DeBUG Nursery

COS 214 Final Project
Created by Code Blooded

Link to Google Docs version:
https://docs.google.com/document/d/1Ymhf6FpDXvqxfgE9jXweNniZir6k7LTdTUMqpMDIdqU/edit?tab=t.0#heading=h.2nivzy7rohzi
Link to GitHub repo:
https://github.com/Phantom124/CodeBlooded/

Code Blooded	
Name	Student Number
Jared	u24581039
Zaman	u24744931
Joshua	u24597092
Obed	u24595889
Ange	u24614484

Research

Plant nurseries are facilities where plants are planted, grown, and cared for until they are sold to customers. The primary focus of nurseries is to provide healthy plants while meeting customer demands for a variety of species.

Plant Care in Nurseries

One of the key functions of nurseries is to manage the conditions required for optimal plant growth carefully. This includes:

- Propagation: Plants are started from seeds, cuttings, or grafting. This stage involves close monitoring of moisture, temperature, and light to ensure successful germination or rooting.
- Soil and Nutrients: Nurseries use well-prepared soil mixes or growing media, often customised by plant type. Fertilisation schedules are designed to supply essential nutrients without overfeeding.
- Watering: Irrigation is carefully controlled through manual watering, sprinklers, or drip irrigation systems to maintain adequate moisture, preventing both drought stress and waterlogging.
- Pruning and Training: Plants are pruned to encourage healthy growth and shaped for aesthetic appeal or to meet size requirements.
- Environmental Controls: Shade cloths, greenhouses, or hoop houses provide protection from extreme weather and allow year-round production for some plants.
- Acclimatisation: Before sale, plants may be gradually exposed to outdoor conditions to harden them off, ensuring better survival after transplanting.

Customer Experience and Purchasing Process

Plant nurseries offer customers a tactile, informative shopping experience:

- Range of Plants: Nurseries stock a diverse array of plants, including annuals, perennials, shrubs, trees, vegetables, and indoor plants, often arranged by category or growing needs.
- Plant Health Assurance: Plants are visually inspected and sold in healthy condition; some nurseries provide guarantees or replacement policies if plants fail to thrive shortly after purchase.
- Purchase Options: Customers can buy plants in pots, trays, or bare-root form, depending on the species and season. Some nurseries offer bulk discounts or wholesale options for landscapers and gardeners.

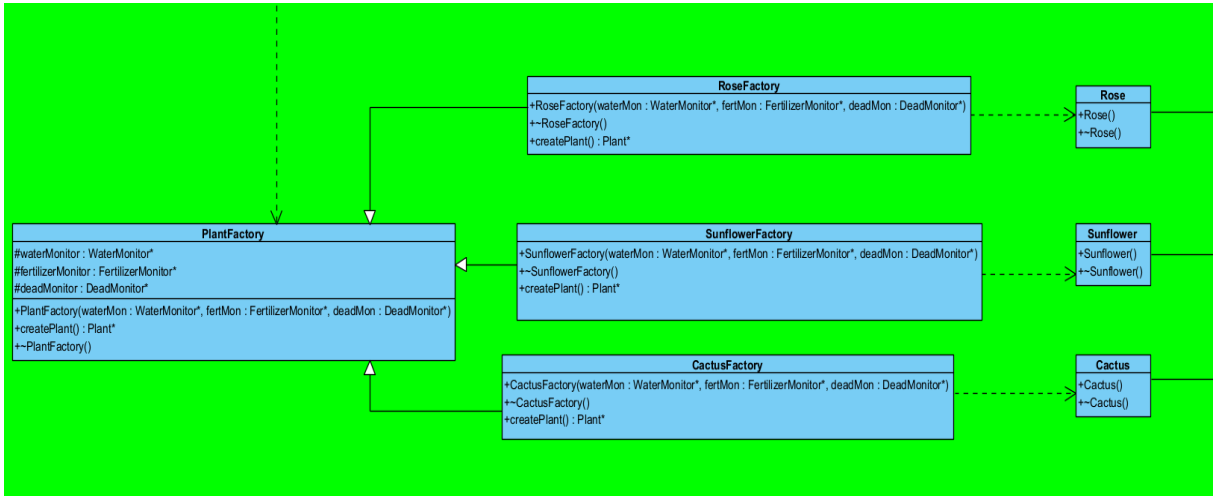
This research informed our system's architecture. For example, since nurseries depend on environmental monitoring, we implemented the **Observer** pattern to model plant health monitoring. The need for controlled access to inventory inspired our **Proxy** pattern. The cyclical nature of plant growth motivated our use of the **State** pattern to capture different growth phases dynamically.

Design Patterns

Design Pattern	Classification
Factory Method	Creational
State	Behavioural
Decorator	Structural
Observer	Behavioural
Command	Behavioural
Chain of Responsibility	Behavioural
Iterator	Behavioral
Builder	Creational
Proxy	Structural
Composite	Structural
Strategy	Behavioral
Facade	Structural
Memento	Behavioral

