Project Team – Report

Git: Project Team Project Folder Git Link

Plant Nursery Operations and Software Architecture

The successful design of our Plant Nursery Simulator comes from extensive research into the operational complexities of a commercial nursery and the specific, heterogeneous care requirements of diverse plant biomes. This research was foundational in defining the functional requirements of our system and justifying the strategic implementation of object-oriented principles and software Design Patterns to achieve a flexible, maintainable, and scalable architecture. Running a nursery, as a dynamic, living system, requires meticulous attention to two core areas: **precise biological management** and **efficient logistical coordination**. Biologically, plants are highly differentiated entities; they are not commodities. Their fundamental needs—including **water, light, fertilizer, and soil pH**—are strictly governed by their native ecological environment. For example:

- **Desert Biome** plants (like succulents or cacti) are xerophytic; they require infrequent, deep watering and well-draining soil to prevent **root rot**, which is a rapid path to the WiltingState. Over-watering is often more lethal than under-watering.
- **Tropical Biome** plants, in contrast, demand consistently moist soil (though not saturated) and often thrive with regular, nutrient-rich fertilization during their growth cycles to maintain health.
- **Wetland/Aquatic Biome** plants require a specialized environment, tolerating or even requiring saturated soil conditions and specific potting media (like the AquaticBasketPot or PeatSoilMix) to ensure their survival and progression to the MatureState.

Ignoring these inherent biological differences leads to plant mortality and financial loss. Therefore, the simulator must dynamically adapt its care routines based on the plant's defined biome. This need for dynamic, interchangeable behaviour led directly to the implementation of the **Strategy Pattern** (CareStrategy), which decouples the care logic from the plant object itself, ensuring **extensibility** for future biomes. Logistically, a commercial nursery demands continuous, interconnected management across its lifecycle stages. Stock availability in the sales area is a direct and immediate result of plant health and maturation in the greenhouse. The transition of a plant from the GrowingState to the MatureState must instantaneously affect its sale status in the inventory. This complex, interdependent coordination—involving the monitoring of plants, the updating of inventory records, the assignment of staff to new tasks, and the fulfillment of customer orders—is managed through the strategic use of the **Mediator** and **Observer** patterns. These patterns were chosen specifically to **reduce coupling** and **centralize control flow**, thus providing the necessary organizational structure (the **NurseryHub**) to transform biological complexity into a coherent, object-oriented system.

This revision directly addresses the prompt by providing:

1. **Specific biological details:** Mentioning xerophytic nature, root rot, demand for moist soil, and specialized media.
2. **Explicit linkage to design decisions:** Tying the biological heterogeneity directly to the need for the **Strategy Pattern**.
3. **Enhanced operational context:** Expanding on the necessary link between the dynamic "living system" and the logistical coordination handled by the **Mediator/Observer** patterns.

1. Nursery Operational Model and System Design

A comprehensive nursery operates as a tightly coupled system of four primary domains: Plant Lifecycle Management (Greenhouse), Inventory & Stock Control, Staff Coordination, and Customer Sales & Interaction. Our software architecture mirrors this reality by employing design patterns to manage communication and complexity between these domains:

A. Inter-Domain Communication and Decoupling

- **Mediator Pattern:** The ChatMediator (Mediator) was implemented to manage and centralize communication between all users of the system, specifically Customer and Staff objects. This pattern ensures that users (Colleagues) do not interact directly with one another, but instead communicate through the Mediator, which enforces rules about which roles can message each other (e.g., Sales <-> Customer, Inventory <-> PlantCare). This reduces coupling, simplifies message routing, and improves system maintainability and flexibility.

- **Observer Pattern:** The system tracks events across SalesService, InventoryService, StaffDashboard, CustomerDashboard, and Greenhouse. InventoryService acts as a central observer, updating system state like stock levels. Dashboards react to events and publish updates to users, ensuring staff and customers see real-time changes. This design decouples event sources from listeners while keeping operations synchronized.

2. Plant Lifecycle Management and Biome-Specific Care

Effective plant care is paramount, as different species require vastly different approaches to watering, fertilizing, and environmental management.

A. Strategy Pattern for Plant Care

- **Research Justification:** Research confirms that care requirements are primarily dictated by a plant's native biome (e.g., desert, tropical, wetland). For instance, Desert plants require low, infrequent watering and excellent drainage to avoid fatal root rot, while Tropical plants demand consistent moisture and regular fertilization during active growth.
- **Design Application:** The Strategy Pattern provides a flexible solution to this highly variable requirement. A concrete Plant object delegates its care tasks (water(), fertilize(), sprayInsecticide()) to a specific CareStrategy object (e.g., DesertStrategy, WetlandStrategy). This implementation allows new biome-specific care routines to be introduced easily (fulfilling the extensibility requirement) without modifying the core Plant class, which remains focused on its state and attributes.

B. Factory and Flyweight for Efficient Plant Creation

- **Abstract Factory Pattern:** To maintain consistency in plant setup based on target biomes, a family of Abstract Factory classes (e.g., DesertFactory, TropicalFactory) is used. Each factory generates compatible components—a plant, its pot, and soil mix—along with a biome-specific CareStrategy (e.g., DesertFactory produces a plant with high water sensitivity, a TerracottaPot, and SandySoilMix). This design ensures that all components are correctly paired for the target environment, mirroring real-world planting guidelines and enforcing consistency across the system.
- **Flyweight Pattern:** Nurseries typically manage thousands of individual plants across a finite number of species. The Flyweight Pattern is used to reduce memory overhead by sharing intrinsic, common data across all instances of a species. The SpeciesFlyweight stores the constant attributes (SKU, Name, Biome, Base Cost), while the individual Plant objects maintain their unique, extrinsic state (e.g., health, moisture level, state). This directly addresses the non-functional requirement of scalability by optimizing memory usage.

3. Assumptions and Time Simulation

- **Time Model**: The simulation time is scaled such that 10 seconds of real-world time is equivalent to one simulated day of plant growth and operation. This allows for observable lifecycle progression and operational changes within the demo window.
- **Seasons**: Seasons are tracked using the host machine's time, and it is assumed that they directly influence the plant's growthRate parameter, reflecting the seasonal cycles of growth and dormancy found in nature.
- **Plant-Biome Mapping**: It is assumed that each plant is uniquely and permanently associated with a single biome, which entirely determines its optimal CareStrategy.

