UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Plant Nursery Simulator in C++

COS 214 Project

Git It Done

2025

GitHub Repository

https://docs.google.com/document/d/1vor777XFPyVh4oTEEA_yd9F96e1kfakO/edit

Content

Group Members

Abdelrahman Ahmed (u24898008)

Abdulrahman Sabah (u24566170)

Antony Van Straten (u24590739)

Avuyile Sapula (u23540282)

Dylan Hebron (u22503685)

Hamdaan Mirza (u24631494)

Joshua Heath (u23541475)

Research Brief

Throughout the development of our nursery system, we have done intensive research in ensuring that our system properly replicates a real-life one. Firstly, we had to note that plants are not static objects that will just be stored in our inventory. In fact, they are put through different production stages that change the attributes and care that is needed while our system can constantly reflect that [1].

Additionally, another important factor is how communication/requests were handled throughout the system. This is especially important as it's the only way that the state of the system will change thus from our research, we noted that it's best to handle the request from a bottom-up hierarchy where commands are passed through the lowest in the systematic hierarchy until finally reaching the top [2].

Due to this system being coded on C++ and our natural desire to create a Graphical User Interface, we had to research a way to achieve this as our knowledge is limited. We noted that from our research that we needed **adapters** to convert our C++ requests into JSON so that we could handle them accordingly as we learnt to do so in COS216 (Web Development) [3]. In addition, we noted that a GUI without a server is pointless, thus through more extensive research we figured out how to create a server on in C++ and host the GUI on our desired port [4].

Finally, we figured out which patterns to use in different scenarios and thus the hardest challenge would arise, which was how they would all work together. We realized that our patterns had to be combined and layered to mimic a real-life system [5]. Resulting in the conclusion of our research.

Bibliography

[1] M. A. L. Smith, "A Realistic Inventory Control Project for Nursery Management Students," HortScience, vol. 21, no. 6, pp. 1289–1291, 1986.

[2] S. McConnell, *Code Complete*, 2nd ed., Microsoft Press, 2004, pp. 127–129.

[3] N. Lohmann, "JSON for Modern C++," GitHub Repository, 2013. [Online]. Available: https://github.com/nlohmann/json

[4] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming: The Sockets Networking API*, 3rd ed., Addison-Wesley, Boston, 2003.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996

Functional Requirements

Subsystem 1: Plant Nursing

- FR1.1: The system shall allow replicating an existing plant.
- FR1.2: The system shall make it possible to produce various plant types.
- FR1.3: The system shall track plants' life cycles and notify about state changes.
- FR1.4: The system shall act upon plants' state changes and execute required actions.
- FR1.5: The system shall take care of all plant types according to their different needs.
- FR1.6: The system shall simulate plants' state transitions.

Subsystem 2: Customer Management

- FR2.1: The system will allow customers to personalize and decorate their plants.
- FR2.2: The system shall organize the communication between the delivery staff and the customers.

Subsystem 3: Staff Management

- FR3.1: The system shall enforce a hierarchy for handling tasks in the nursery and allocate them to a suitable handler.

Subsystem 4: Inventory Management

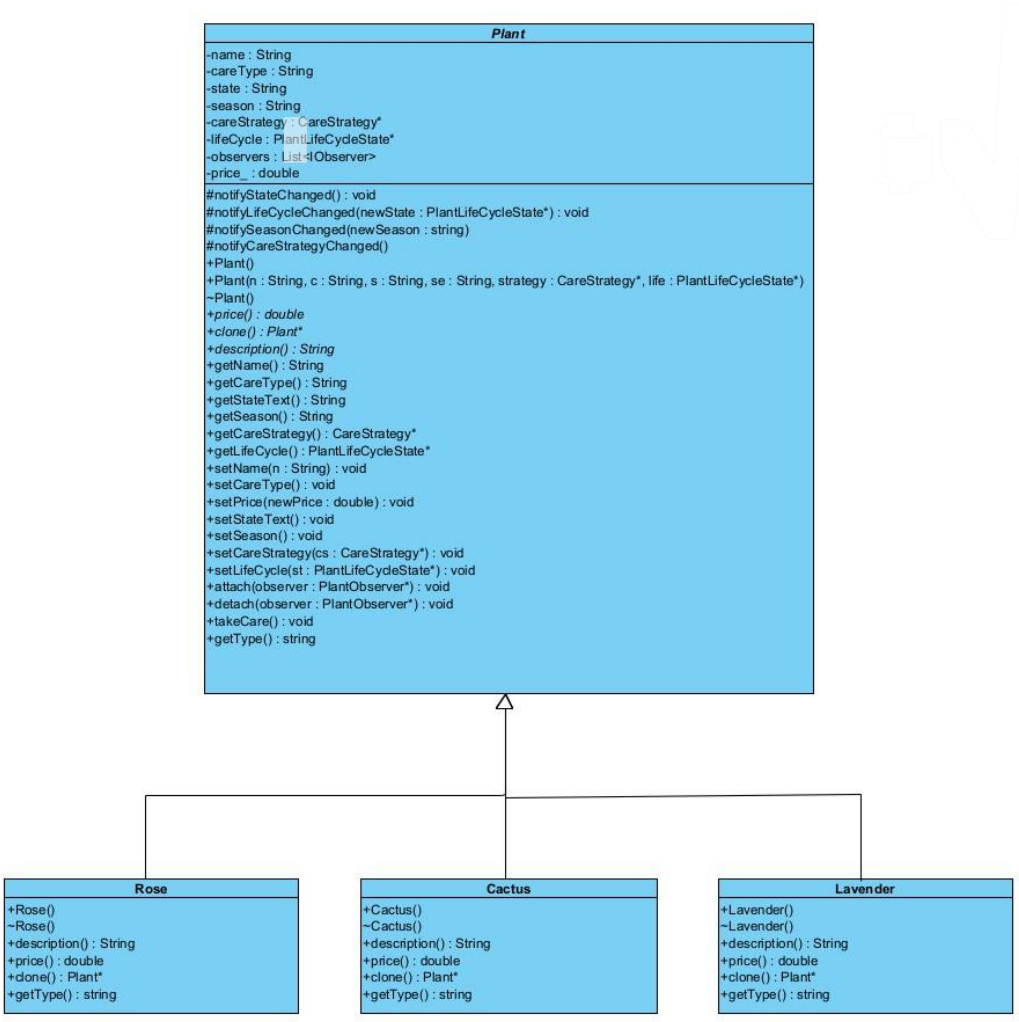
- FR4.1: The system shall enable staff to traverse through all plants in the inventory.
- FR4.2: The system shall ensure compatibility between the web application and the inventory interfaces.

Prototype (FR1.1)

Explanation:

- This will be utilized when we want to replicate an existing plant exactly. Instead of merely constructing an existing one, which takes time for individuals in general, this will help the system simplify things.

