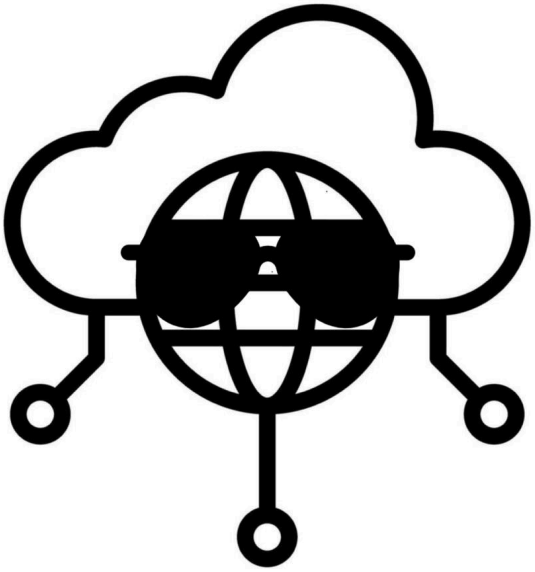


COS214 Report

THE TEAM :

ANÉ DE RIDDER - U23542676

JOSHUA KRETSCHMER- U22883925

DAVID KALU - U23534975

OPHELIA GREYLING - U23586312

MICHELLE NJOROGE - U21448842

KEAGAN VAN BILJON - U23594412

BRIDGET NKOSI - U22664638

https://docs.google.com/document/d/1-I_49XCp5JTZUpwk8WRsAEfSgu-q_DB94tuHgr63K9w/edit?usp=sharing

Research Brief

We researched nursery management to build a semi-realistic Plant Nursery Simulator, drawing from horticultural resources and plant nursery operation contexts. Focus areas included plant types, care routines, life cycles, inventory, staff roles, and customer interactions.

Key findings:

- Plant Types and Care: Implemented plants include succulents like Peanut Cactus (bright direct light, water when dry, reduce in cool winters) and Houseleek (full sun, minimal water); flowers such as Orchid (bright indirect light, water weekly) and Marigolds (full sun 6-8 hours, water when top inch dry); shrubs like Bee Blossom (full sun, drought-tolerant, infrequent deep watering) and Honeysuckles (full to partial sun, regular watering especially when young, prune after bloom). Overwatering causes issues like root rot.
- Life Cycles: Stages range from seeding (1-2 weeks germination) to growing (4-12 weeks), maturity (sellable), and decline.
- Management: Inventory uses real-time tracking; staff handle plant care and sales; customers seek personalization and advice.

This shaped the design: The care variable led to Strategy pattern; life cycles to State pattern; plant diversity to Factory; inventory to Composite; interactions to Chain of Responsibility and Command for caring for both plants and customers alike.

Assumptions/Definitions:

- Simplified growth (command-based states)
- No external factors like weather
- "Maturity": sellable state
- "Care Routine": simple actions per plant

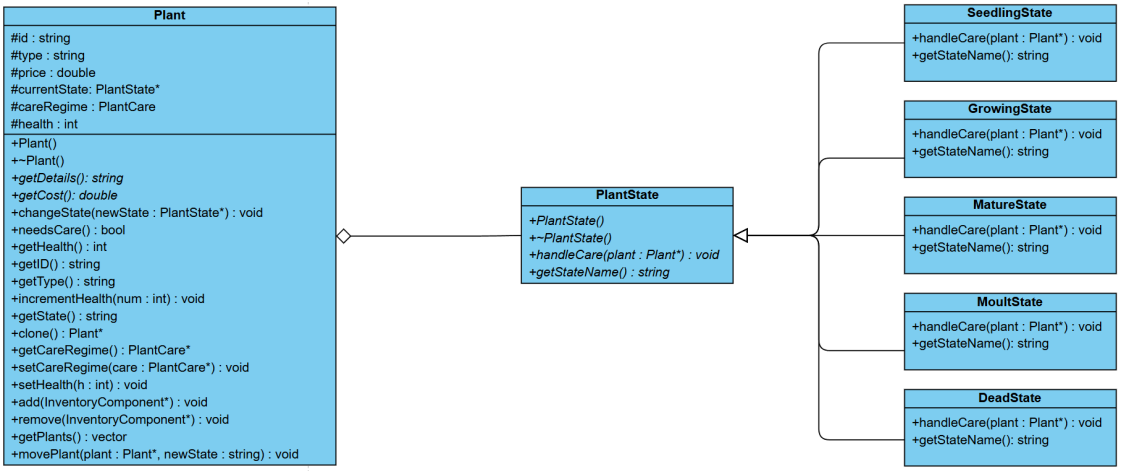
References:

1. FarmERP. (2023). "How a Plant Nursery Management System Can Streamline Inventory Control?"
https://www.farmerp.com/blog/how-a-plant-nursery-management-system-can-streamline-inventory-control/

2. Royal Horticultural Society. "Beginner's guide to gardening"
https://www.rhs.org.uk/advice/beginners-guide

Design Pattern Application Details

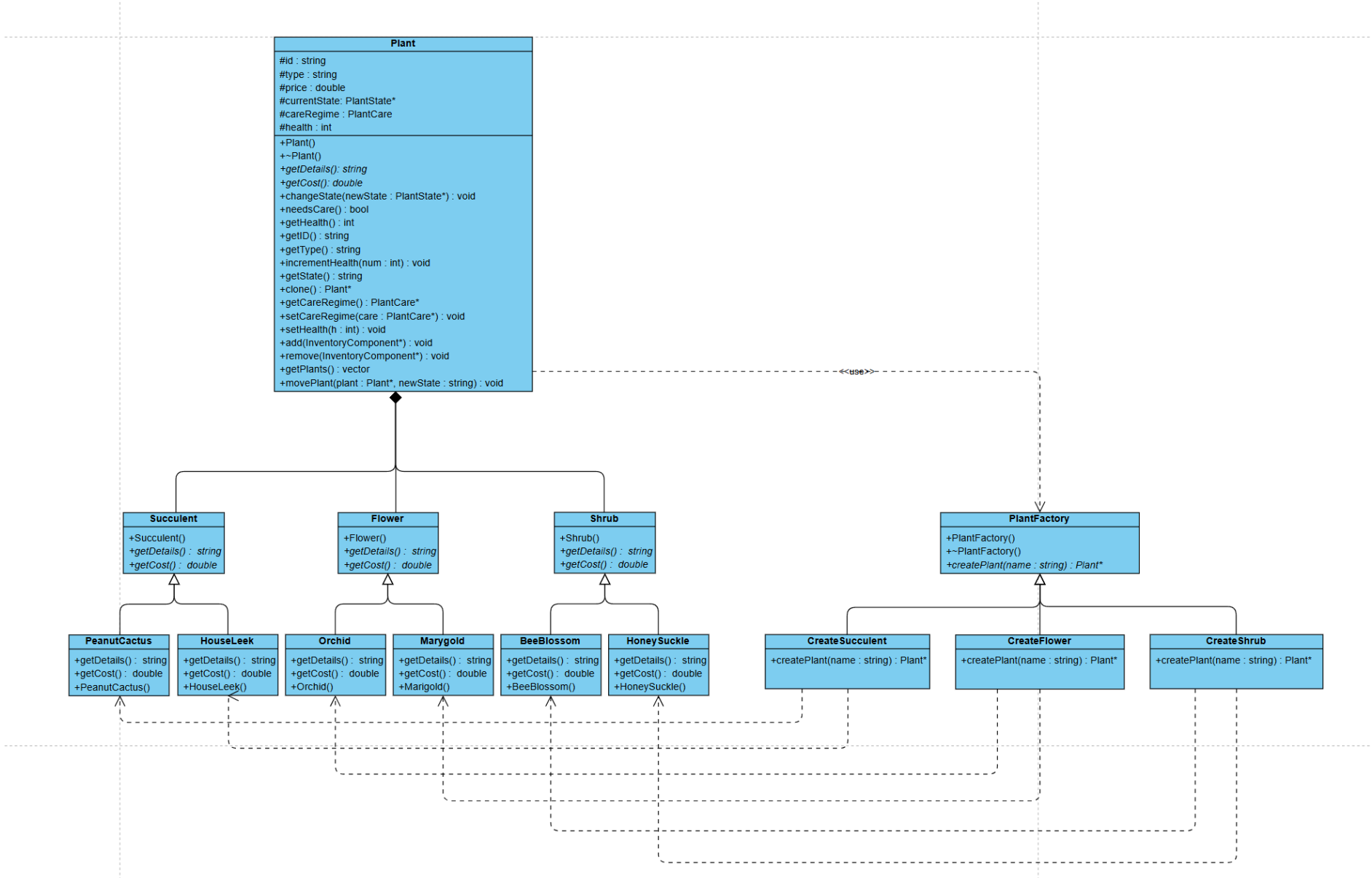
Plant States (State)