References:

- **General Object-Oriented Design & Patterns:** Gamma, E. et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.
- **Nursery Management:** Smith, M. *Commercial Plant Production and Management.*
- ***Frontiers in Pharmacology. (2024).*** *Climate change and the sustainable use of medicinal plants: a call for "new" research strategies. (Discusses the need for effective strategies to preserve and increase plant populations against environmental factors).*
- ***ResearchGate. (2025).*** *Mechanisms and Strategies of Plant Microbiome Interactions to Mitigate Abiotic Stresses. (Explores how plants face extreme environmental conditions, known as* ***abiotic stresses****, that affect their growth and development, necessitating varied biological strategies).*
- ***PMC. (2023).*** *Editorial: The adaptation strategies of plants to alleviate important environmental stresses. (Documents the complex array of* ***plant responses*** *to various environmental pressures, such as drought, and highlights the potential of biochar and microorganisms to combat stress).*
- **Plant Care Biome-Specifics:** Various online resources on xeriscaping and tropical horticulture.
- https://www.cs.up.ac.za/cs/lmarshall/TDP/TDP.html

Design Pattern Application

The Plant Nursery Simulator leverages **13 distinct design patterns** to model the inherent complexity, heterogeneity, and dynamic workflows of a live nursery environment. The strategic selection of these patterns directly addresses the project's **Functional Requirements (FRs)** and ensures compliance with the necessary **Non-Functional Requirements (NFRs)**.

1. Creational Patterns (3 Used: Singleton, Prototype, Factory Method)

Creational patterns abstract the instantiation process, making the system independent of how its objects are created.

Pattern	Functional Requirement (FR)	Justification & Implementation	Non-Functional Requirement (NFR)
Singleton	FR 1: The plant lifecycle must progress through states (Seedling to Growing to Mature to Wilting to Dead) based on conditions and time[cite: 273, 274].	**Justification:** Guarantees that only one instance of each **PlantState** class exists. This is critical as state objects are stateless and represent shared behaviour, avoiding unnecessary object creation and memory use. **Implementation:** Applied to all concrete PlantState classes (DeadState, MatureState, etc.)[cite: 273, 274].	**Reliability:** By using a single, universally accessed instance, the risk of inconsistent state logic or duplicate state objects is eliminated, supporting system **Reliability**[cite: 298].
Prototype	FR 5: For a specified biome, the factory must create the correct "Pot" and "SoilMix" subtypes[cite: 281].	**Justification:** Avoids costly creation processes by cloning pre-configured components. This is vital for rapidly generating the complementary components (Pot, SoilMix) that constitute a plant kit when a new plant is created[cite: 281]. **Implementation:** The abstract classes (Pot, SoilMix) implement a virtual clone() method, which concrete products (TerracottaPot, AquaticSoilMix) use for self-replication.	**Scalability:** Enables fast, memory-efficient creation of numerous accessory objects during batch restock operations, contributing to performance and **Scalability**[cite: 297].

Factory Method	FR 5: For a specified biome, the factory must create the correct "Pot" and "SoilMix" subtypes[cite: 281].	**Justification:** Defines an interface (`PlantKitFactory`) for creating objects but delegates the specific component instantiation (`createPot`, `createSoilMix`) to concrete subclasses (`DesertFactory`, `TropicalFactory`). This enforces biome-specific component matching[cite: 281, 282]. **Implementation:** The `PlantKitFactory` declares the abstract `createPot()` and `createSoilMix()` methods, implemented by concrete factory classes[cite: 281].	**N/A** (Primarily a Functional Enabler)

2. Structural Patterns (3 Used: Facade, Flyweight, Decorator)

Structural patterns focus on composing classes and objects into larger structures to build flexible and efficient systems.

Pattern	Functional Requirement (FR)	Justification & Implementation	Supporting Non-Functional Requirement (NFR)
Facade	FR 10: "NurseryFacade" must expose a single API for common workflows: create plant/order, customize a package, check stock, apply care, and subscribe to updates[cite: 291].	**Justification:** Provides a simplified, high-level interface to the complex subsystems (Inventory, Sales, Command Invoker). This shields external users from the intricate internal dependencies and logic[cite: 291]. **Implementation:** `NurseryFacade` acts as the orchestrator, delegating calls like `checkout()` or `waterPlants()` to the relevant services.	**Usability:** Directly satisfies the **Usability** NFR by ensuring that common workflows are invocable through a single facade method, requiring "no subsystem knowledge" from the client[cite: 299].
Flyweight	FR 9: Species and care presets must be stored as flyweights identified by "SpeciesKey" to avoid duplication across plants[cite: 289].	**Justification:** Essential for memory optimization by sharing intrinsic, immutable species data (**SKU, Biome, GrowthRate**) across a massive population of `Plant` objects[cite: 289]. **Implementation:** `PlantFlyweight` is the abstract interface; `SpeciesFlyweight` is the concrete flyweight storing the shared immutable properties.	**Scalability:** Supports the **Scalability** NFR by enabling the system to support "up to 10,000 plants" with linear tick time growth, as only extrinsic (unique) data is stored per plant instance[cite: 297].

Decorator	FR 7: The system must use the abstract factory to create variable parts of the Product (CustomPlantPackage)[cite: 285].	**Justification:** Provides a flexible alternative to subclassing for extending sale item functionality. It wraps a **SaleItem** (PlantItem) to dynamically add costs and descriptions for customer customization options (e.g., premium pot upgrade, gift wrap) at runtime. **Implementation:** SaleItem is the component. SaleDecorator is the abstract decorator, with concrete decorators modifying the final cost() and description().	**N/A** (Primarily a Functional Enabler, enhancing personalization beyond basic factory creation)