Structure:



Participants:

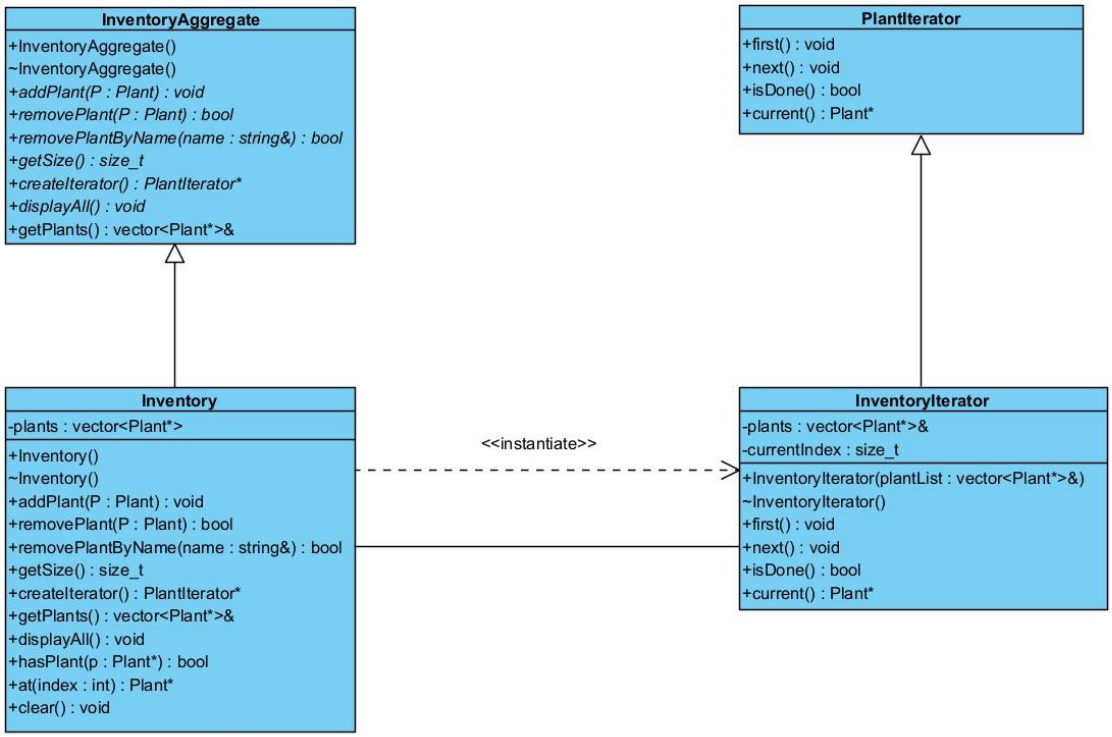
Participant	Class name
Prototype	Plant
ConcretePrototype	Rose
ConcretePrototype	Cactus
ConcretePrototype	Lavender

Iterator (FR4.1)

Explanation:

- Used to allow staff to traverse through the entire inventory without the need to worry about different inventory representations or behaviour .
- Therefore, to properly iterate through the Inventory class, we will include functions like ***first()***, ***next()***, ***isDone()***, and ***current()*** in our methods.

Structure:



Participants:

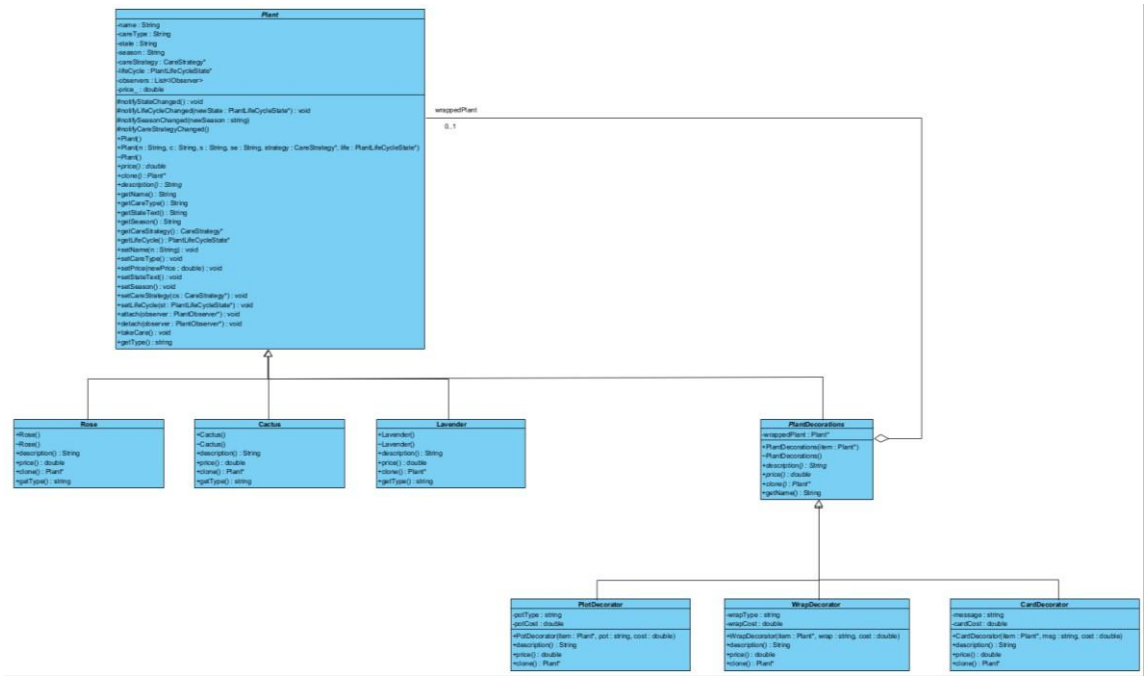
Participant	Class name
Iterator	PlantIterator
ConcreteIterator	InventoryIterator
Aggregate	InventoryAggregate
ConcreteAggregate	Inventory

Decorator(FR2.1)

Explanation:

- When a consumer wants to purchase something, we will provide them with additional features like a plot decorator, a wrap decorator, and a card decorator. They will be able to select the kind of decoration they like to add when they purchase the plant.

