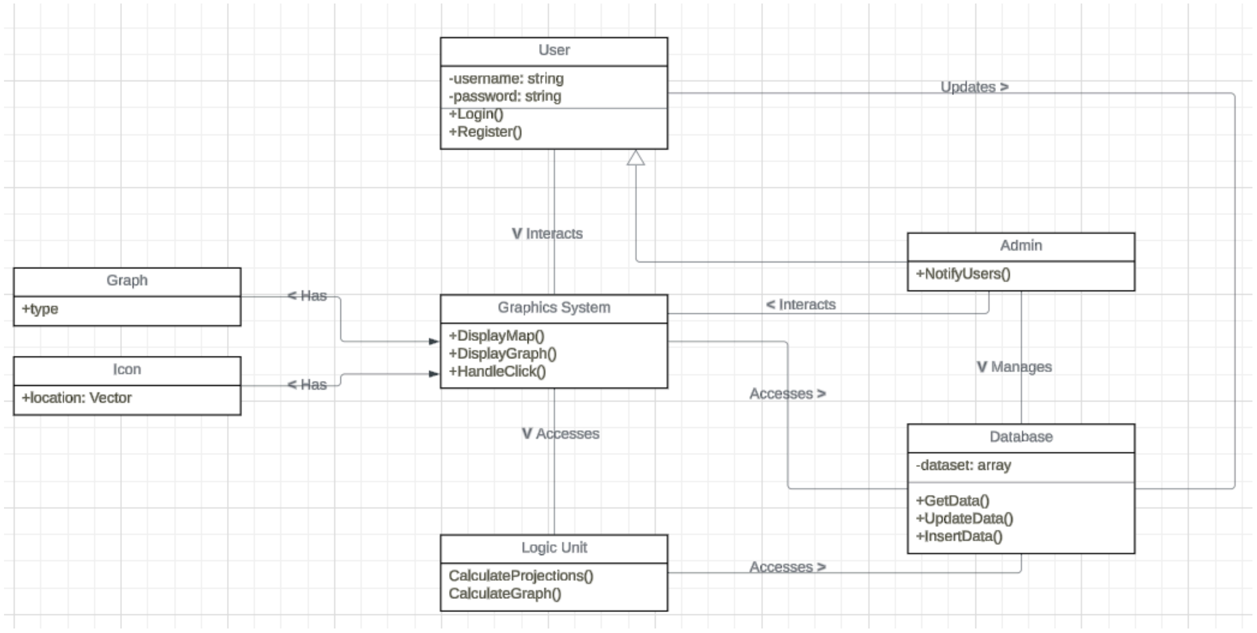
M3: Formal Analysis and Architecture COSC 310

Sequence diagrams (5 marks)




Admin
:Interface
<<create>>
:Message
1.1:notify(Message)
1.1:Message()
displayScreen
screen
2:saveMessage()
3: [invalid title/message] errorMessage()
4:sendMessage(Users)
messageConfirmation
5:confirm()
messageSentStatus

- Class diagram (5 marks)



Testing plan

1. Introduction
    a. Objective: By implementing a testing plan for our project, we will ensure consistency throughout the development process while also promoting test-driven development practices.
    b. Scope: The scope for the testing mainly focuses on all of the user stories we are implementing. Before release, we want to ensure that all our features are thoroughly tested, with a high level of coverage and test automation, to ensure that the app remains healthy long into the future.
2. Testing Strategy
    a. Testing Levels
        i. Unit testing: After implementing any new code unit, we must develop a test file that tests the code to ensure it works as intended. When we connect our backend to our web app front end, we will test a GET / PUT / POST for each database table to ensure it works properly.

        ii. Integration testing: Once code units are merged in our code base, we will conduct black and white box testing to ensure the code works after integration. Whenever we merge code into the pull request, we will perform QA testing of all basic web app functions to ensure that everything works as intended.
        iii. System testing: Going a step further from integration testing, our system testing process will involve writing test cases involving multiple units interacting with each other, ensuring working code. The number of tests written will be based on any common user interaction that requires communication between two or more code units. After each feature is complete, we will write tests that ensure that all the work units function together properly.
    b. Risks: While different levels and types of testing each have their own risks, we will take an approach that involves multiple testing strategies, which will allow us to overcome the downsides of some testing methods. Also, with five team members writing tests, we may run into consistency issues.
3. Testing Environment
    a. Software Requirements: Most of our testing will be done in VS Code, Python, SQL Lite, and the live code environment. We will be able to write test cases in Python and do QA work in the live environment to find and fix bugs throughout the development process.
    b. Managing Test Data: Throughout the development process, we will create test data in the backend. For SQL Lite, we will create test login data to test the login and other features with mock data. For Python testing, we will use assert statements and create mock user interactions that will allow testing to be most effective.
4. Objectives
    a. Testing Coverage: For any Python code, we want at least 90% code coverage from our corresponding test files. For database testing, we want to ensure that each column or data has been tested individually. Lastly, for front-end