3. Behavioural Patterns (7 Used: Command, Iterator, Mediator, Observer, State, Strategy, Template Method)

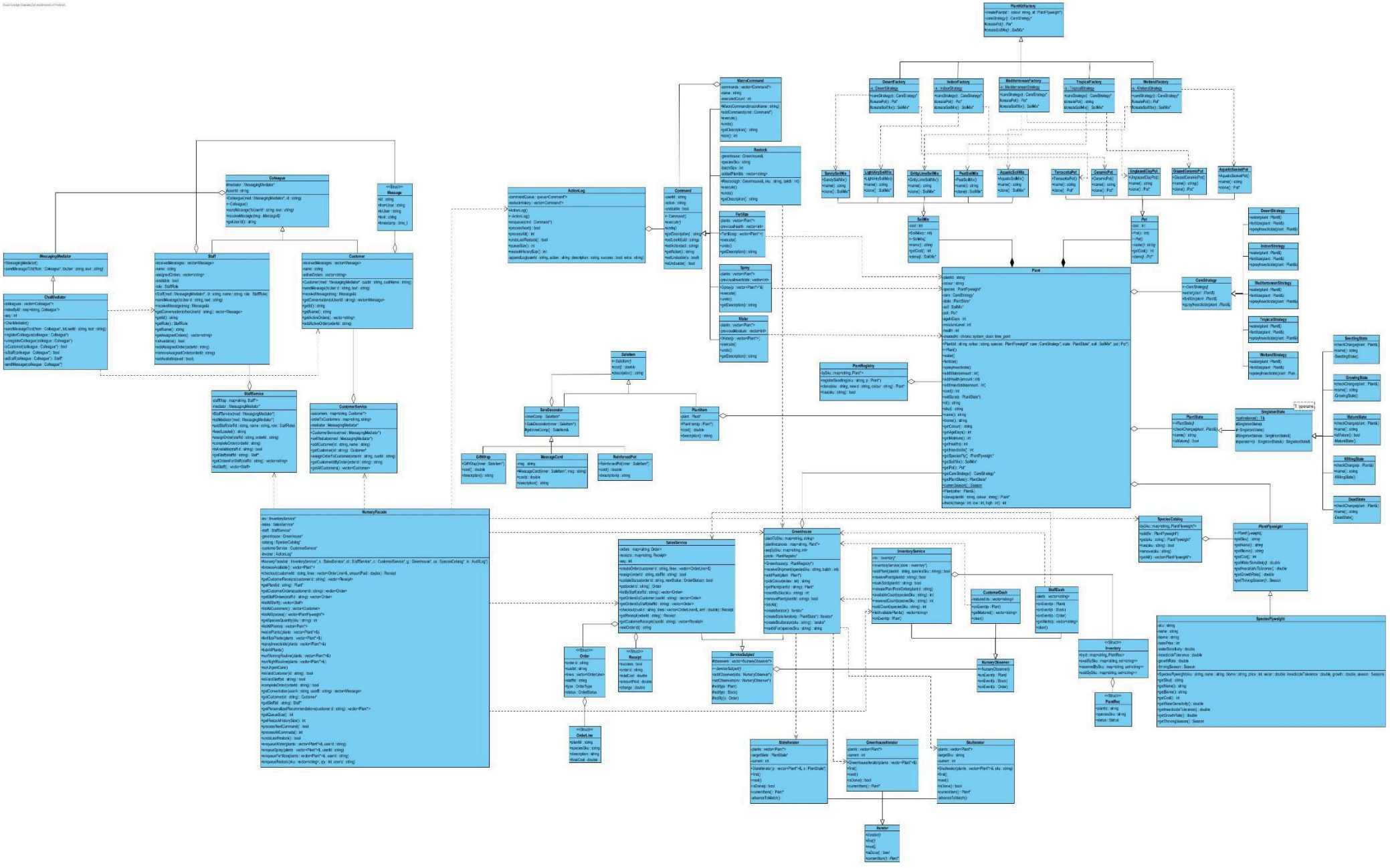
Behavioural patterns manage algorithms, the assignment of responsibilities, and communication between objects.

Pattern	Functional Requirement (FR)	Justification & Implementation	Supporting Non-Functional Requirement (NFR)
Command	FR 3 & FR 4: Staff must be able to issue care commands ("Water," "Fertilize") and inventory/sales commands ("Restock") cite: 278, 279].	**Justification:** Encapsulates staff actions as objects, supporting a robust command queue (ActionLog) and enabling the crucial **undo/redo** functionality (e.g., for Restock actions)[cite: 278, 279]. **Implementation:** Command is the abstract interface. Concrete commands like WateringCommand and Spray encapsulate the action. **MacroCommand** is the composite command.	**Security:** The pattern's separation allows the ActionLog to implement the **Security** NFR by validating the authenticated staff role *before* executing the command that mutates state, rejecting unauthorized calls[cite: 300].
Iterator	FR 8: The system must provide iterators over Greenhouse, PlantState, and SKU collections to traverse plants deterministically without exposing the internal structure [cite: 287].	**Justification:** Allows for standardized traversal through collections of plants regardless of the underlying container, enabling efficient and reusable batch operations (e.g., watering all plants in a specific state)[cite: 287]. **Implementation:** The abstract Iterator interface is implemented by specialized iterators (GreenhouseIterator, StateIterator, SkuIterator).	**N/A** (Primarily a Functional Enabler)