Structure:



Participants:

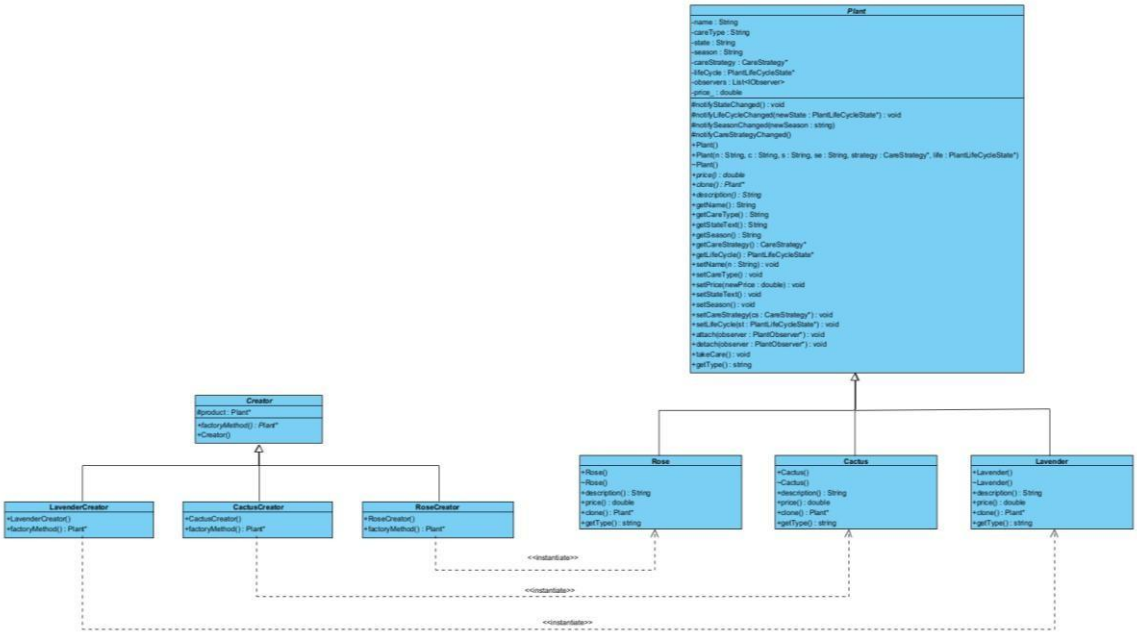
Participant	Class name
Component	Plant
ConcreteComponent	Cactus
ConcreteComponent	Lavender
ConcreteComponent	Rose
Decorator	PlantDecorations
ConcreteDecoratorA	PotDecorator
ConcreteDecoratorA	WrapDecorator
ConcreteDecoratorA	CardDecorator

Factory Method(FR1.2)

Explanation:

- The primary duty of the plant factory is to produce various plant varieties for our nursey. So, it follows a simple factory Design pattern, where a single factory class is used to produce multiple plant objects depending on the requested concrete type using concrete factories.

Structure:



Participants:

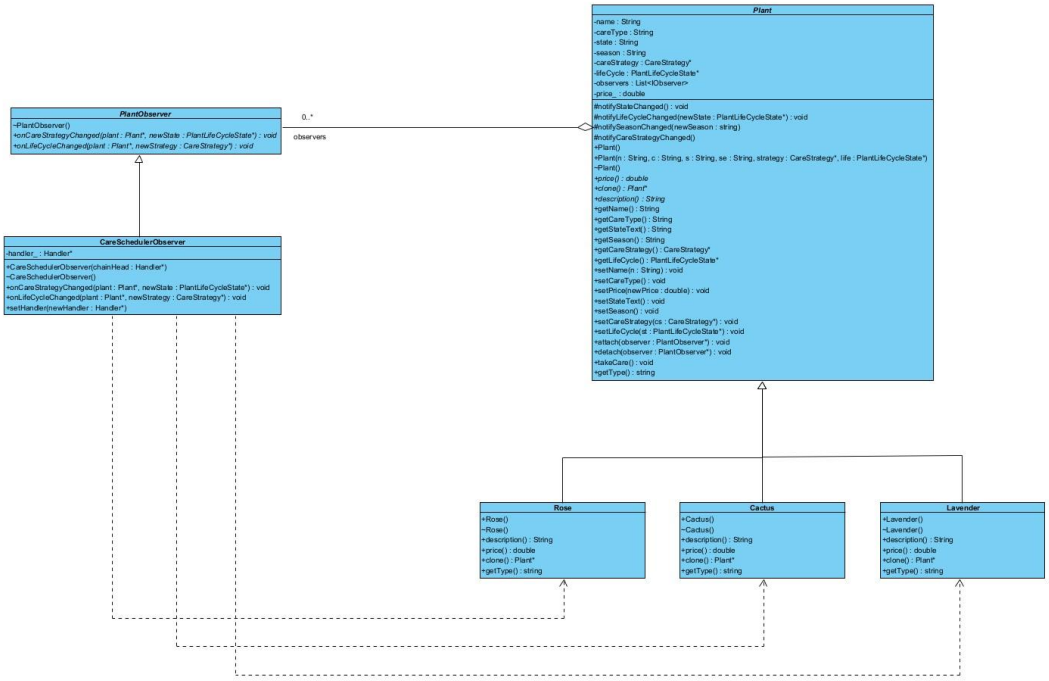
Participant	Class name
Creator	Creator
ConcreteCreator	CactusCreator
ConcreteCreator	LavenderCreator
ConcreteCreator	RoseCreator
Product	Plant
ConcreteProduct	Cactus
ConcreteProduct	Lavender
ConcreteProduct	Rose

Observer (FR1.3)

Explanation:

- An observer pattern is implemented in systems where one-to-many dependencies are prevalent. In Git It Done nursery this relationship is between the class ***Plant*** (Subject) and the class ***PlantObserver*** (Observer). This is done so when the plant changes its state, such as from ***SeedlingState*** to ***GrowingState,*** the observer classes mentioned above will be notified by a command such as ***GrowingStateCmd.*** This matches real life nurseries where plants are monitored by real time sensors, and when the plants' environment changes, staff are notified to act.

Structure:



Participants:

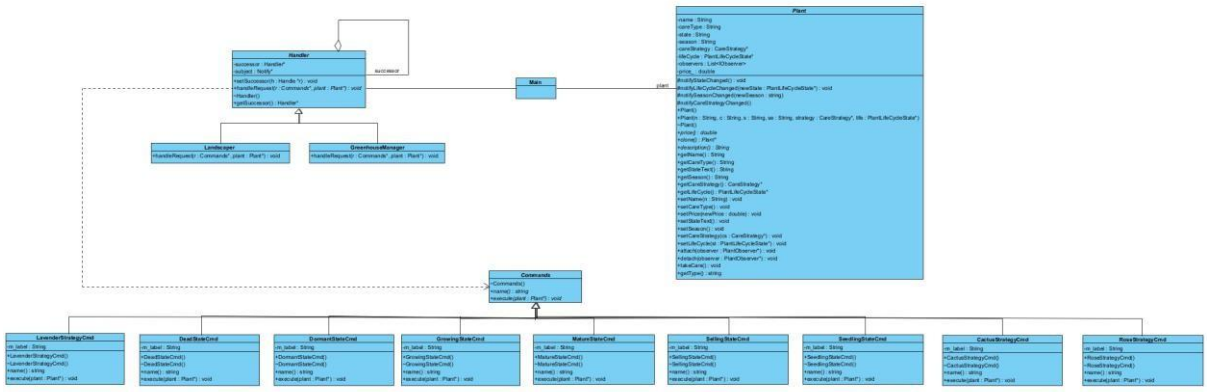
Participant	Class name
Subject	Plant
ConcreteSubject	Cactus
ConcreteSubject	Lavender
ConcreteSubject	Rose
Observer	PlantObserver
ConcreteObserver	CareScheduleObserver

Command (FR1.4)

Explanation:

Our commands, which are distinct objects, will encapsulate actions or requests that get executed on the different states of plants and according to care strategies after being notifies about a state or strategy change.