testing, we will implement User / QA testing, where the team will test the front-end functionality after every code integration. The requirements for this will be specific based on the code being merged.

   b. Goals: Our code should remain healthy throughout the development process by conducting specific testing in this manner.

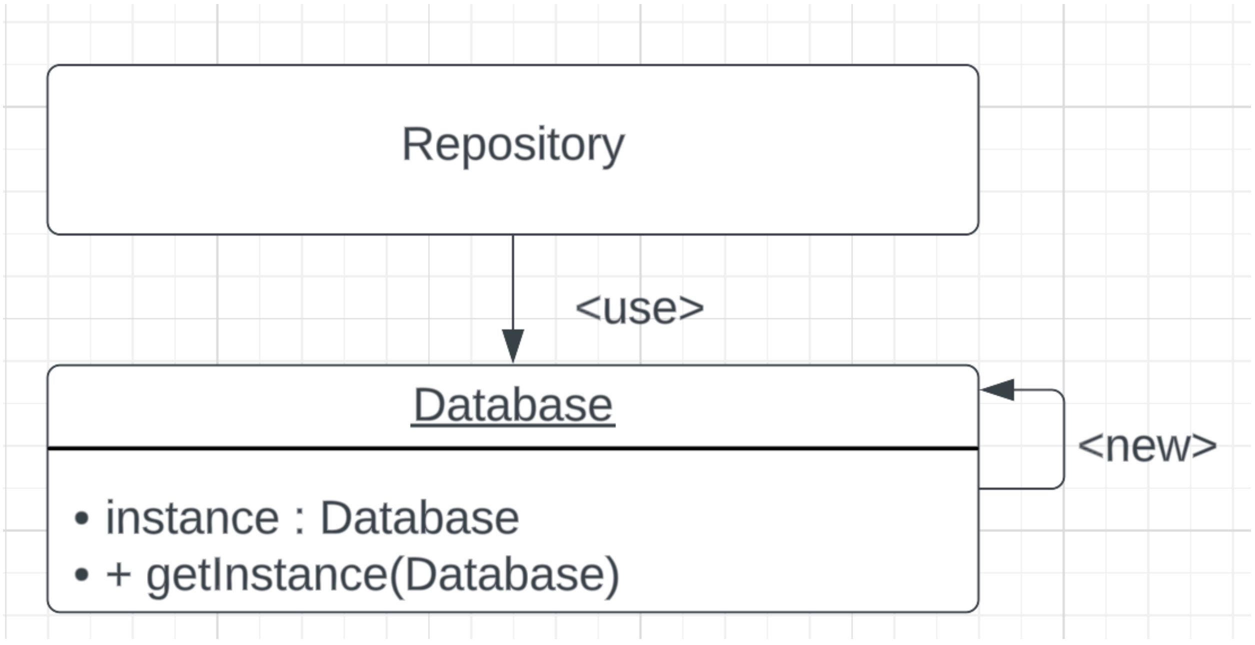
5. Prioritization and Management
   a. Tracking: We will track testing on the Kanban board and within the specific issues. Each programming task will have a test file requirement to ensure that the code has been tested properly; the requirements will differ depending on the type of programming task undertaken. Also, upon a pull request, the team will perform user testing to ensure all requirements have been met and test the code on their local machine to ensure everything functions as it should.
   b. Prioritization: Testing and pull requests will be FIFO, meaning that as they come in, we will test the team's code and ensure that there is no backlog. As code is being written, we will follow test-driven development, so before any major programming, we should have the proper tests in place to ensure that the code written will meet the task requirements.
6. Test Automation
   a. Tools: Using Github actions, we will implement test automation to run all the tests we have created (unit integration, etc.).
   b. Scope: Our test automation will mainly cover the tests in our test files and some integration testing to ensure that each time a new code is added, the system as a whole still works properly.
7. Approvals and Reporting
   a. Approval Process: When creating a PR, we will ensure that at least 2 team members approve of the code, with at least 1 performing QA tasks to test the code.

b. Test reporting: We will communicate the testing process on the Kanban board and on PRs with comments to keep the team updated throughout the testing process.