Behavioural: 7
Structural: 4
Creational: 2
Total: 13

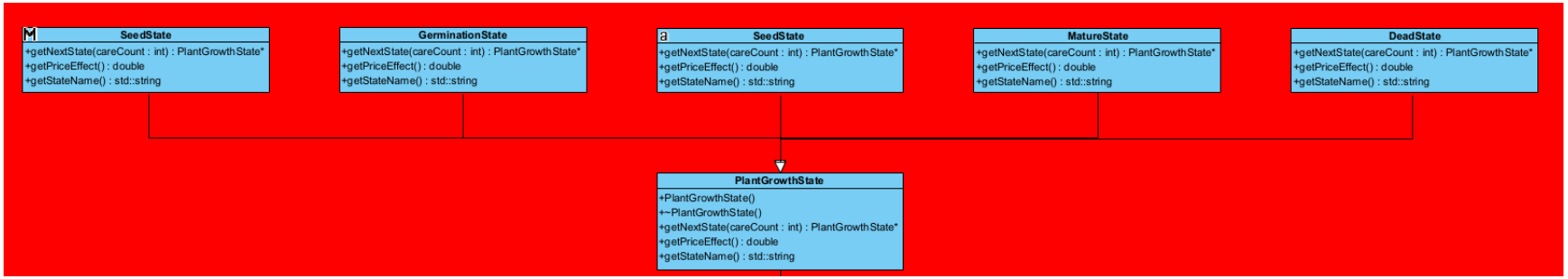
1. Factory Method

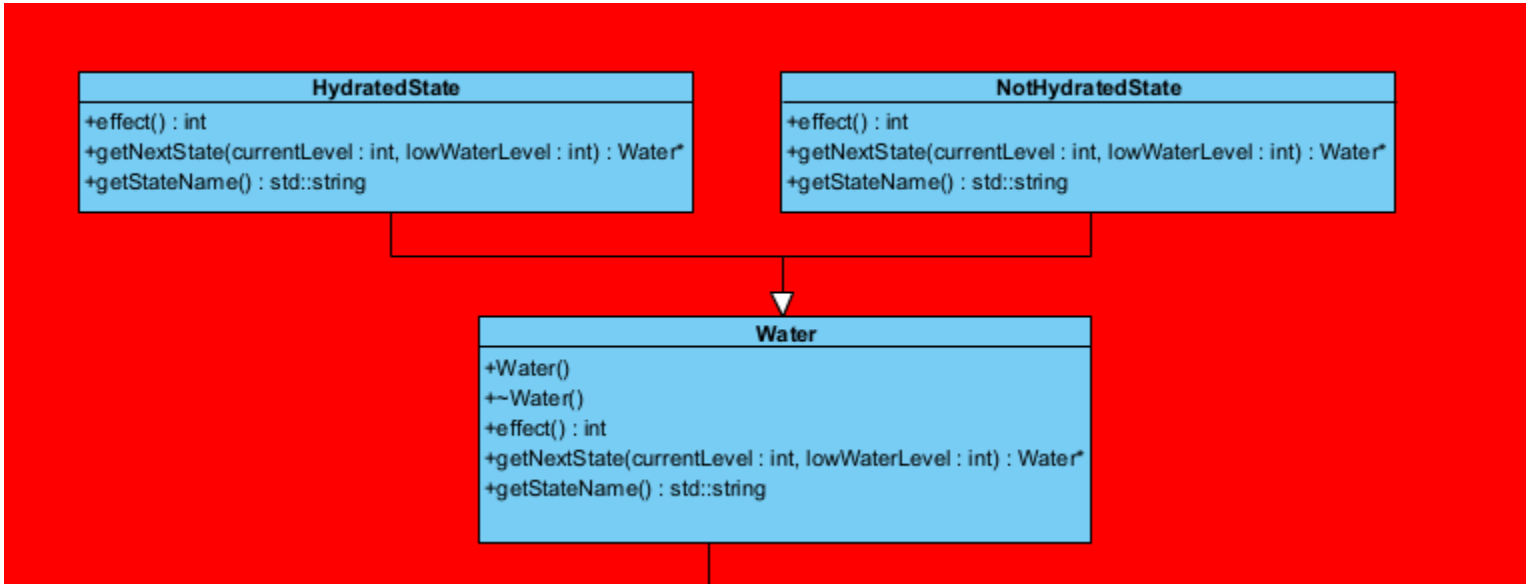


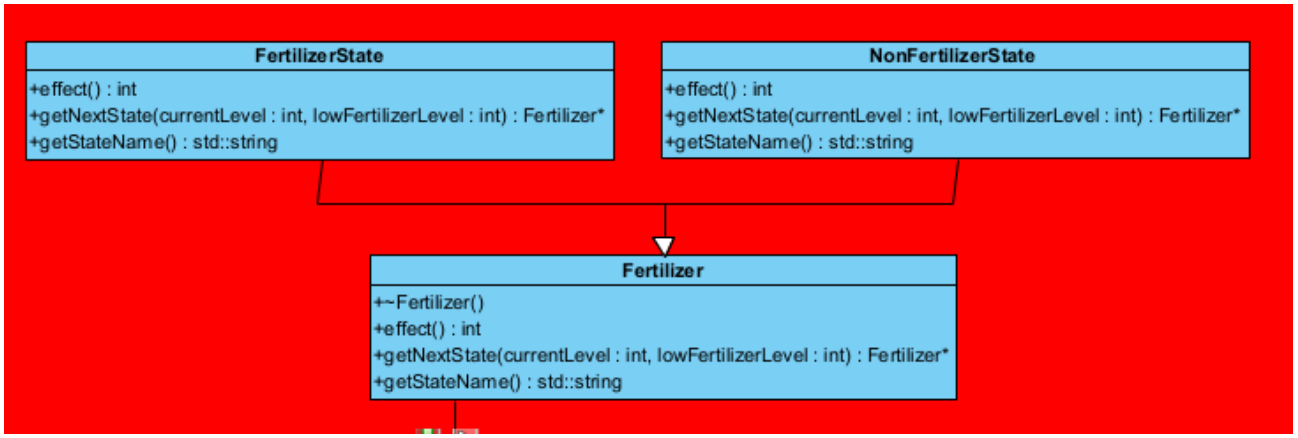
Why: The Factory Method was used in this system to implement the initial process of creating a plant.Since all plants are created at the exact same point, and all the specific plants have the same attributes; we use a Factory to create the plants. This avoids repetitive coding.
Where: In the start of the system when we are creating plants.
How: In the constructor of the Factory, we passed in the monitors for the plants such that the Factory can dynamically assign those monitors to the plants.
Participants:

- Creator: PlantFactory
- ConcreteCreator: RoseFactory, SunflowerFactory, CactusFactory
- Product: Plant
- ConcreteProduct: Rose, Sunflower, Cactus

2. State


SeedState
+getNextState(careCount : int) : PlantGrowthState*
+getPriceEffect() : double
+getStateName() : std::string
GerminationState
+getNextState(careCount : int) : PlantGrowthState*
+getPriceEffect() : double
+getStateName() : std::string
SeedState
+getNextState(careCount : int) : PlantGrowthState*
+getPriceEffect() : double
+getStateName() : std::string
MatureState
+getNextState(careCount : int) : PlantGrowthState*
+getPriceEffect() : double
+getStateName() : std::string
DeadState
+getNextState(careCount : int) : PlantGrowthState*
+getPriceEffect() : double
+getStateName() : std::string
PlantGrowthState
+PlantGrowthState()
+~PlantGrowthState()
+getNextState(careCount : int) : PlantGrowthState*
+getPriceEffect() : double
+getStateName() : std::string


HydratedState
+effect() : int
+getNextState(currentLevel : int, lowWaterLevel : int) : Water*
+getStateName() : std::string
NotHydratedState
+effect() : int
+getNextState(currentLevel : int, lowWaterLevel : int) : Water*
+getStateName() : std::string
Water
+Water()
+~Water()
+effect() : int
+getNextState(currentLevel : int, lowWaterLevel : int) : Water*
+getStateName() : std::string

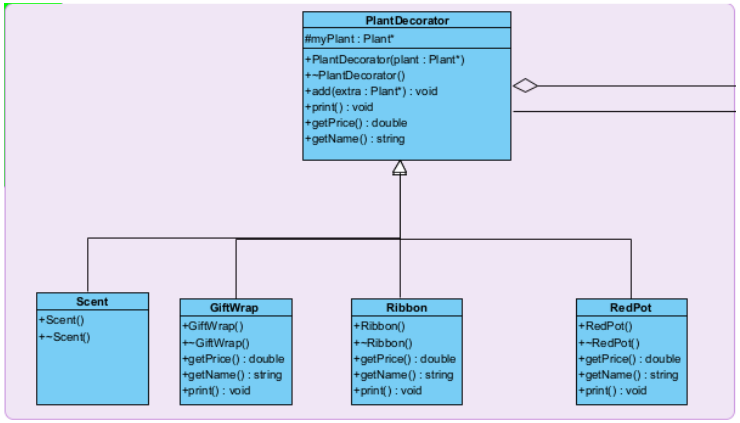


Why: The state design pattern allows us to alter the behaviour of the plant's growth and health depending on the state. It allowed us to easily manipulate how much the plant grew and lost health per cycle. It also allowed for us to easily determine the "hydration" and the "fertilizer" state per plant, thus making it extensible if the nursery were to add new plants.
Where: Inside each of the plant objects.
How: We allowed the plants to pass in their current water and fertilizer levels to their respective states, and allowed the states to handle which state the plant should be in. When time elapsed we took information from the states to tell us what the effect on the plant's health and growth was for that time cycle.
Participants:

Context: Plant
We have three state hierarchies

1. *PlantGrowthState*
    a. State: PlantGrowthState
    b. ConcreteState: SeedState, GerminationState, SaplingState, MatureState, DeadState
2. *Water*
    a. State: Water
    b. ConcreteState: HydratedState, NotHydrated
3. *Fertilizer*
    a. State: Fertilizer
    b. ConcreteState: FertilizedState, NonFertilizedState

3. Decorator

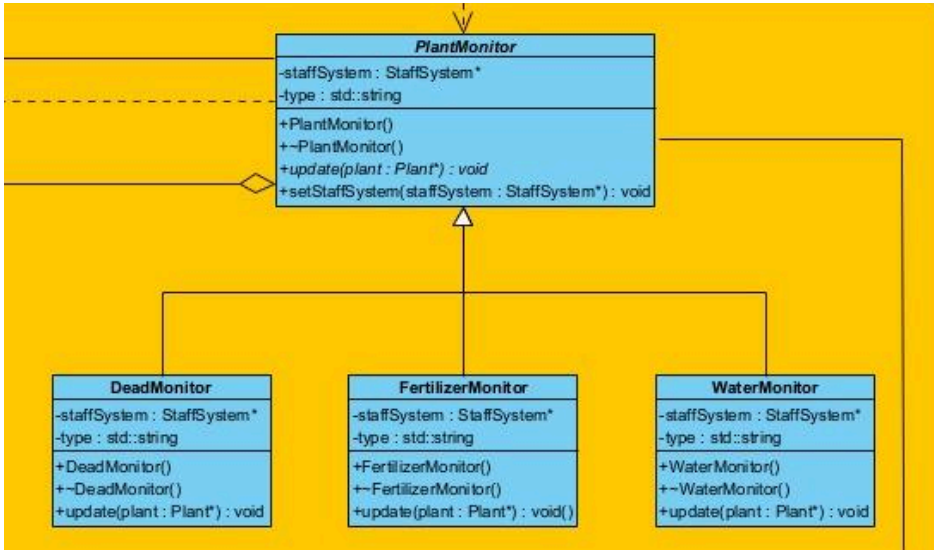


Why: The Decorator pattern lets us add decorations to plants without making many subclasses for every combination. We just wrap decorators around the base plant. This makes it easy to add new decorations later without changing existing code.
Where: On-top of the plants to add customizations.
How: Both Plant and PlantDecorator inherit from PlantComponent. Each decorator wraps around a PlantComponent (either a plant or another decorator). When you call getPrice, the decorator calls getPrice on whatever it's wrapping, adds its own cost, and returns the total. Same thing with getDescription, each decorator adds its name to the string.
Participants:

- Component: PlantComponent
- ConcreteComponent: Rose, Sunflower, Cactus
- Decorator: PlantDecorator
- ConcreteDecorator: Ribbon, RedPot, Scent, GiftWrap

---

4. Observer

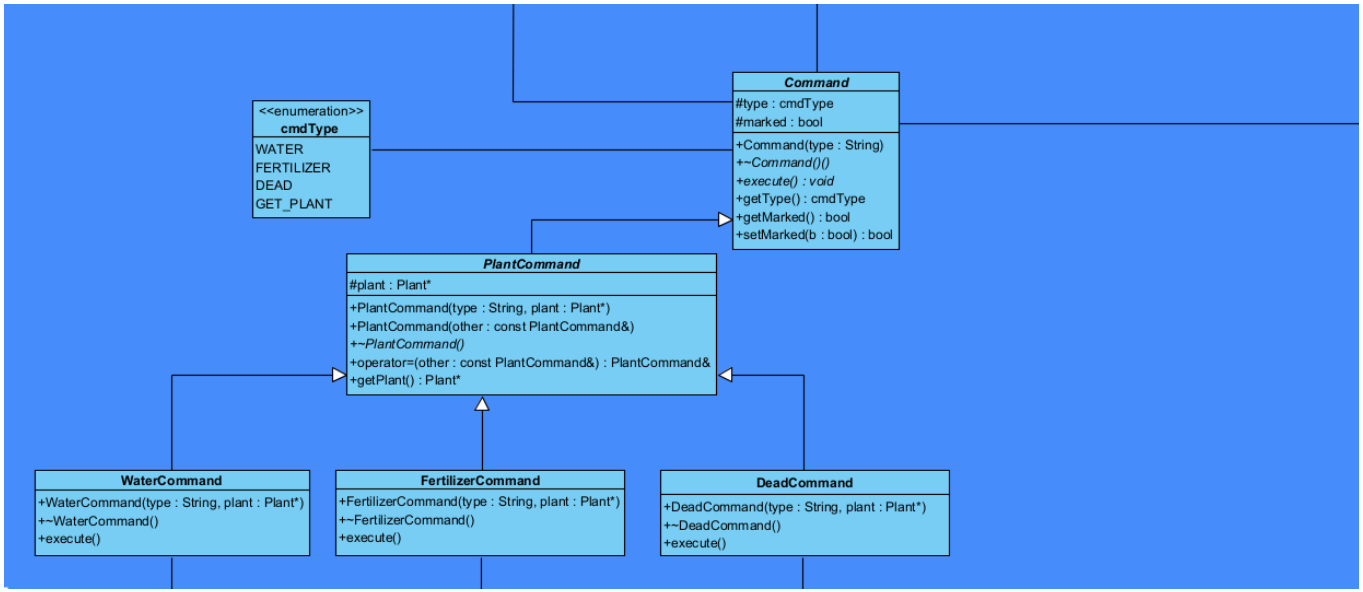


Why: The observer pattern makes our code more extensible when adding more factors to monitor the plant. It allows us to easily check the specific conditions of the plant in order to create the correct command to send to the staff system to be handled
Where: Monitoring the plant and sending commands.
How: When the plant's states go into a "negative" state (NotHydrated, NotFertilized) the plant notifies the specific monitor which the monitor then checks the respective state of the plant.
Participants:

- Subject: Plant
- ConcreteSubject: Rose, Sunflower, Cactus
- Observer: PlantMonitor
- ConcreteObserver: DeadMonitor, FertilizerMonitor, WaterMonitor

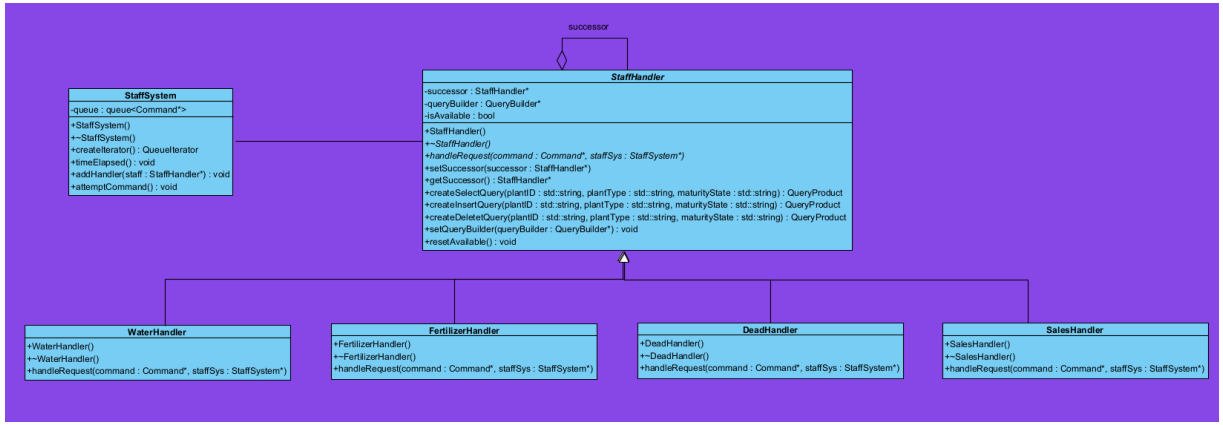
---

5. Command



Why: The Command design pattern allows the function of watering plants, giving them fertilizer or removing dead plants to be encapsulated in an object which can then be sent down the Chain of Responsibility and enqueued for later execution. The pattern also helps to make the system more extensible because to add new functions to the plant one would simply need to create a new class for that command.
Where: Used to communicate between plants and the staff handlers.
How: One of the PlantMonitor subclass objects will be notified when a plant undergoes a state change of the corresponding type and it will generate the command object and that will be sent to the staff system and handled. However, if no appropriate handler is found then the command object will be enqueued and will be passed through the chain of responsibility again at the start of the next clock cycle or whenever the user decides to pass them through.
Participants:

- Invoker: Customer, WaterMonitor, FertilizerMonitor, DeadMonitor
- Command: Command
- ConcreteCommand: WaterCommand, FertilizerCommand
- Receiver: Plant