Structure:



Participants:

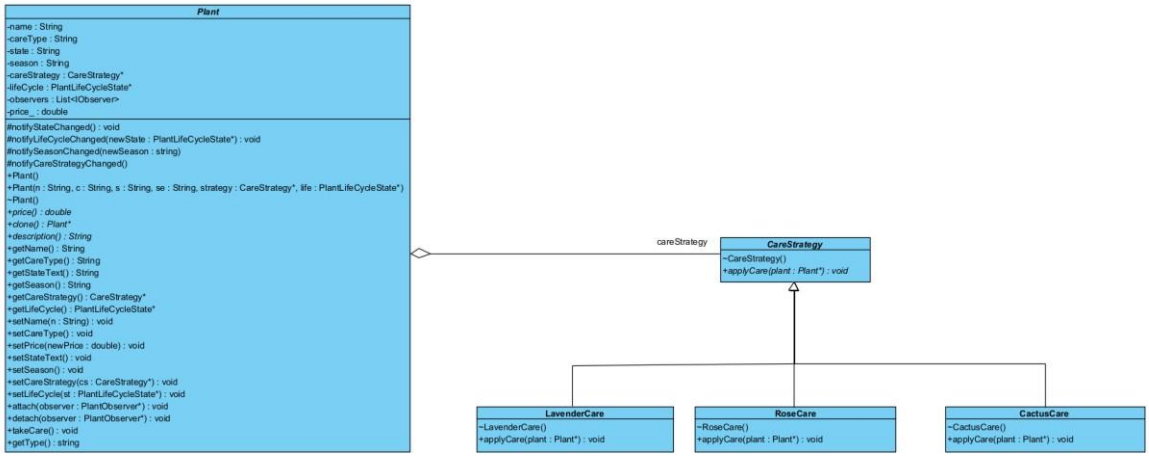
Participant	Class name
Command	Commands
ConcreteCommand	DeadStateCmd
ConcreteCommand	DormandStateCmd
ConcreteCommand	GrowingStateCmd
ConcreteCommand	MatureStateCmd
ConcreteCommand	SellingStateCmd
ConcreteCommand	SeedlingStateCmd
ConcreteCommand	CactusStrategyCmd
ConcreteCommand	LavenderStrategyCmd
ConcreteCommand	RoseStrategyCmd
Invoker	Handler
Reciever	Plant
Client	Main

Strategy (FR1.5)

Explanation:

- This class shows how different plants have different ways to be cared for. The project uses the Strategy Pattern, where each type of plant has its own "care plan" or strategy, rather than writing all the care logic inside a single large class. Thus, we'll have the following We will have RoseCare, a specific care type for roses, CactusCare, a specific care type for cactus type plants, and LavenderCare, a specific care type for lavenders.

Structure:



Participants:

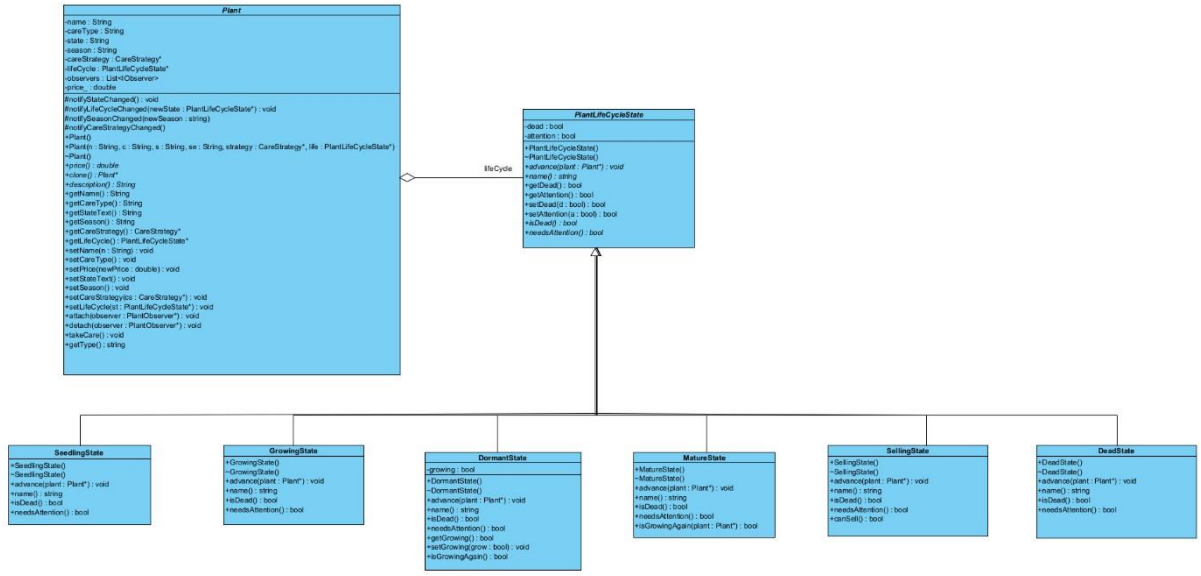
Participant	Class name
Strategy	CareStrategy
ConcreteStrategy	CactusCare
ConcreteStrategy	LavenderCare
ConcreteStrategy	RoseCare
Context	Plant

State (FR1.6)

Explanation:

- When an object's behaviour modifies its present state, the state pattern will be applied. For our project, the plant will go through multiple phases of its life cycle: Seedling → Growing → Dormant → Mature → Selling. Also, Dead state is possible.

Structure:



Participants:

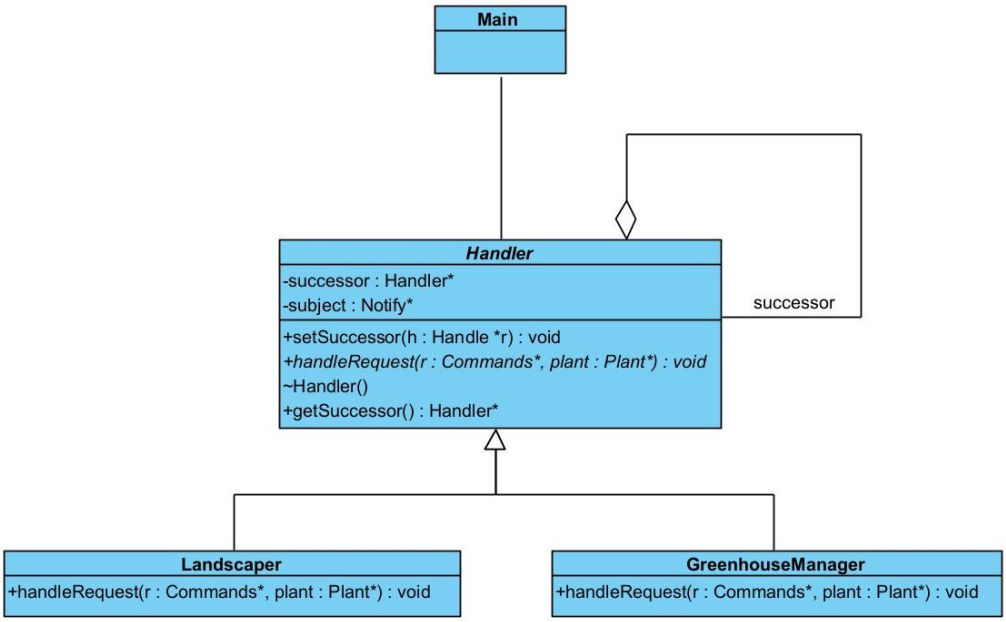
Participant	Class name
State	PlantLifeCycleState
ConcreteState	SeedlingState
ConcreteState	GrowingState
ConcreteState	DormantState
ConcreteState	MatureState
ConcreteState	SellingState
ConcreteState	DeadState
Context	Plant