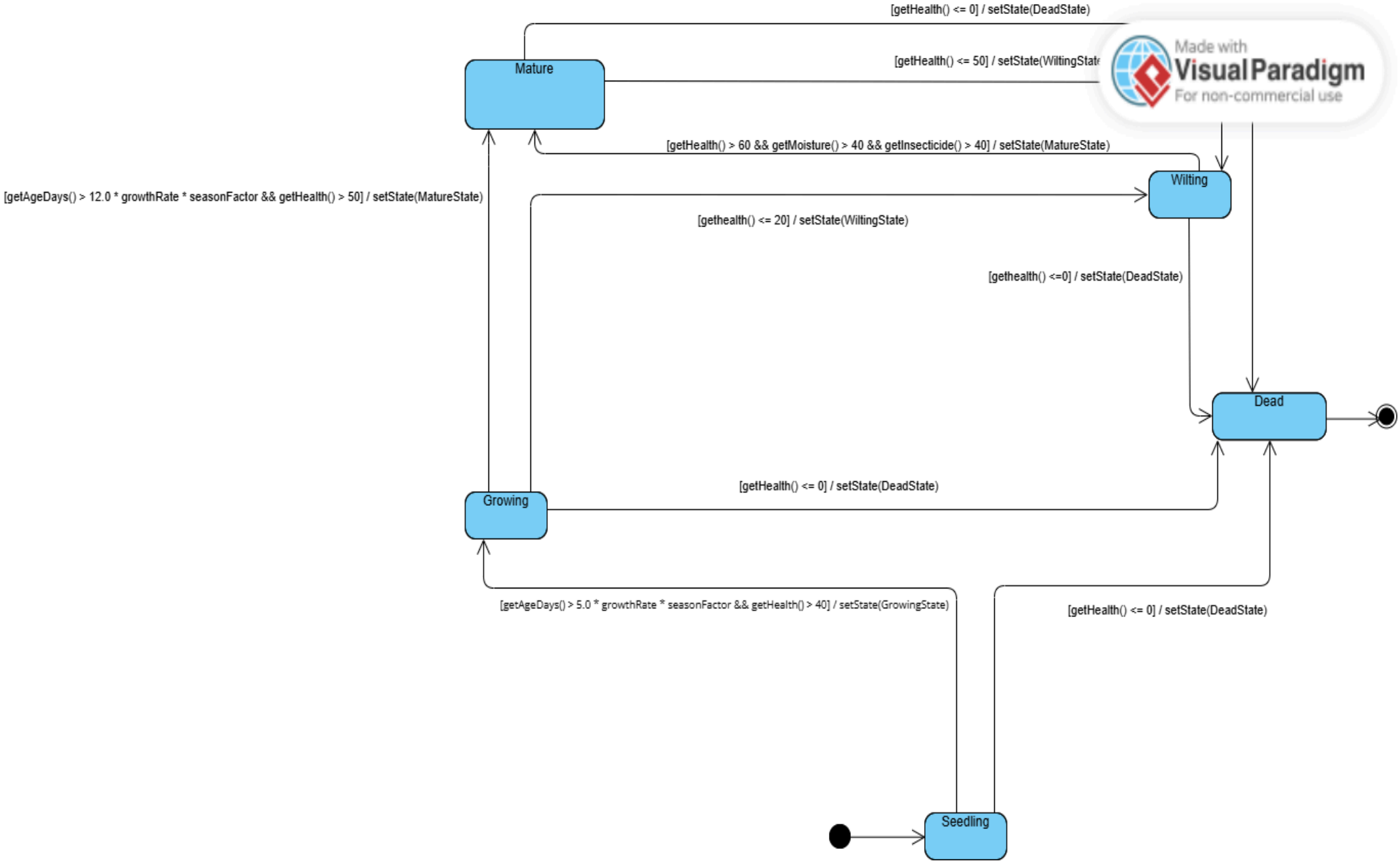
Mediator	FR 12: Customer and Staff instances must publish events (e.g., MessageSentEvent, MessageReceivedEvent) to observers (ChatMediator) [cite: 295].	**Justification:** Establishes a one-to-many dependency: the subject (e.g., Customer) notifies its observer (ChatMediator) when an action occurs (e.g., sends a message), decoupling the event source from its listeners [cite: 295]. **Implementation:** MessagingObserver is the abstract observer. ChatMediator acts as the concrete observer, receiving and routing events between colleagues.	**Reliability:** Observer design ensures all event handlers are idempotent (per NFR), so duplicate messages or notifications do not cause inconsistent states, stabilizing the system [cite: 298].
Observer	FR 11: Customer and Staff instances must publish events (e.g., MessageSentEvent, MessageReceivedEvent) to observers (ChatMediator)[cite: 293].	**Justification:** Establishes a one-to-many dependency: the subject (e.g., Customer) notifies its observer (ChatMediator) when an action occurs (e.g., sends a message), decoupling the event source from its listeners [cite: 293]. **Implementation:** MessagingObserver is the abstract observer. ChatMediator acts as the concrete observer, receiving and routing events between colleagues.	**Reliability:** Observer design ensures all event handlers are idempotent (per NFR), so duplicate messages or notifications do not cause inconsistent states, stabilizing the system [cite: 298].

State	FR 1: The plant lifecycle must progress through states (Seedling to Growing to Mature to Wilting to Dead) based on conditions and time[cite: 273, 274].	**Justification:** Allows a plant object to dynamically change its behavior (e.g., growth rate, reaction to care) when its internal state changes. This is the core model for the **Plant Life Cycle**[cite: 273, 274]. **Implementation:** PlantState is the abstract state. Concrete states (SeedlingState, MatureState, WiltingState) define the unique behavior and transitions (the **Singleton** pattern is used to instantiate them).	**N/A** (Primarily a Functional Enabler)
Strategy	FR 2: The system must select a watering strategy based on a plant's biome (e.g., Desert, Tropical, Indoor, Mediterranean, Wetland)[cite: 276].	**Justification:** Encapsulates varying care algorithms (watering, fertilizing) into interchangeable objects. This is the structural solution to the plant heterogeneity problem, allowing for new biomes and care routines to be added easily[cite: 276]. **Implementation:** CareStrategy is the abstract interface. Concrete strategies (DesertStrategy, TropicalStrategy) implement the specific care logic. The Plant object holds a reference to its current strategy.	**N/A** (Primarily a Functional Enabler, supporting extensibility)
Template Method	FR 6: "createPlant" must combine factory methods and inject state, strategy and flyweight into the Plant[cite: 283].	**Justification:** Ensures a consistent plant creation process in PlantKitFactory while allowing subclasses to define specific components via Factory Methods[cite: 283]. **Implementation:** The `PlantKitFactory::createPlant()` is the template method, which sequentially calls the abstract factory methods to get components and then constructs the final `Plant` object.	**N/A** (Primarily a Functional Enabler)