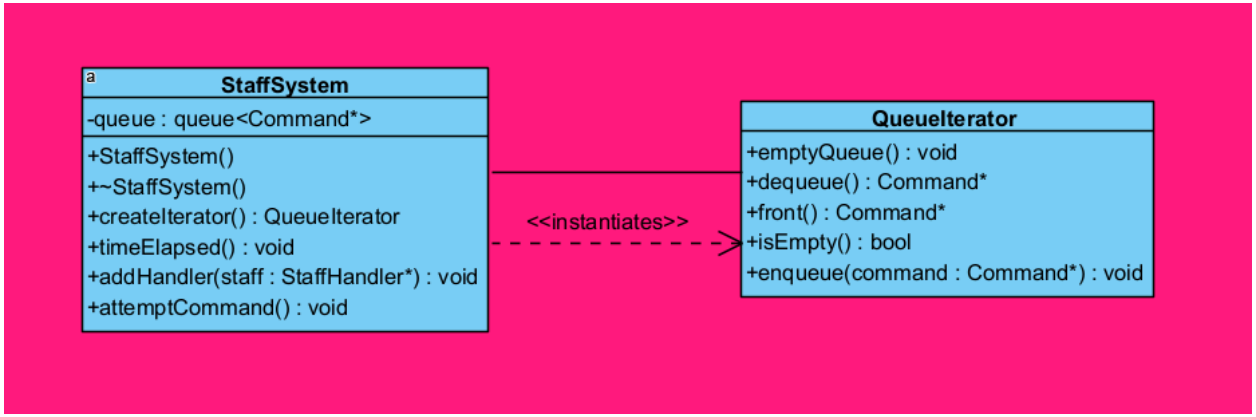
6. Chain of Responsibility

Why: The chain of responsibility allows us to pass commands in without knowing who will handle the command in advance. We considered using a mediator to send the command to all of the handlers, but we soon realised that that would not work because in the case when there are multiple instances of the same concrete handler class, they would have to somehow communicate about which one would complete the task which would be more complicated than just passing it down the chain until an appropriate handler was found.
Where: In the staff handlers in order to delegate tasks for the plants.
How: A command goes through the Chain of Command, and if the appropriate handler is found and the handler is available (i.e. not busy with some other command), the handler will then set isAvailable to false and handle the request. However, if no handler is found, the command will be placed in the queue in the StaffSystem object. The next time that timeElapsed() is called, all the commands in the queue will be sent through the chain again to be handled if any of the previously unavailable staff are now available.
Participants:

- Client: StaffSystem
- Handler: StaffHandler
- ConcreteHandler: WaterHandler, FertilizerHandler, DeadHandler, SalesHandler

---

7. Iterator

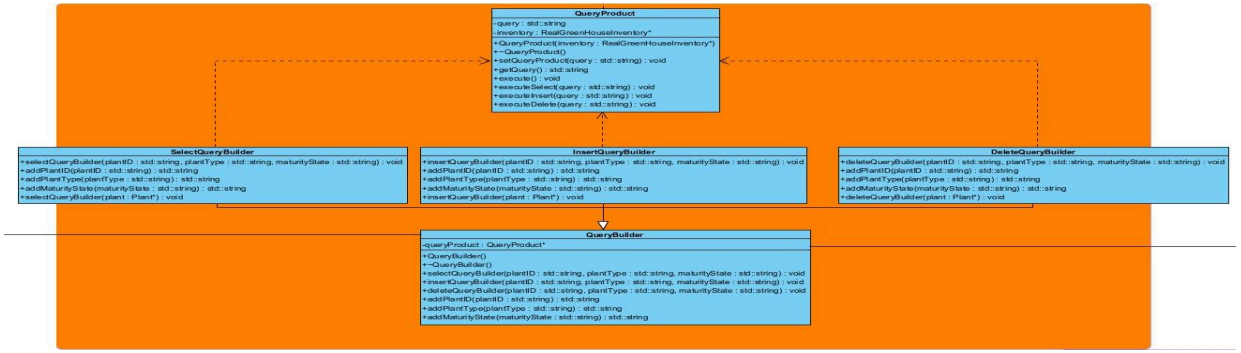


Why: The Iterator design pattern was chosen because we needed to move through the queue sequentially and be able to perform operations on the queue's elements easily, without having to expose the queue by making it a public attribute. A queue was used for fairness. We want the plant requests to be handled in the order that they came in.
Where: To handle the uncompleted tasks being queued inside the queue.
How: Whenever some operation needs to be performed on the queue, the createIterator() function will be called and an instance of the queue will be returned. The instance will be on the stack because it would be unnecessary and could cause memory leaks to place it on the heap just to delete it soon after. We decided to collapse the hierarchy because we do not see the need to extend the system to have multiple different aggregates.
Participants:

- ConcreteAggregate: StaffSystem
- ConcreteIterator: QueueIterator

---

8. Builder



Why: The Builder Design Pattern was implemented to construct complex query commands (such as SELECT, INSERT, AND DELETE) in a flexible and readable manner. In this project, users like customers or staff need to create SQL-like queries that target the greenhouse inventory. Instead of manually concatenating query strings – which can be error prone – the Builder Design Pattern modularizes query creation, ensuring each query follows a valid structure.

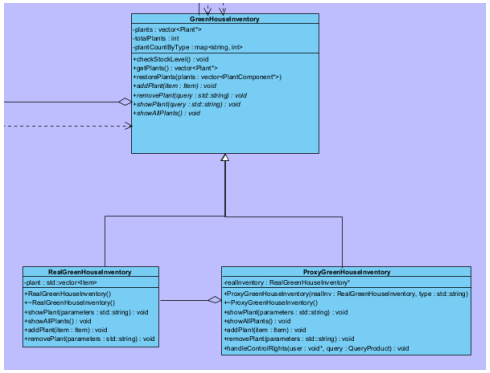
How: The Builder pattern separates the construction of a query from its representation.
The QueryBuilder class provides methods such as:

- selectQueryBuilder()
- insertQueryBuilder()
- deleteQueryBuilder()

Each method builds a structured command piece-by-piece (plant ID, type, maturity state), which is then passed to the QueryProduct for execution.
This pattern improves maintainability, allowing future query types (e.g., UPDATE) to be added without modifying existing code.
Participants:

- Director: Customer, StaffHandler
- Builder: QueryBuilder
- ConcreteBuilder: SelectQueryBuilder, InsertQueryBuilder, DeleteQueryBuilder
- Product: QueryProduct

9. Proxy



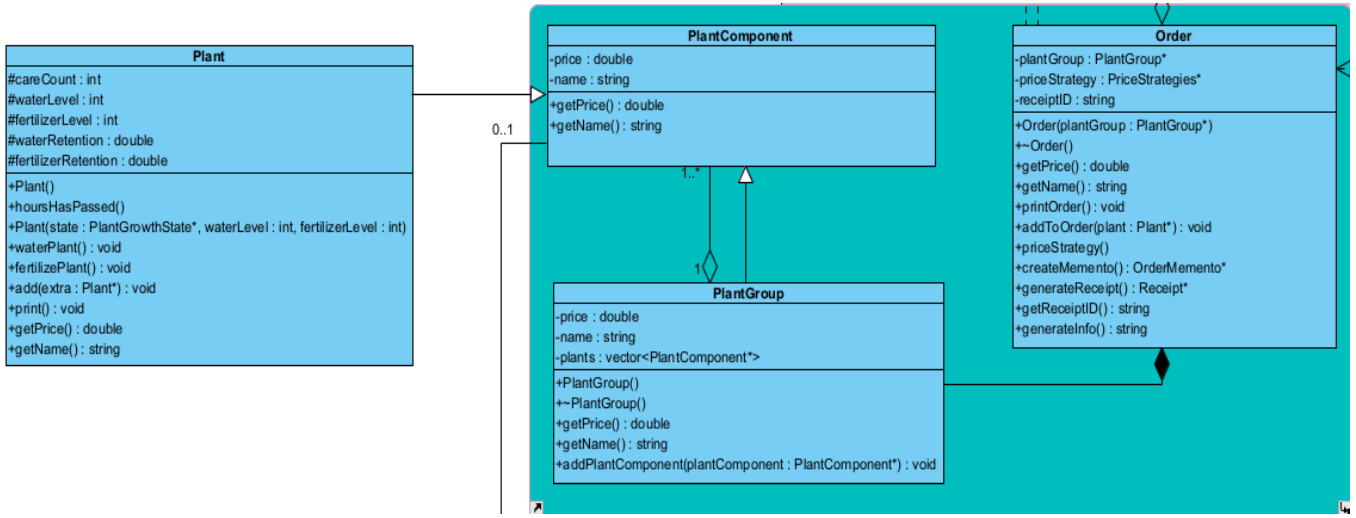
Why: The Proxy pattern was introduced to control access to the RealGreenHouseInventory.In a real-world nursery, customers should only be allowed to view plants, while staff members can insert or delete plants. The Proxy acts as a secure middle layer that enforces these access rights before any operation reaches the real inventory.