Chain of responsibility (FR3.1)

Explanation:

- The requests, which are commands, will be able to move up the chain of command until one of the following classes (***DeliveryStaff***, ***Landscaper***, or ***GreenhouseManager***) can handle them. All handlers will use the handler class as their base class.

Structure:



Participants:

Participant	Class name
Handler	Handler
Concrete Handler	DeliveryStaff
Concrete Handler	Landscaper
Concrete Handler	GreenhouseManager
Client	Main

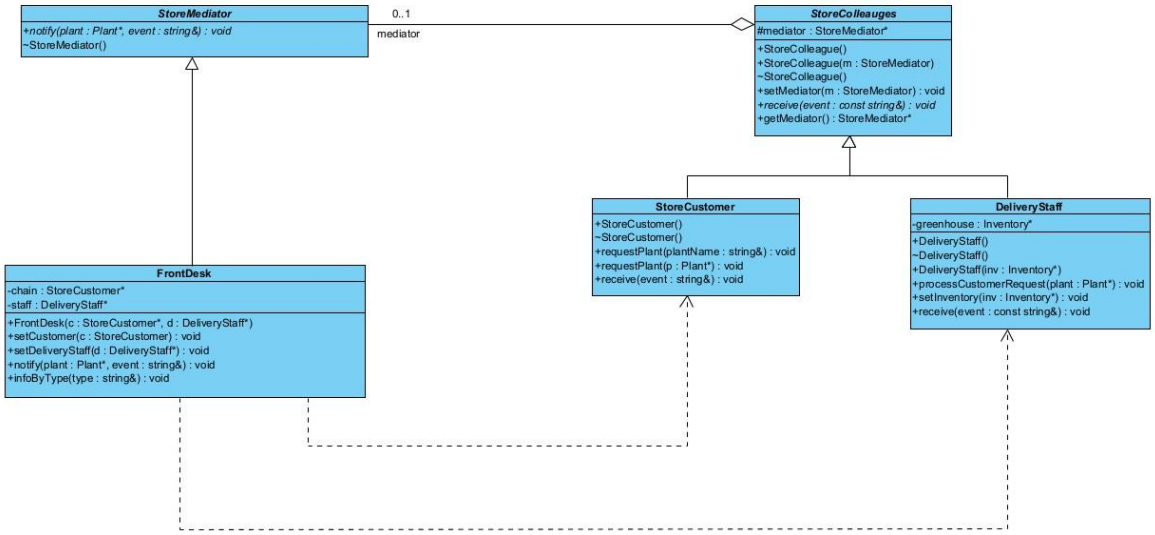
Mediator (FR2.2):

Explanation:

The front desk will serve as the cashier. When a customer asks the cashier for a plant, the cashier will ask the delivery staff where it is and use the plant iterator to check if it is there or

not. If it is, the cashier will inform the cashier and then the cashier will inform the customer that it has been delivered. Additionally, the customer will ask the cashier for information about cactus, rose or lavender. Mostly communication between the delivery staff and the customer.

Structure:



Participants:

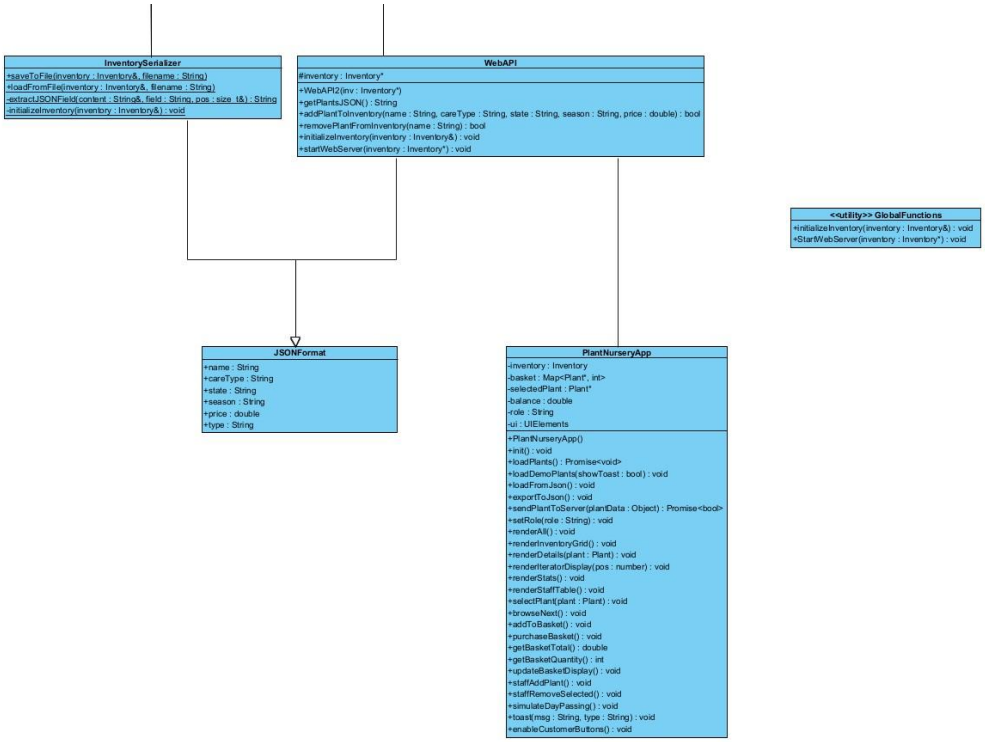
Participant	Class name
Mediator	StoreMediator
ConcreteMediator	FrontDesk
Colleague	StoreColleagues
ConcreteColleague	StoreCustomer
ConcreteColleague	Delivery Staff

Class Adapter (FR4.2):

Explanation:

The JSON Format will be used as an adapter that allows the inventory serializer and the web API to work together. Which enables the plant nursery app to use the web API.