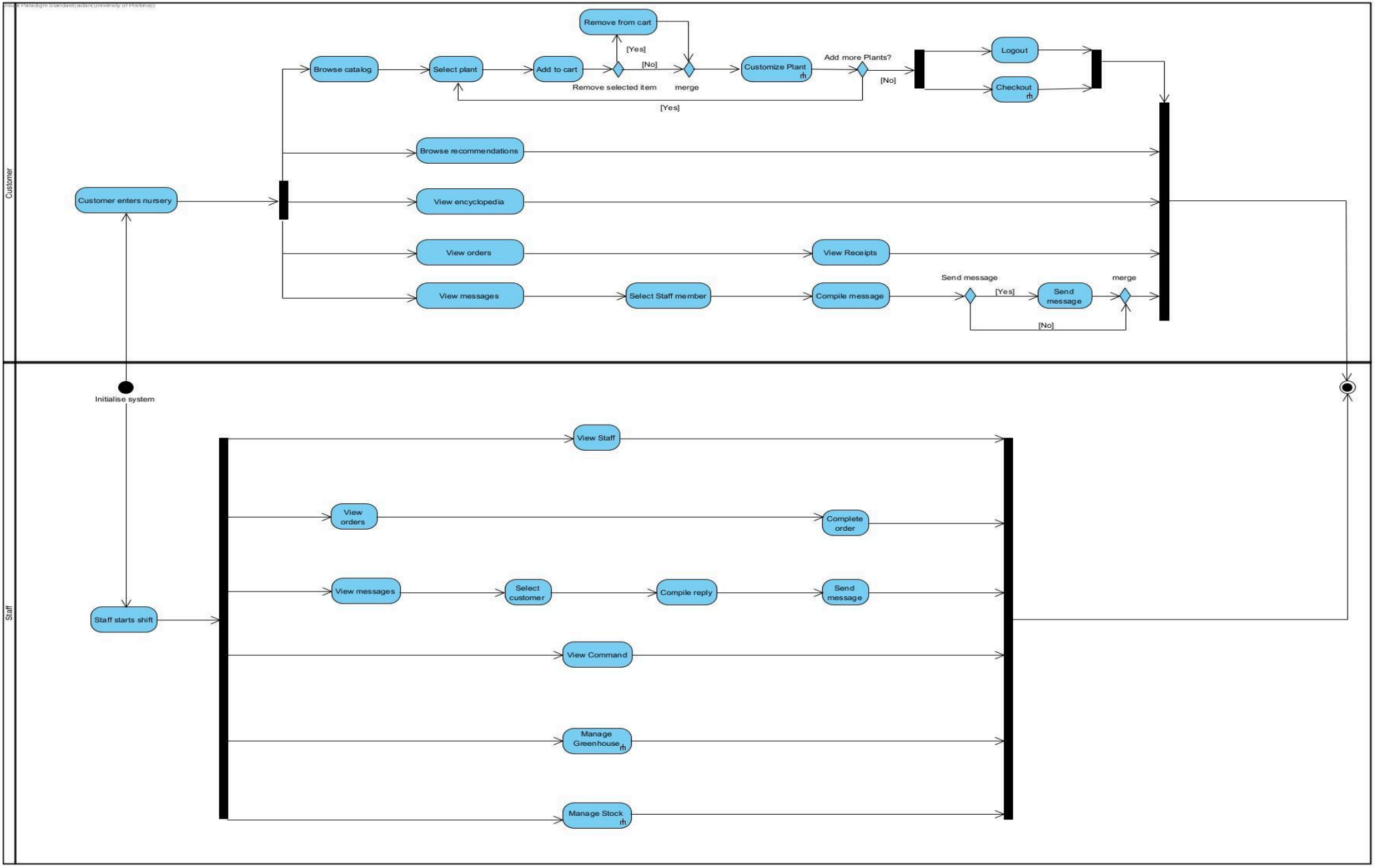
Class diagram

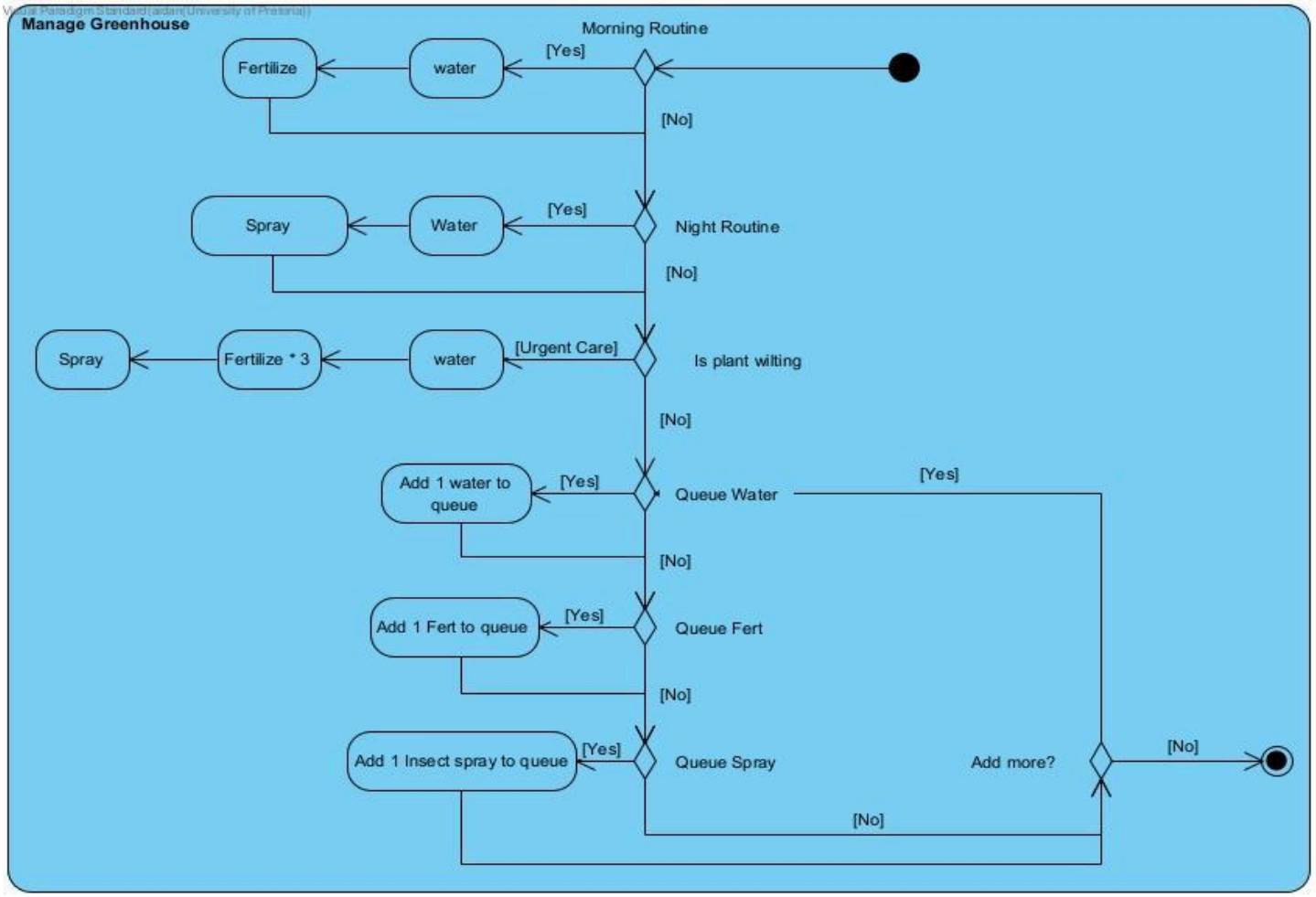


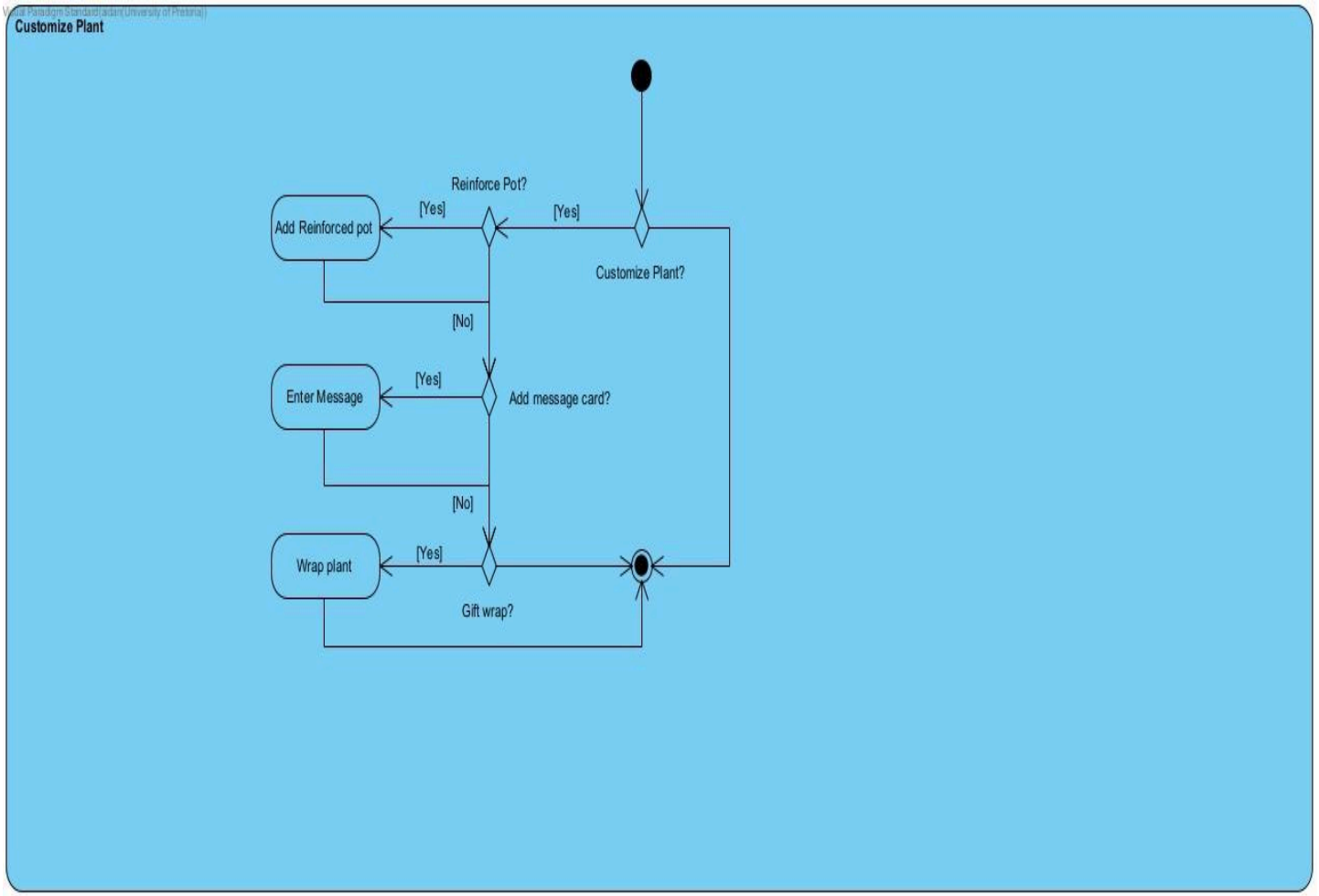
UML State Diagram



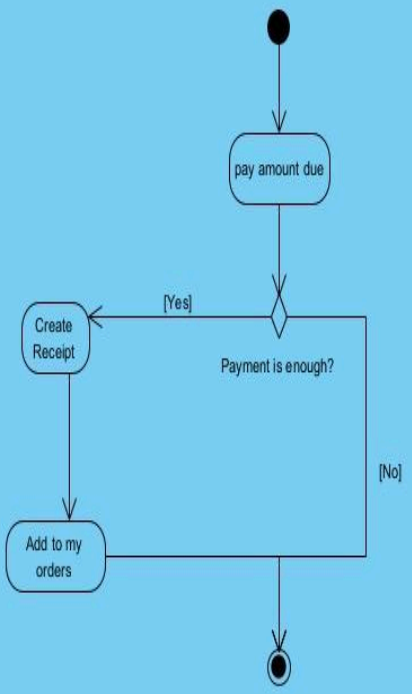
UML Activity Diagram



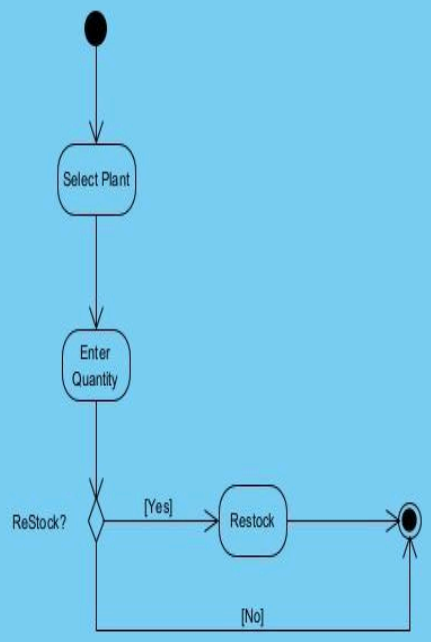




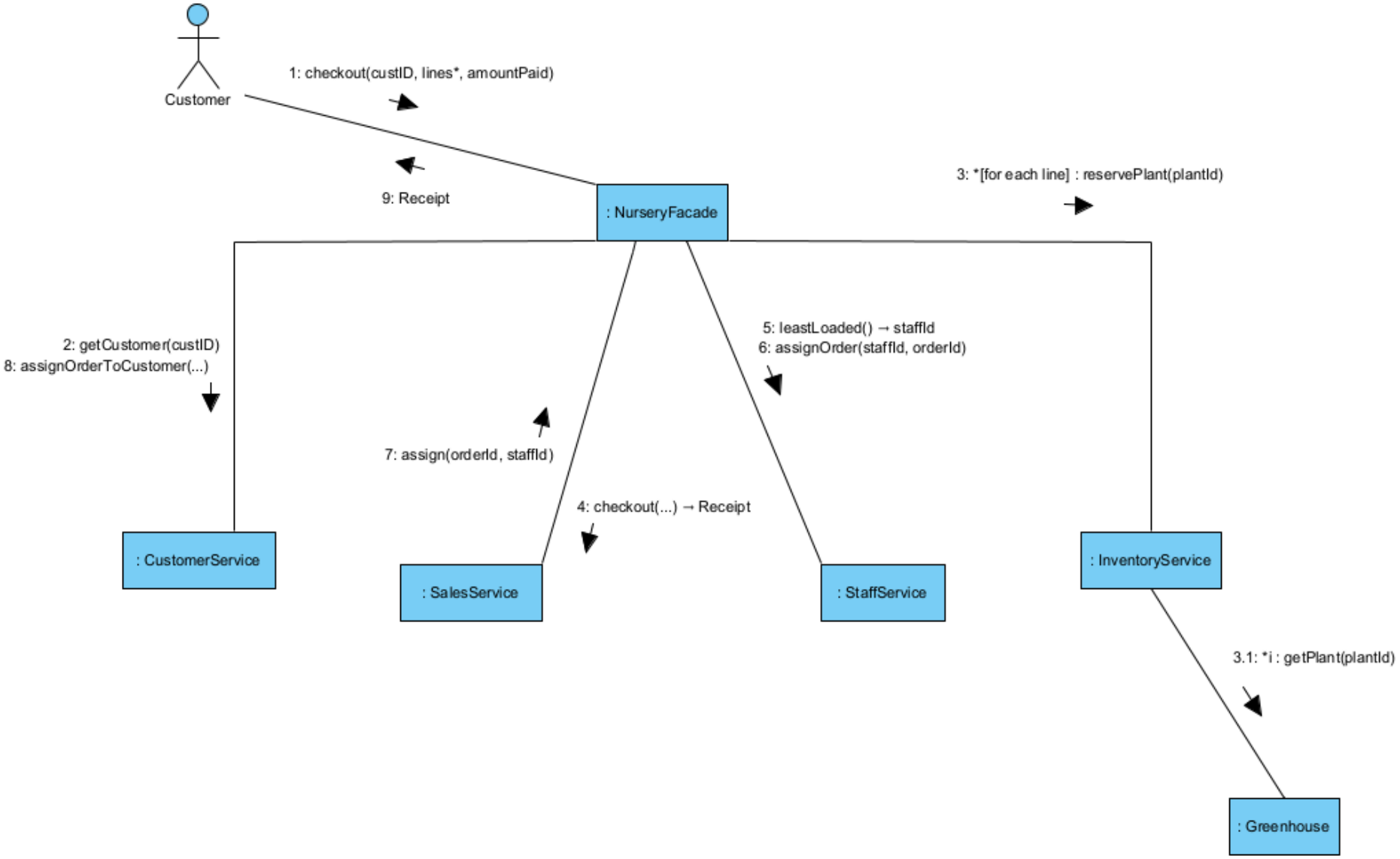
Checkout



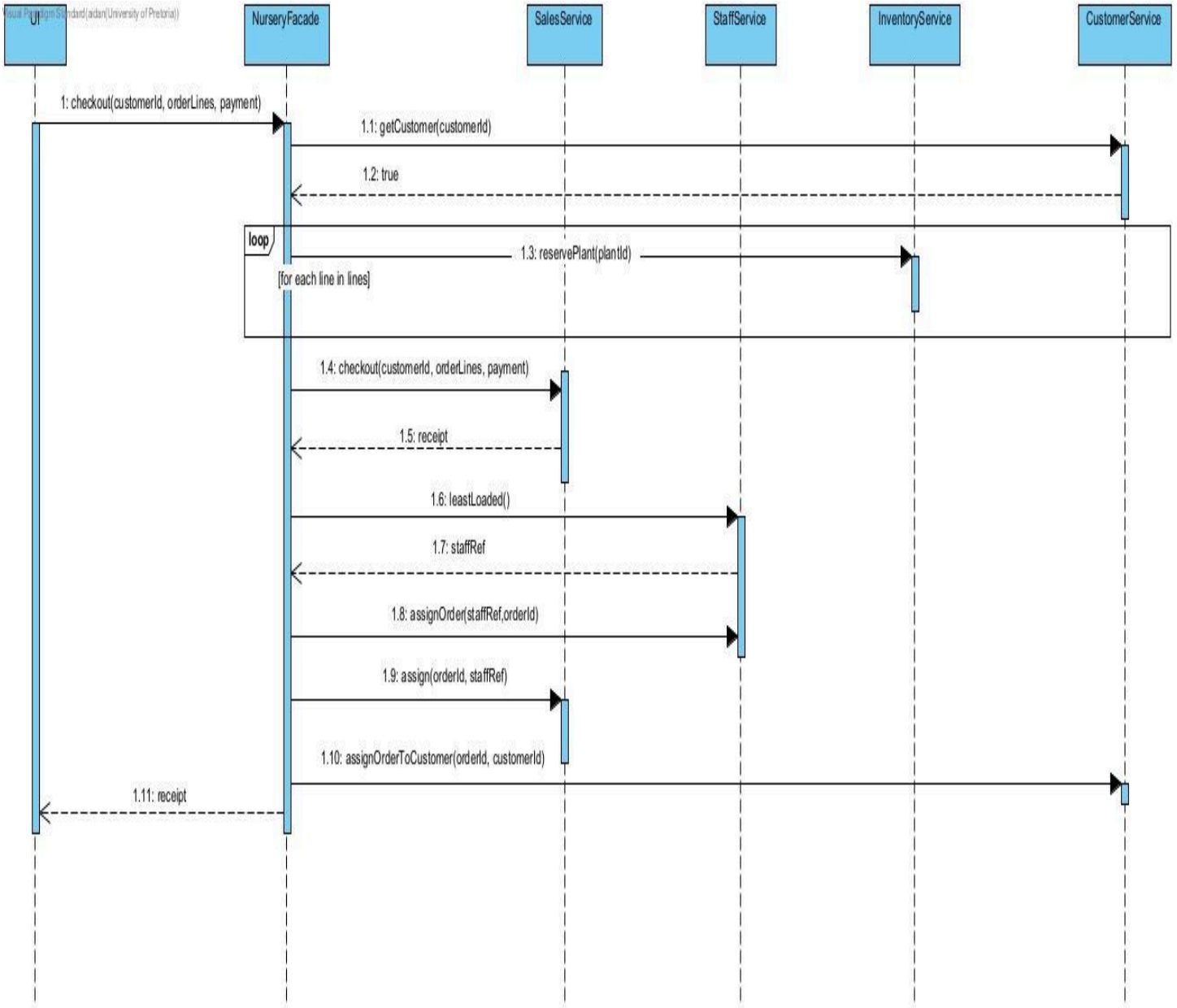