How: The ProxyGreenHouseInventory acts as a gatekeeper between the user and the RealGreenHouseInventory. When a query is executed, it passes through the proxy's handleControlRights() method. This method determines the user's role (Customer or Staff) using dynamic_cast, then decides: Customers → can only SELECT (view plants). Staff → can INSERT, DELETE, and SELECT. If an unauthorized action is attempted, the proxy prints a control access warning message instead of executing the command. This pattern ensures security, separation of concerns, and controlled access to the sensitive operations of the system.

Participants:

- Subject: GreenHouseInventory
- RealSubject: RealGreenHouseInventory
- Proxy: ProxyGreenHouseInventory

10. Composite



Why: The Composite pattern organizes everything in a customer order under one interface. Whether it's a single plant or a bunch of plants, they all work the same way. It handles nested structures well

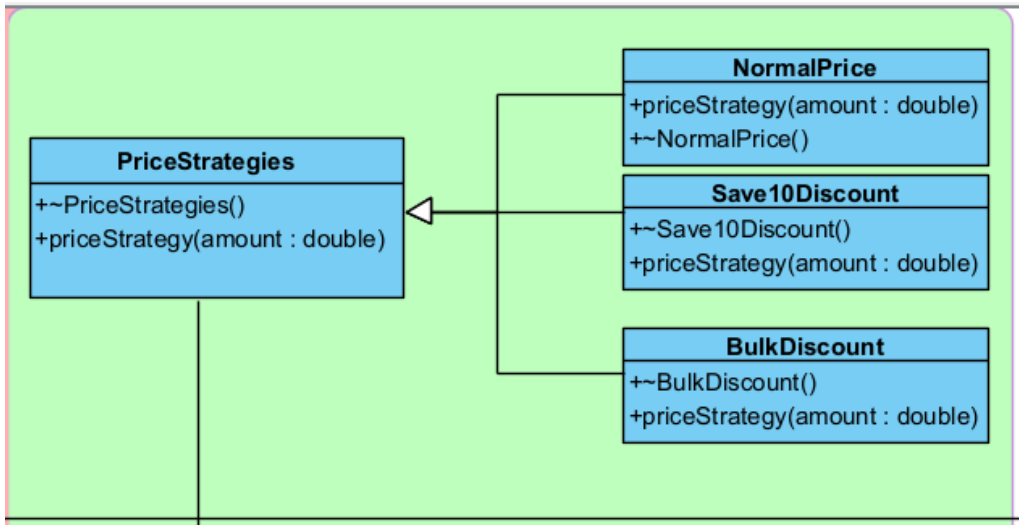
Where: In the order object to store all the plants.

How: We have a PlantComponent base class that defines what all components need to do. Individual plants and decorators are leaf nodes that return their own values. PlantGroup is the composite that holds a list of components. When you call getPrice on a PlantGroup, it loops through its children, calls getPrice on each one, and adds everything up. This works recursively, so groups can contain other groups.

Participants:

- Component: PlantComponent
- Leaf: Plant, PlantDecorator (Ribbon, RedPot, Scent)
- Composite: PlantGroup
- Client: Order class and test code

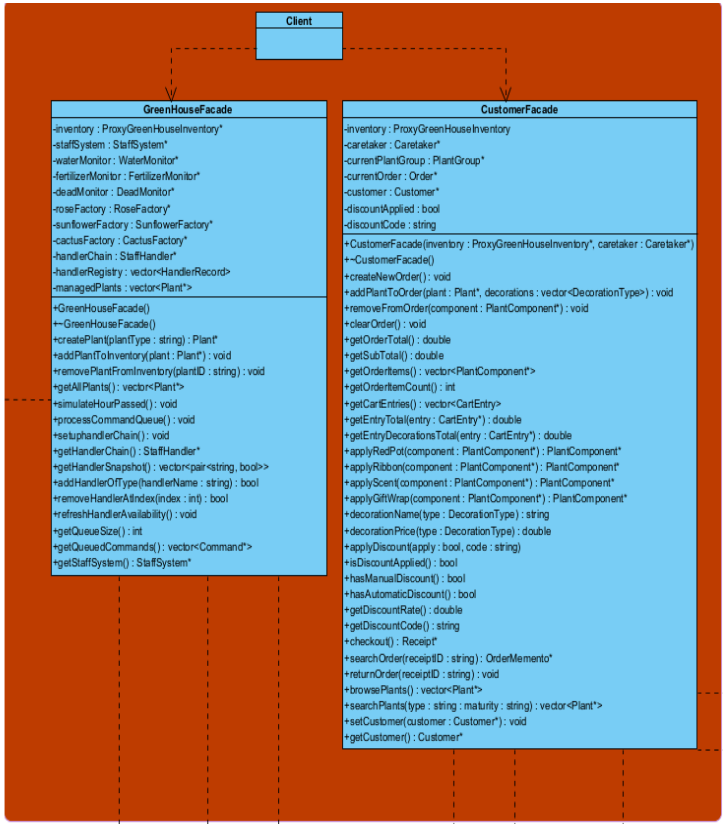
11. Strategy



Why: The Strategy pattern lets us handle different pricing options for orders without stuffing all the pricing logic into the Order class. We can easily switch between normal pricing and discounts if the customer has a coupon code that they redeem..
Where: Price Strategy into the order class.
How: We made a PriceStrategies base class that all pricing strategies follow. The Order class keeps track of which strategy it's using. When we need the final price, the Order calculates the total from all the items, then based on the number the right strategy is chosen. Switching strategies is made simple, just call setPriceStrategy with a different one.
Participants:

- Context: Order
- Strategy: PriceStrategies
- ConcreteStrategy: NormalPrice, Save10Discount, BulkDiscount

12. Facade

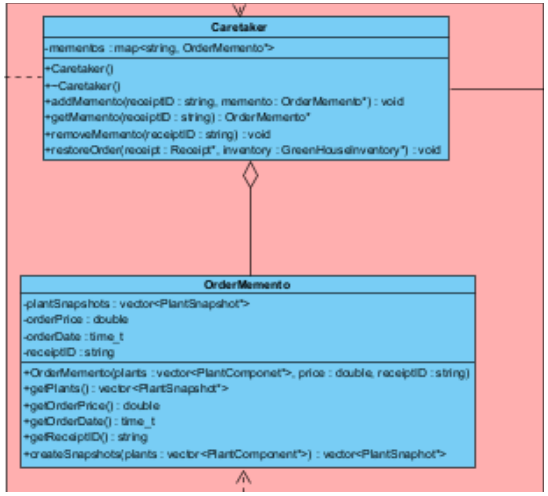


Why: The Facade design pattern is used to provide a simplified and unified interface for both customer interactions and greenhouse management. This reduces system complexity, improves usability, and decouples clients from the underlying subsystems.
Where: Used by GUI on startup.
How: The CustomerFacade manages all customer-related operations such as creating orders, managing carts, applying decoration costs and discounts, processing checkouts, and handling order returns. It communicates with the ProxyGreenHouseInventory to access and update plant stock.
On the management side, the GreenHouseFacade coordinates more complex internal operations by encapsulating plant factories, plant monitors, staff handlers, and command processing. This allows staff to manage greenhouse activities without needing to directly interact with multiple subsystem components.

Participants:

- **Facade:** CustomerFacade, GreenHouseFacade
- **Subsystem classes**: ProxyGreenHouseInventory, Order, PlantGroup, PriceStrategies, StaffSystem, StaffHandler, PlantMonitor, PlantFactory

13. Memento



Why: The Memento design pattern is used to preserve the state of an order so that the plants from returned purchases can be accurately restored to the greenhouse inventory. This ensures data consistency and allows the system to revert to a previous state without violating encapsulation.
Where: When an order has been completed and when a customer returns an order.
How: The Order class is responsible for saving the state of all plants in an order as a memento, which is stored in an OrderMemento object. These mementos are managed by the Caretaker, which also keeps track of the receipt ID associated with each saved state. When a customer returns an order, the CustomerFacade retrieves the saved state linked to that receipt. Before proceeding, it verifies that the receipt is still within the acceptable return period. If the return is valid, the Caretaker restores the corresponding plant data from the saved state and places the plants back into the GreenHouseInventory.