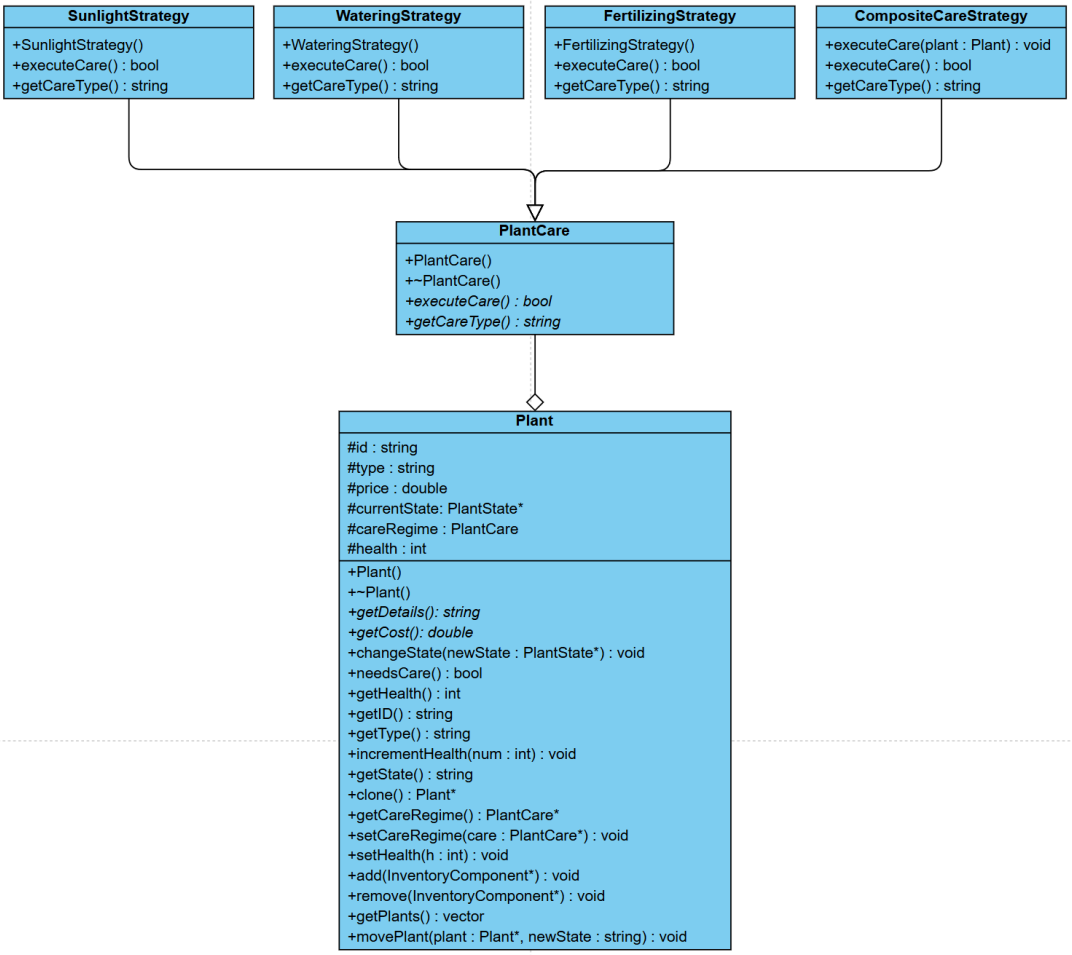
This design pattern simulates what happens when Staff (specifically the Horticulturist) tend to the plants' needs. After each care execution, the plant's state checks health levels to determine if progression or regression should occur, creating a simulation of where care strategies would directly impact lifecycle states.