Structure:



Participants:

Participant	Class name
Adapter	JSONFormat
Adaptee	InventorySerializer
Target	WebAPI
Client	PlantNurseryApp

Non – Functional Requirements

1.Security:

- NFR1.1: Only authorized employees can add or modify plants; clients are not permitted.
- NFR1.2: The Greenhouses' data (plants) will be restricted. Clients only have access to public plants listings or exhibitions.
- NFR1.3: Employees can get access to staff-only features using authentication(username and password).
- NFR1.4: Data about customers, staff, and plant inventory should be securely stored and protected from unauthorized access.

2.Usability:

- NFR2.1: Both customers and employees will find the design to be seamless.
- NFR2.2: When a system error occurs, the users should receive clear instructions and feedback to solve the issue.

3.Scalability:

- NFR3.1: The system should be able to handle any increase in the numbers of plants, staff and customers without resulting in a performance decrease.

4.Performance:

- NFR4.1: If the consumer makes a mistake, like inputting the incorrect password, the response will be returned to him.
- NFR4.2: The client will be able to view the update after the data is added from the staff without delays.

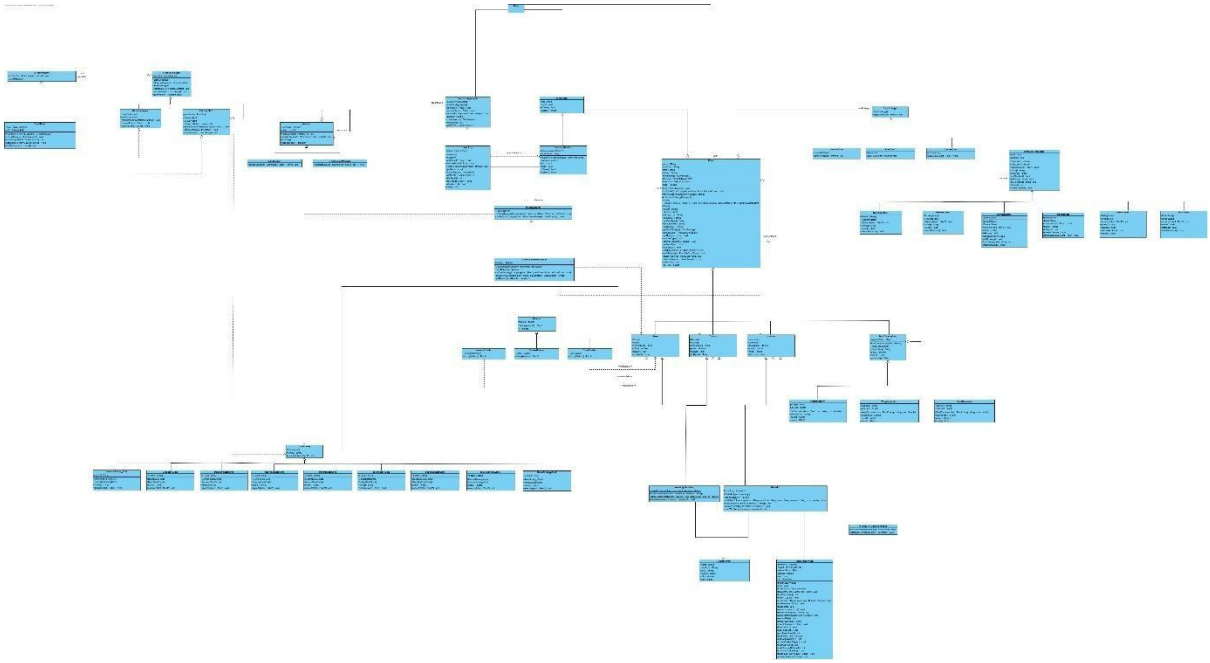
5.Maintainability:

- NFR5.1: For them to know whether to add or remove a function in the event of a merge dispute, we will be documenting our code.
- NFR5.2: In our project, unit testing will be used. Every individual will conduct their own unit testing.

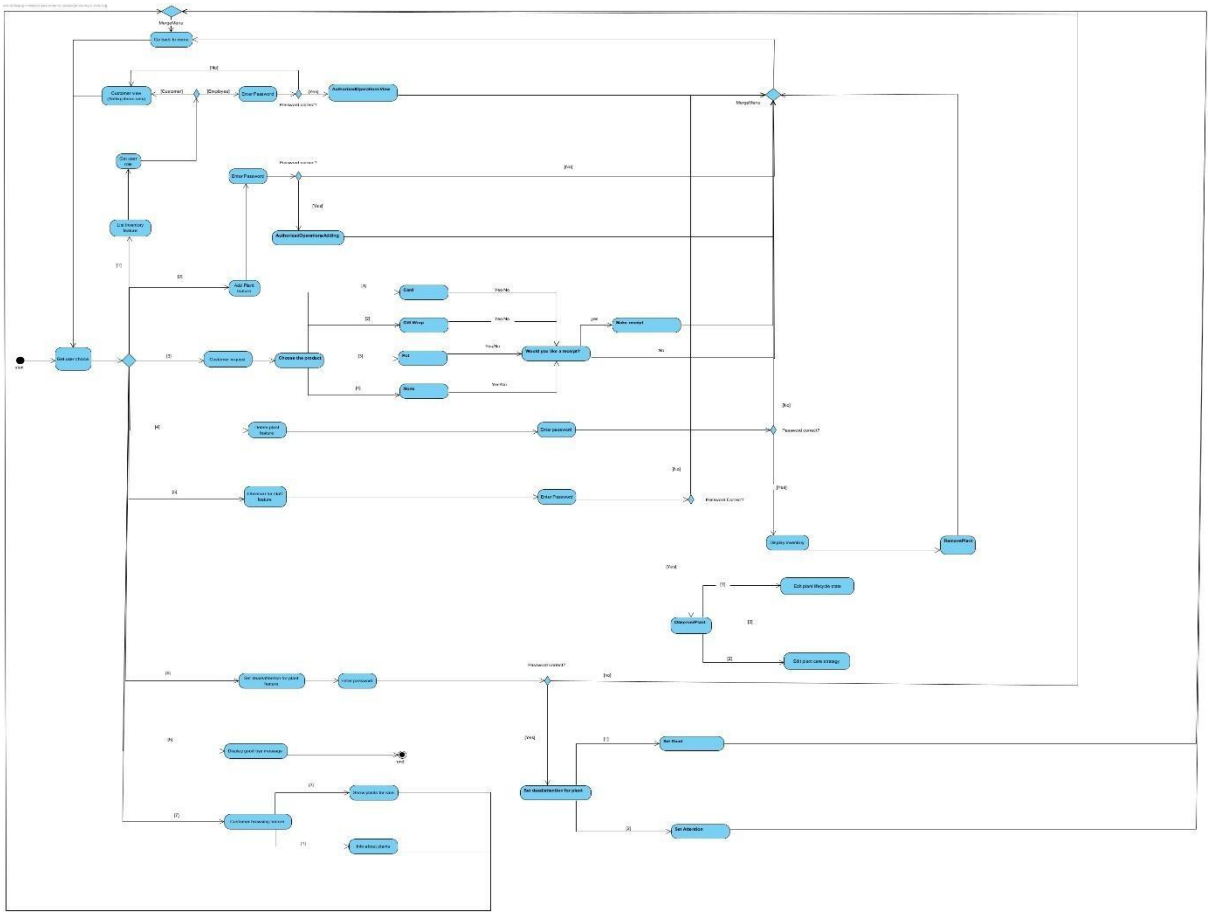
6.GUI:

- NFR6.1: Our graphical user interface shall be responsive, attractive and consistent across all supported devices .
- NFR6.2: The communication between the backend and the frontend should be smooth and stable.