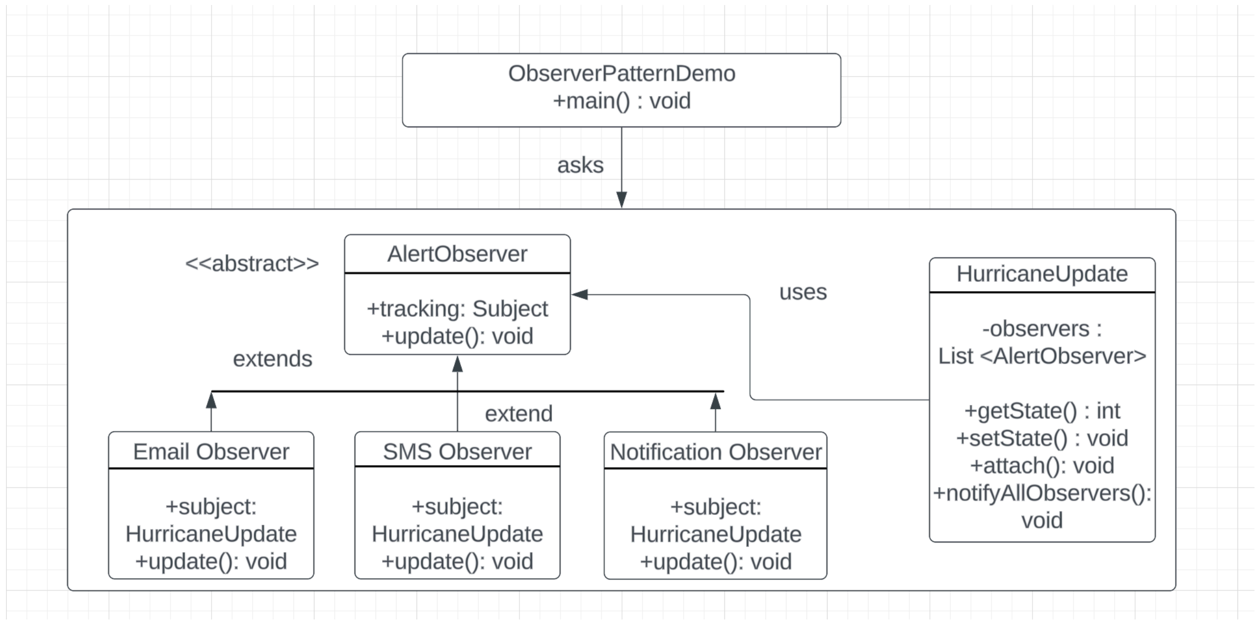
Design patterns

Singleton Pattern: Chosen for ensuring that only one instance of a class, such as a centralized database, exists throughout the application. It ensures global access and control over actions like data updates or alerts distribution, crucial for maintaining consistent state across the application.

The overall interaction depicted in this diagram showcases the Singleton pattern's primary goal: ensuring that there is only one instance of the Database class (ensuring a single database connection or access point in this context) and providing a global access point to that instance through the getInstance method.



Observer Pattern: Selected for managing updates between the system and its users, especially for real-time hurricane tracking and alert notifications. Each observer subscribes to updates from the HurricaneUpdate subject. When the state of the subject changes (signifying a new hurricane update), each subscribed observer's update() method is called, enabling each observer to act upon the new information, such as sending out alerts through different channels. This design ensures that all parts of the system that need to react to changes in hurricane activity can do so promptly and accurately.


ObserverPatternDemo
+main() : void
asks
<<abstract>>
AlertObserver
+tracking: Subject
+update(): void
uses
HurricaneUpdate
-observers : List <AlertObserver>
+getState() : int
+setState() : void
+attach(): void
+notifyAllObservers(): void
extends
extend
Email Observer
+subject: HurricaneUpdate
+update(): void
SMS Observer
+subject: HurricaneUpdate
+update(): void
Notification Observer
+subject: HurricaneUpdate
+update(): void