Adding of New Plants (Factory)

The Factory Method pattern fulfills the functional requirement for creating different plant types by implementing specialized factory classes (CreateSucculent, CreateFlower, CreateShrub) that encapsulate plant creation logic. It allows the system to easily generate succulents, flowers and shrubs without coupling client code to concrete plant classes. This allows for adding new plants and plant types easily while maintaining clean separation of concerns.

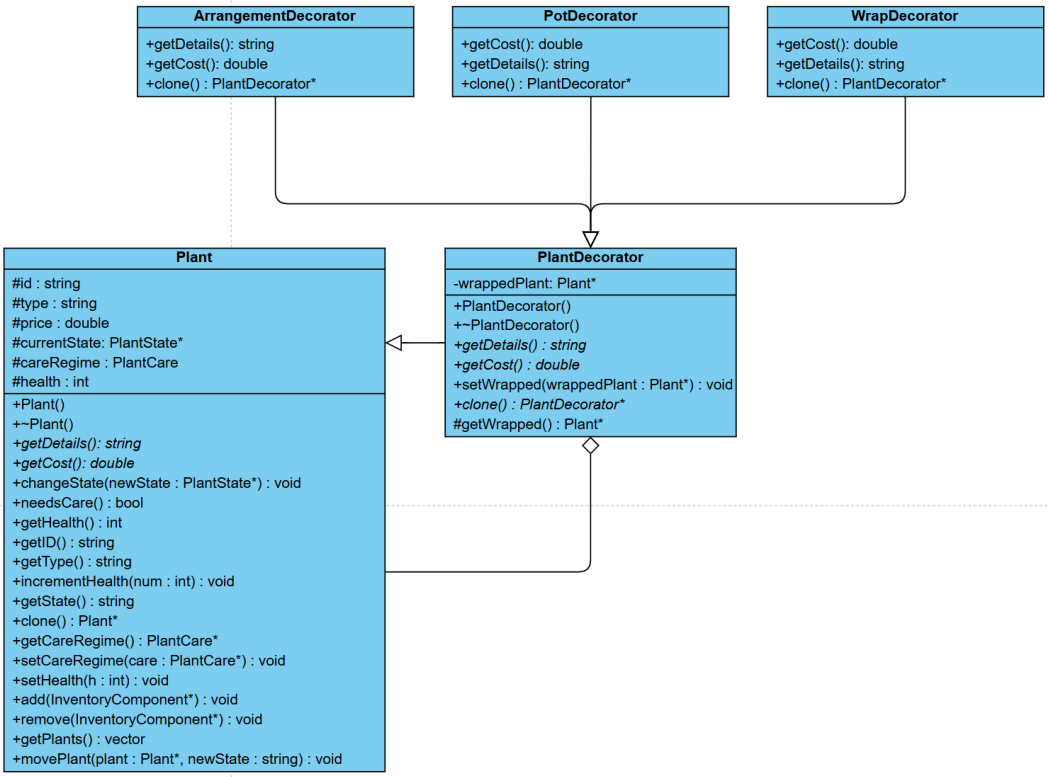

Plant
#id : string
#type : string
#price : double
#currentState: PlantState*
#careRegime : PlantCare
#health : int
+Plant()
+~Plant()
+getDetails(): string
+getCost(): double
+changeState(newState : PlantState*) : void
+needsCare() : bool
+getHealth() : int
+getID() : string
+getType() : string
+incrementHealth(num : int) : void
+getState() : string
+clone() : Plant*
+getCareRegime() : PlantCare*
+setCareRegime(care : PlantCare*) : void
+setHealth(h : int) : void
+add(InventoryComponent*) : void
+remove(InventoryComponent*) : void
+getPlants() : vector
+movePlant(plant : Plant*, newState : string) : void
<<use>>
Succulent
+Succulent()
+getDetails() : string
+getCost() : double
Flower
+Flower()
+getDetails() : string
+getCost() : double
Shrub
+Shrub()
+getDetails() : string
+getCost() : double
PlantFactory
+PlantFactory()
+~PlantFactory()
+createPlant(name : string) : Plant*
PeanutCactus
+getDetails() : string
+getCost() : double
+PeanutCactus()
HouseLeek
+getDetails() : string
+getCost() : double
+HouseLeek()
Orchid
+getDetails() : string
+getCost() : double
+Orchid()
Marygold
+getDetails() : string
+getCost() : double
+Marigold()
BeeBlossom
+getDetails() : string
+getCost() : double
+BeeBlossom()
HoneySuckle
+getDetails() : string
+getCost() : double
+HoneySuckle()
CreateSucculent
+createPlant(name : string) : Plant*
CreateFlower
+createPlant(name : string) : Plant*
CreateShrub
+createPlant(name : string) : Plant*