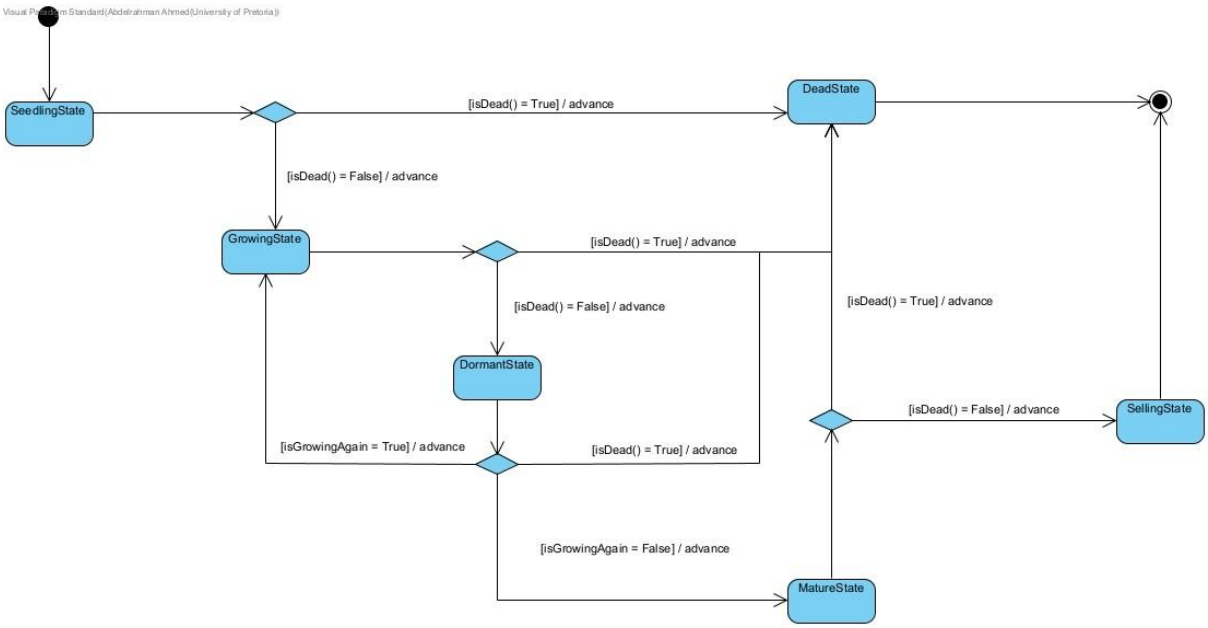
System Class Diagram



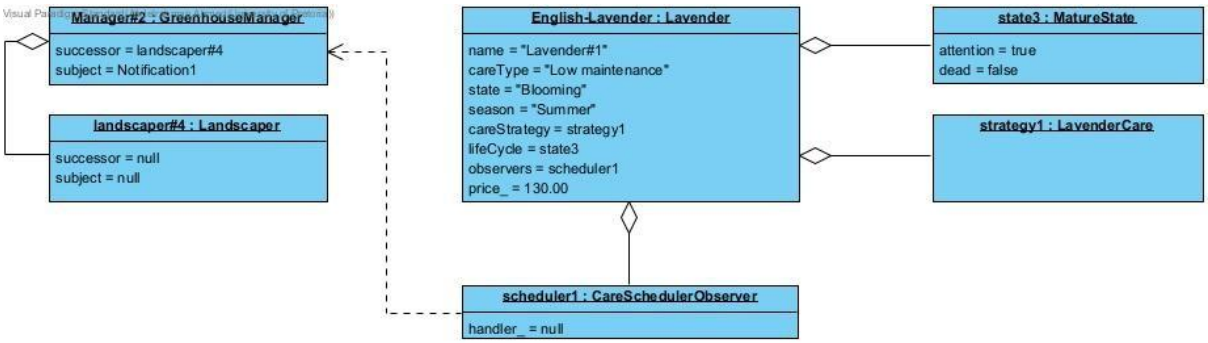
System Activity Diagram



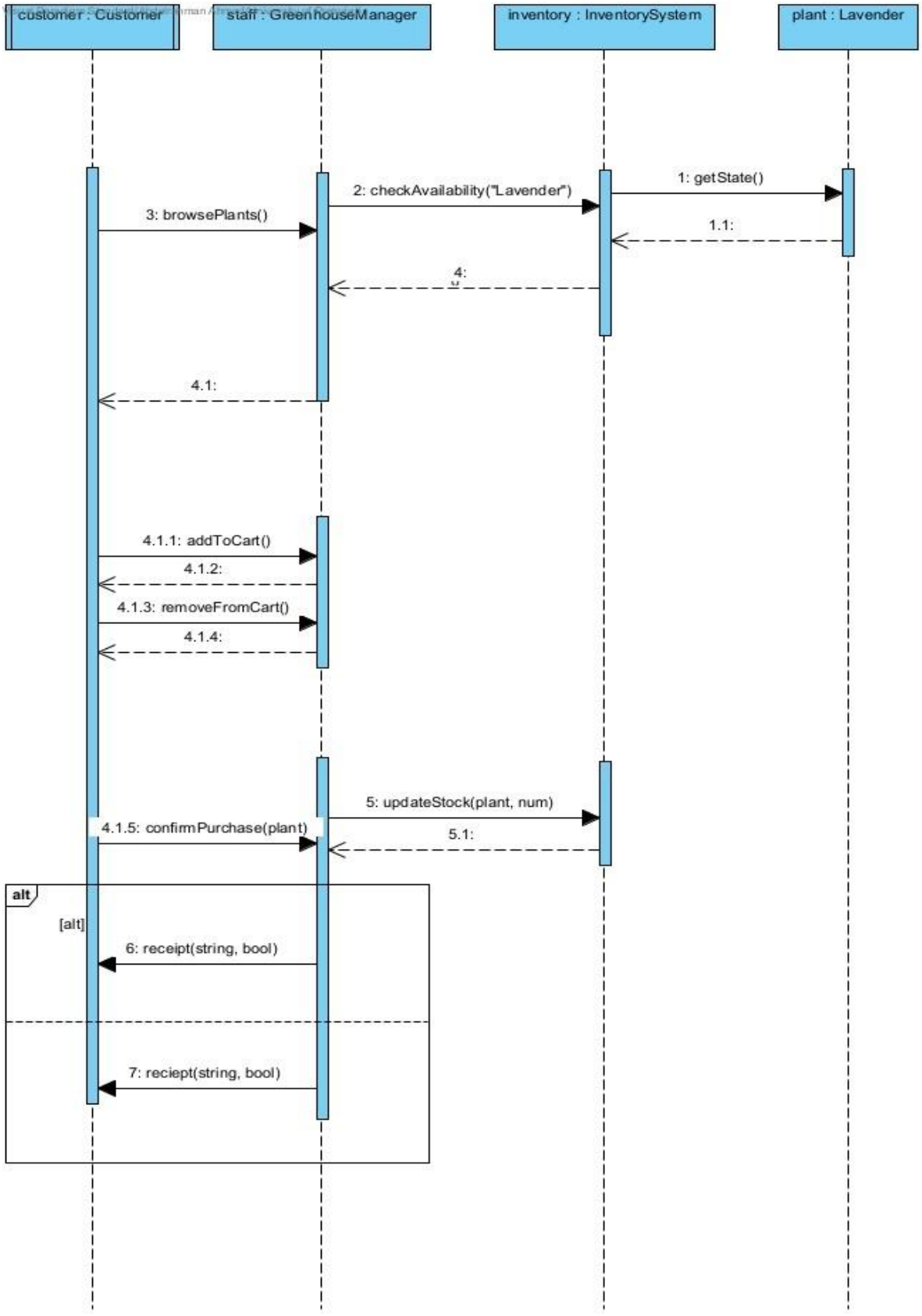
Plant State Diagram



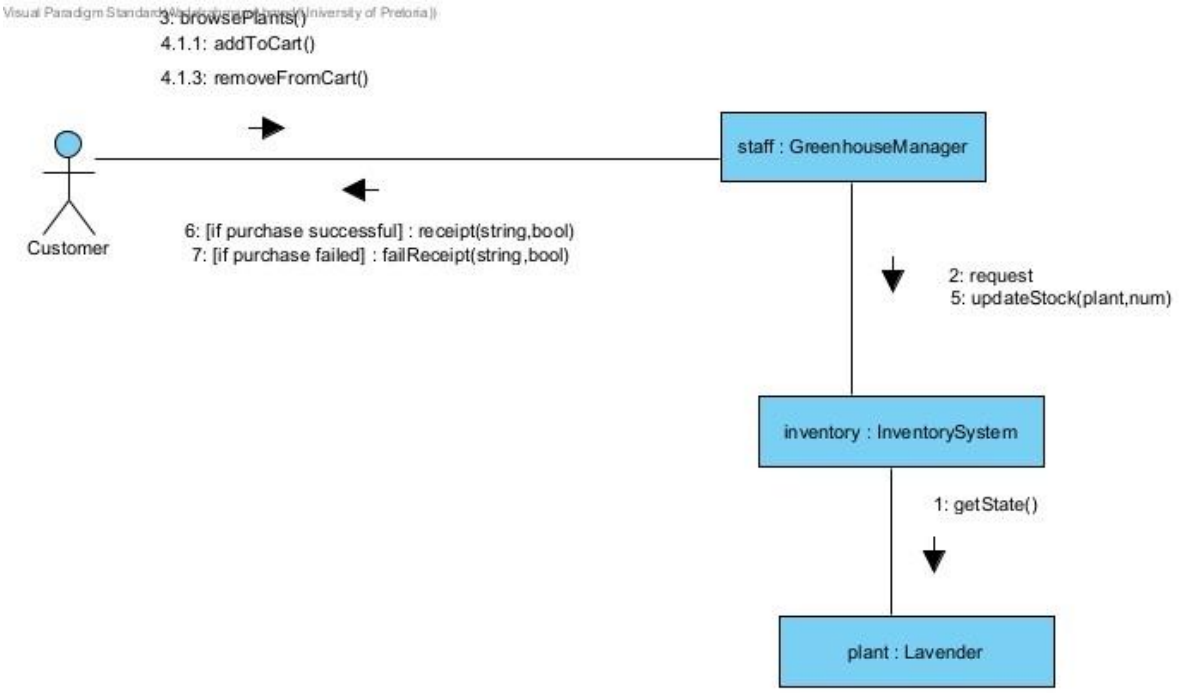
Plant Object Diagram



Purchase Sequence Diagram



Purchase Communication Diagram



Version Control with GIT

- A **secondary_main** branch was used to ensure everything is working before merging to **main**. Each member developed their own design pattern independently. Afterwards, all patterns were merged into the **Unit_Testing** for testing together. When patterns were linked, team members reviewed and added comments where needed. We fixed issues and improved code before final merging. Merge conflicts were resolved collaboratively.

Code Documentation

- Comprehensive documentation was produced using Doxygen. All the classes, methods and complex logic were explained. Both the header files and the source files were documented. General Information regarding the author, date and pattern involved were also included beside the code description.

Automated Unit Testing

- The **doctest** framework was used to implement the unit testing. We assigned the tests in a way that all design patterns had their own files in the 'testFile' folder. We performed thorough testing, though not 100%, all key functionalities were tested, such as state transitions in state design pattern, creation of products (Rose, Cactus, Lavender) and the necessary things for the other design patterns. All tests were successful ensuring that our system works as expected. Additionally, we used Valgrind to confirm there were no memory leaks present, ensuring we did memory management effectively. Lastly, all group members contributed in implementing the unit tests.