Participants:

- **Originator:** Order
- **Memento**: OrderMemento
- **Caretaker**: Caretaker

Functional and Non-Functional Requirements

Functional Requirements

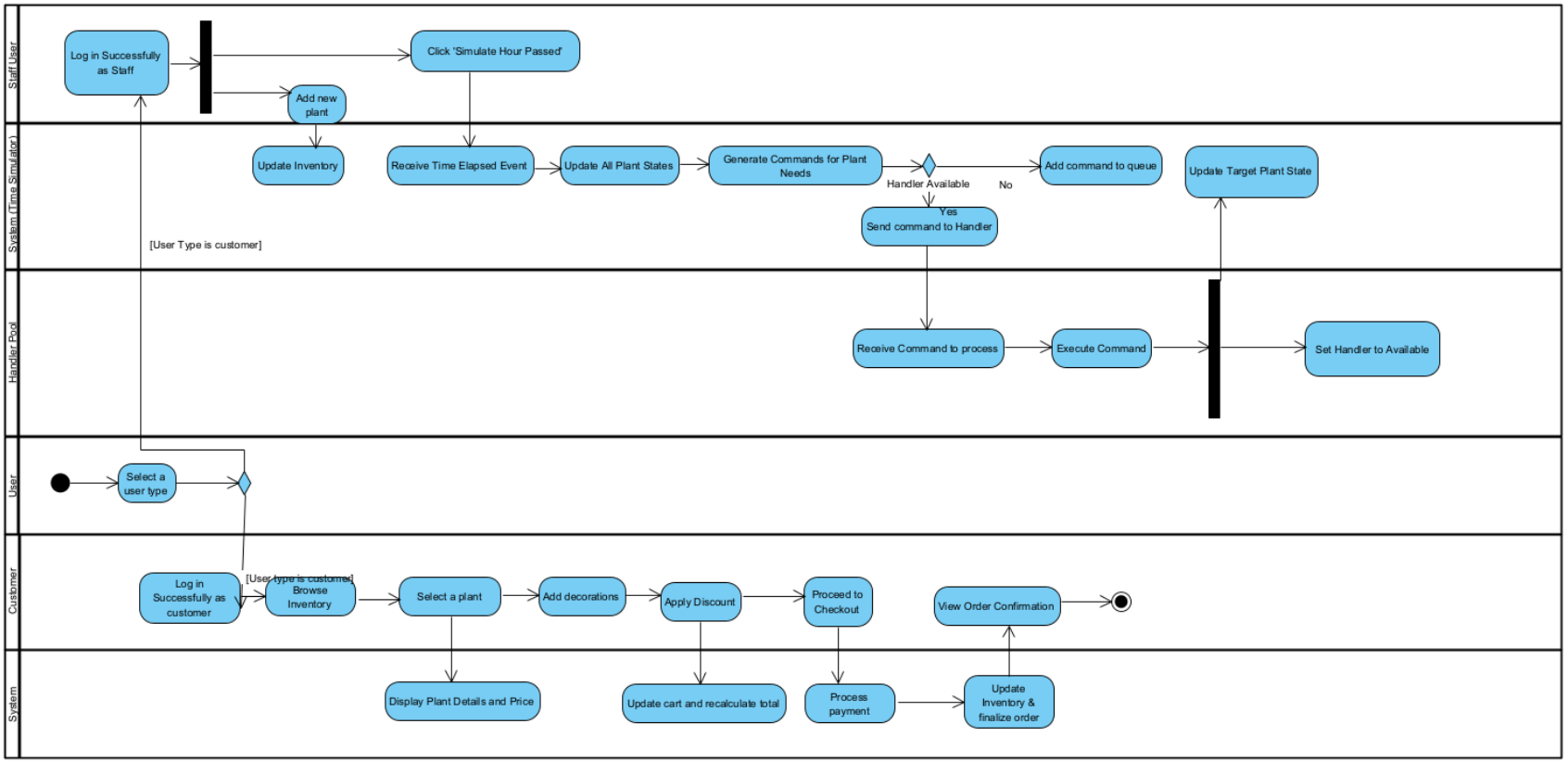
Pattern	Subsystem / Class	Functional Requirements
Factory Method	PlantFactory	The system will create new plant objects using the appropriate factory.
State	Plant, PlantGrowthState	System will transition a plant's state over time
Decorator	PlantDecorator	The system will allow the user to decorate plants.
Observer	PlantMonitor	Monitor each plant's hydration and fertilization and send alerts.
Command	PlantCommand	The system will handle plant commands.
Chain of Responsibility	StaffHandler	System delegates tasks to appropriate staff handlers.
Iterator	StaffSystem	System will iterate through queued tasks in order
Builder	QueryBuilder	System will construct structured database.
Proxy	ProxyGreenHouseInventory	System will enforce user access rights through a secure interface
Composite	PlantGroup	System groups plants

		and apply operations recursively.
Strategy	Order	System will calculate pricing using different discount strategies
Facade	CustomerFacade, GreenHouseFacade	System will provide a unified interface for customer and staff interactions.
Memento	Order, Caretaker	The System will restore previous plant states for order returns.

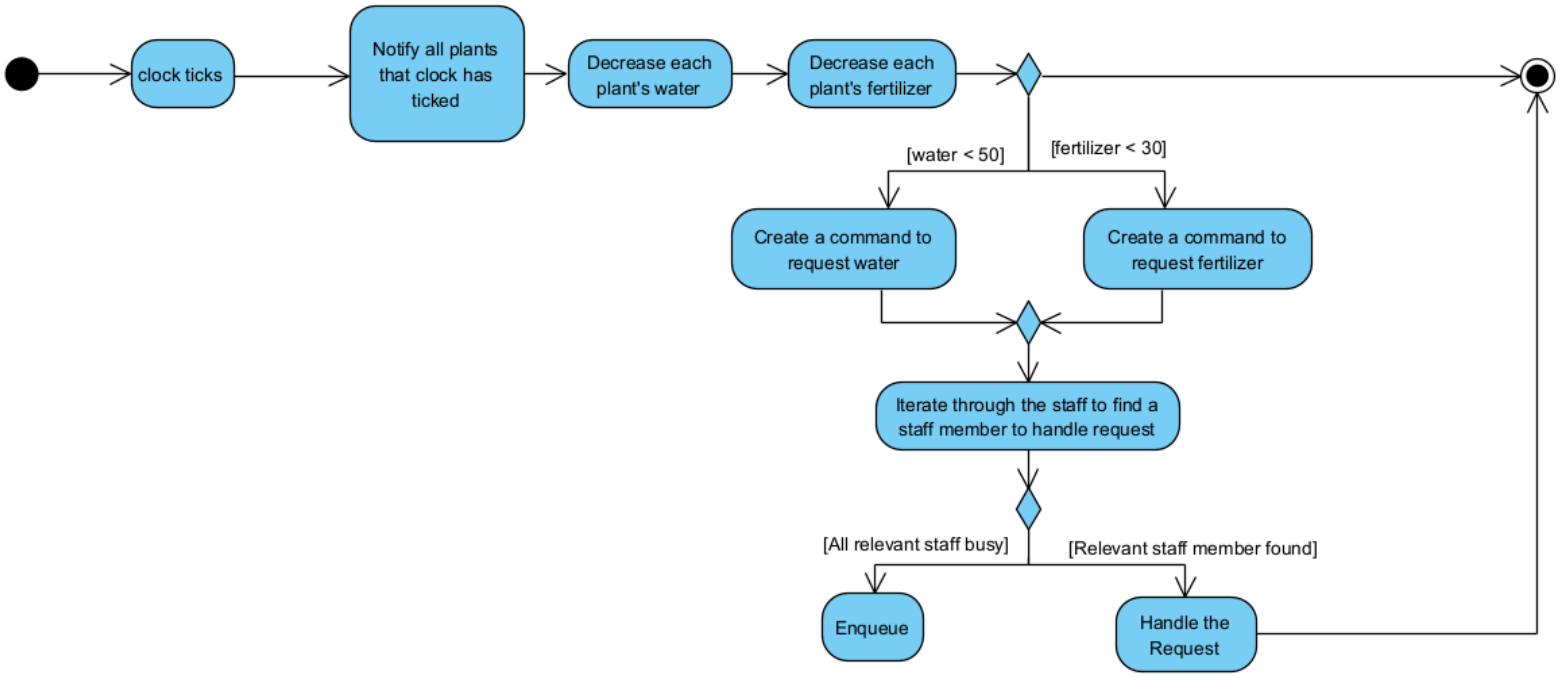
Non-Functional Requirements

Quality / Attribute	Requirements
Modularity	The system will maintain a modular architecture by assigning each class a single responsibility.
Extensibility	The system will allow new plant and command types to be added without altering existing code.
Reliability	The system will validate all pointer dereferences to prevent runtime crashes.
Testability	The system will use isolated unit tests for core subsystems (Factory, Proxy, State).
Maintainability	The system will include Doxygen documentation for all classes and methods.
Portability	The system will follow C++11 standards for compatibility across OS platforms.
Scalability	The system will be able to simulate at least 100 plants concurrently without performance degradation.