Staff Plant Care Execution (Strategy)



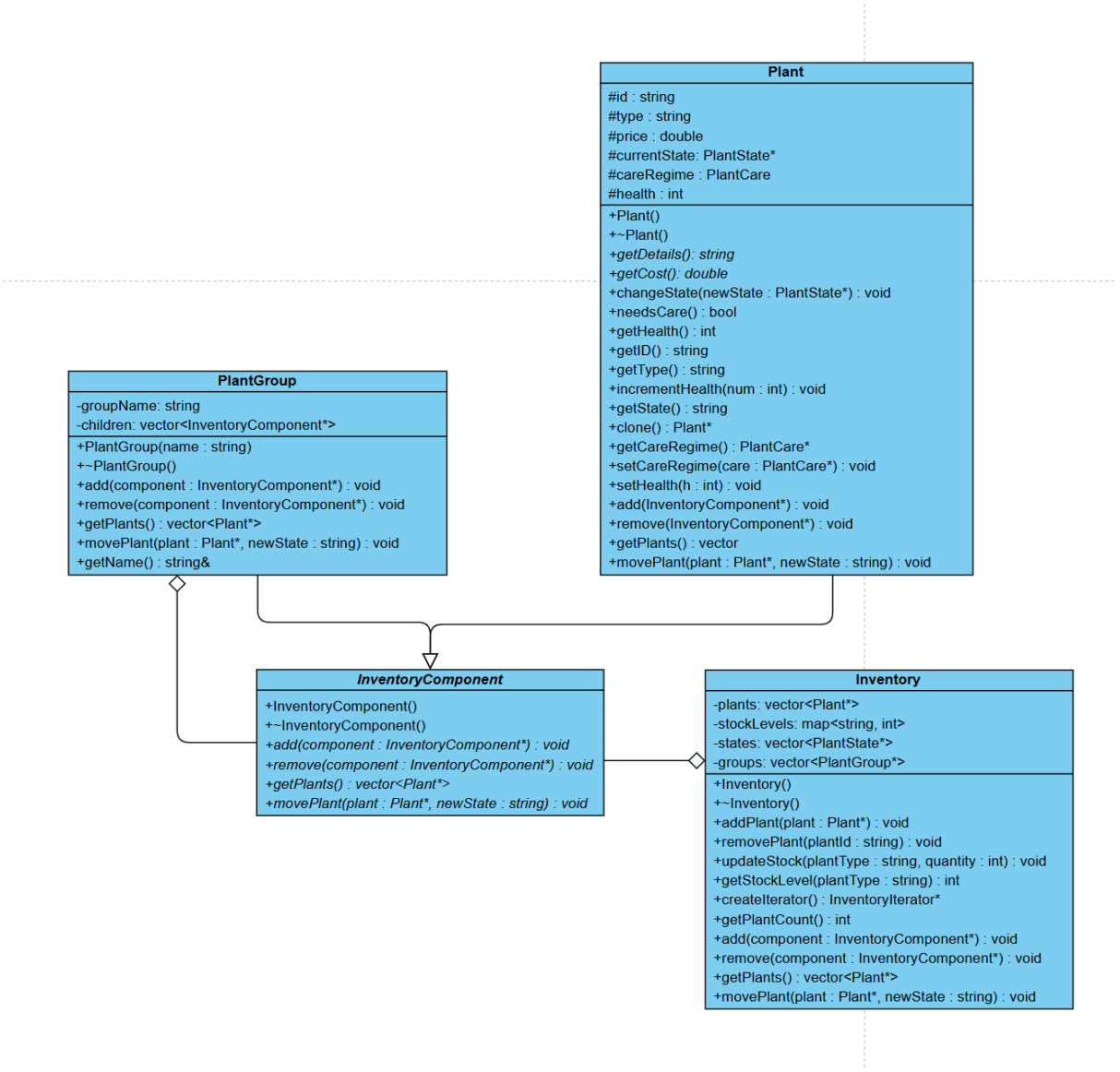
This pattern enables staff to execute specialized care operations tailored to each plant type's specific needs, working in direct coordination with the State pattern. As Staff apply these care strategies, they modify plant health metrics and that triggers state transitions in the plant's lifecycle.