(Bonus) GitHub Actions Linter and Tester

- We implemented the CI pipelines using GitHub Actions to automate our projects building, linting and testing tasks. The pipeline is activated on both push and pull requests. This ensures that the code committed is valid before we merge our branches. The pipelines perform a few tasks sequentially which includes, installing dependencies, caching the build files, formatting all our code using clang-format, executing all our unit tests, ensuring no memory leaks in our unit tests and in our TestingMain. The CI pipelines automatically validate all of this, ensuring reliability and consistency throughout our project.

(Bonus) GUI

- A simple GUI (website) was implemented so that users can have a platform to interact with the nursery. It was implemented using html, CSS, and JS. The C++ engine was used as an API.

Assumptions

- The **inventory will already be filled** with plants before the system starts.
- Staff members will **check plant availability**, if the plant is in the inventory or not, **before selling** to customers
- The **Front Desk** will handle **customer interactions**, such as:
    - checking which plant/s the customer wants to buy ○ confirming if the plant is available in the inventory or not ○ sending the order to the correct staff member and the staff member handles it correctly (Chain of Responsibility)
- When a **delivery staff** member takes plants to a customer:
    - They will **ask the customer about decorations** or any custom preferences on the plant such as having a wrapped plant or having the pot decorated or writing a card.
- When a plant reaches the selling state, the plant will be **sold**. Thus, there is no need to check if the plant is dead or not.