Manage Stock



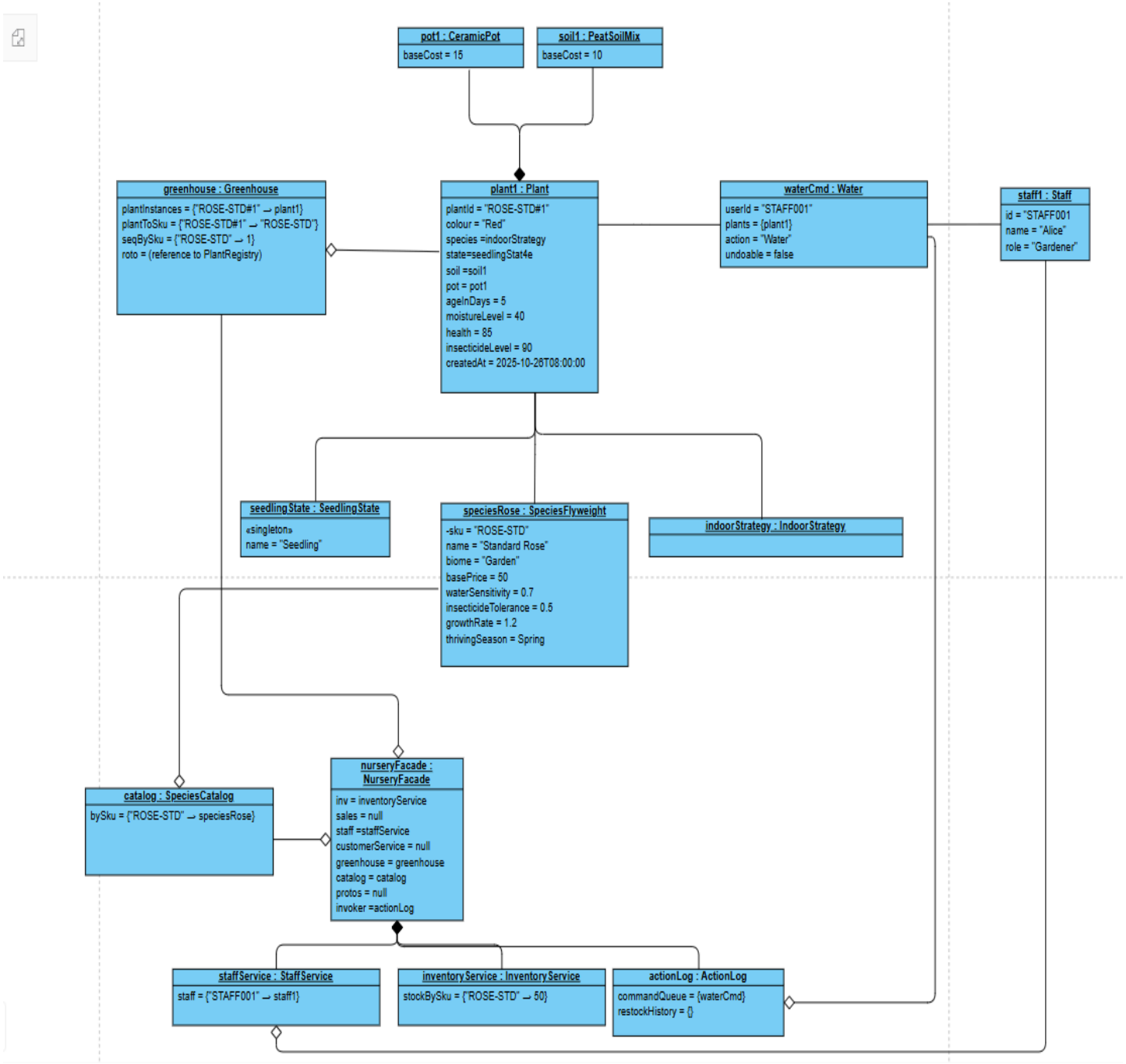
Communication Diagram



UML Sequence Diagram



Object Diagram



Git Branching Strategy:

Gitflow Workflow: Our team adopted the Gitflow branching model to manage development effectively. Using separate branches for features, development, and releases allowed multiple team members to work on different parts of the project simultaneously without interfering with each other's work. Gitflow enforced a clear structure for integrating code, handling hotfixes, and preparing stable releases, which minimized merge conflicts and improved collaboration. By having defined roles for feature, develop, release, and main branches, we maintained a clean history, ensured code stability, and accelerated our workflow as a team.

Google Docs Link: [Project Team Google Drive Link to The Final Report]

Conscientious Students:

GS: Jeandre Opperman 23542773
Ayrtonn Talijaard 24856462
Aaron Kim 21494305
Damian Moustakis 24564738
Franky Liu 24673898
Shaun Marx 24673481
Aidan Dawson 24593542