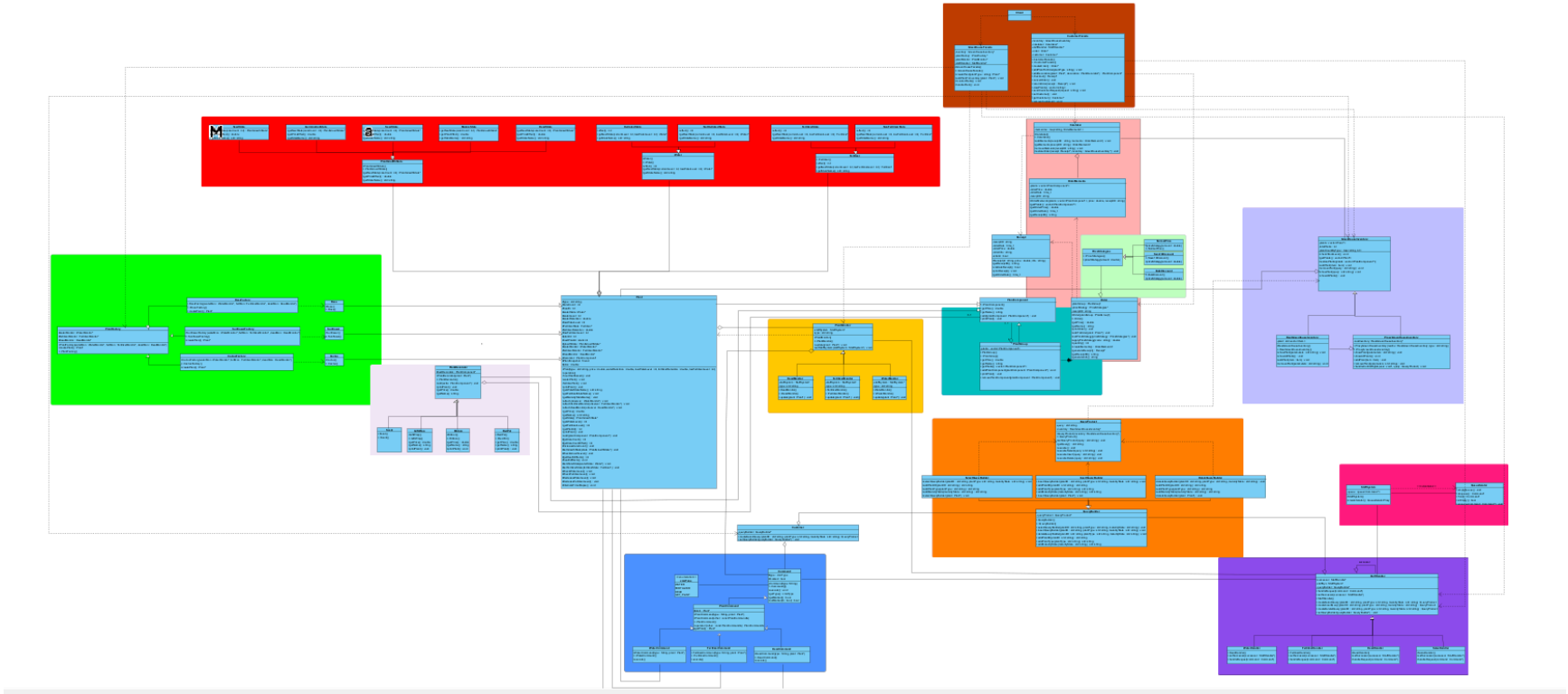
HIGH-LEVEL ACTIVITY DIAGRAM OF WHOLE SYSTEM:



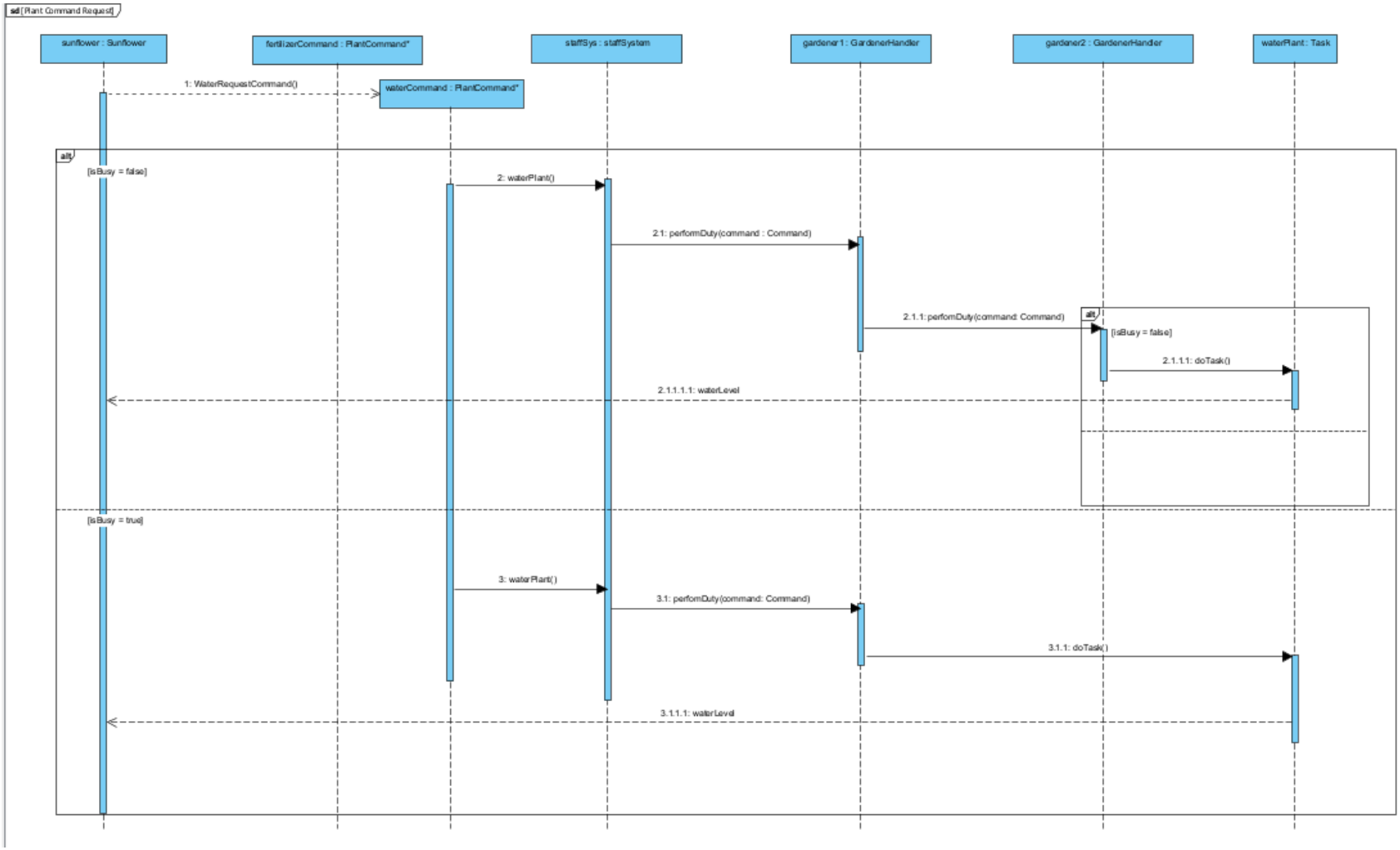
ACTIVITY DIAGRAM:



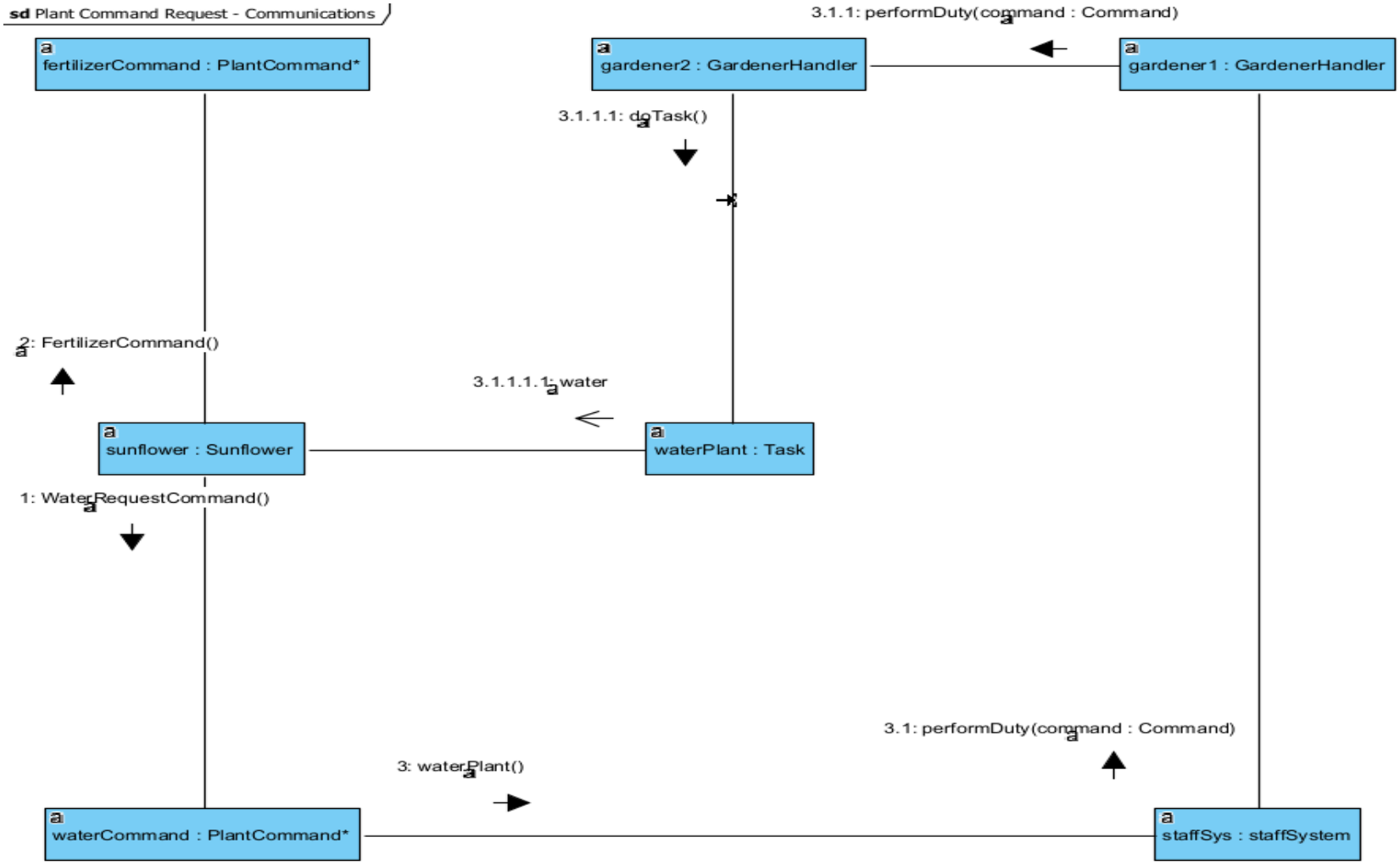
FULL CLASS DIAGRAM:



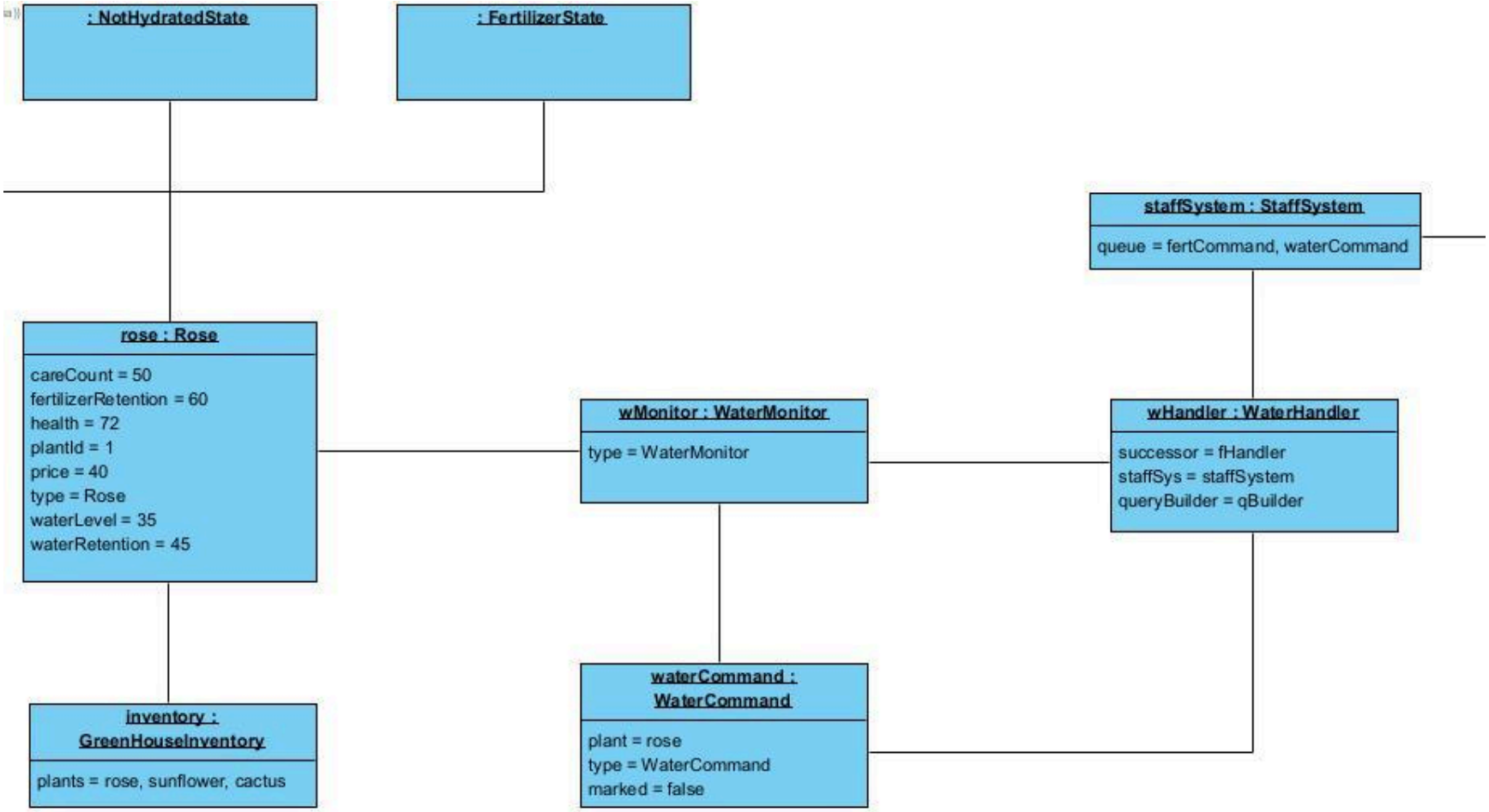
SEQUENCE DIAGRAM:



COMMUNICATION DIAGRAM:



OBJECT DIAGRAM:



STATE DIAGRAM