Personalisation of Plant Purchases (Decorator)



Customers are able to personalize their purchases with decorative pots, gift wrapping and special arrangements (without modifying the core plant classes). The pattern helps maintain the Plant interface, enhancing functionality and allowing unlimited customisation combinations while automatically calculating accumulated costs. This provides a flexible purchasing experience where customers can build customized plant orders through simple composition rather than complex inheritance hierarchies.

Plant Grouping and Organization (Composite)

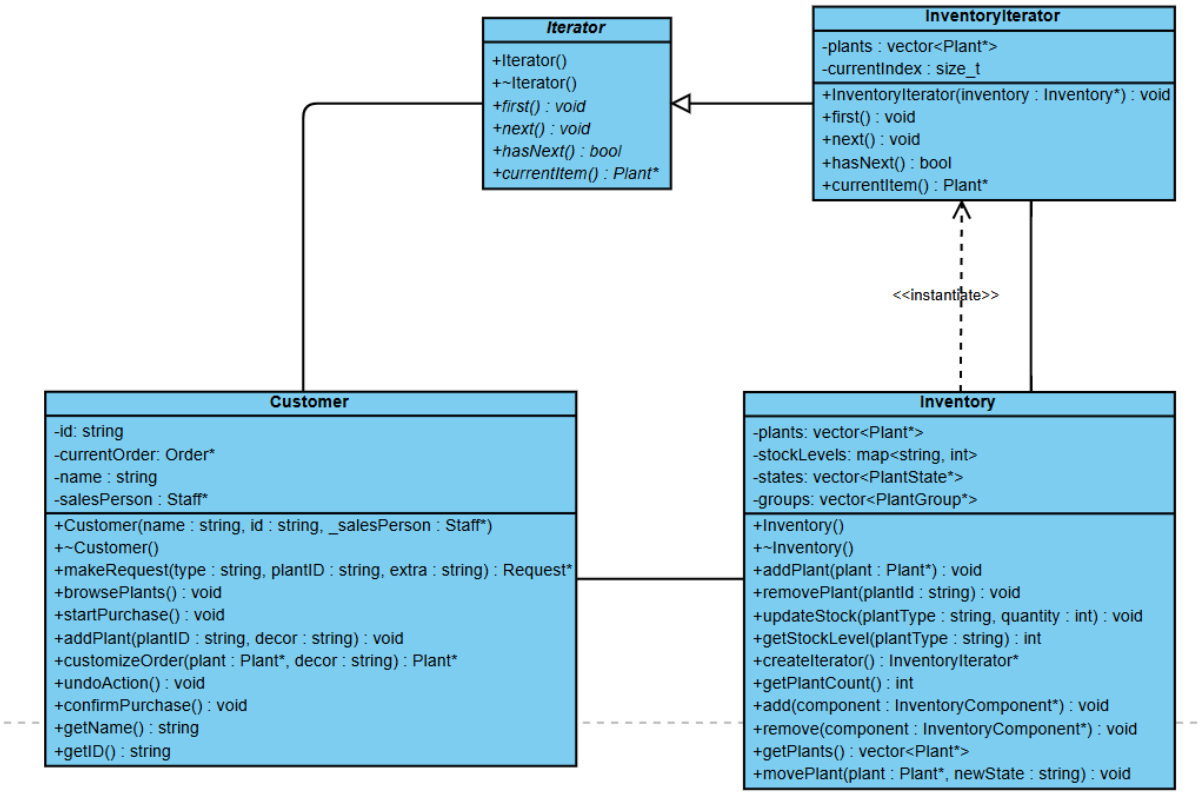


To treat individual plants and plant collections uniformly we implemented the Composite pattern.

This enables the system to manage plants in logical groups such as greenhouse sections or care categories while maintaining a consistent interface for operations. Staff can apply care routines to entire plant groups through single commands. The pattern works with the Iterator to enable efficient traversal of composite structures.

This hierarchical approach supports bulk operations, simplifies inventory management and ensures that care strategies can be applied consistently across both individual plants and entire collections.

Customer Browsing and Viewing Available Plants (Iterator)

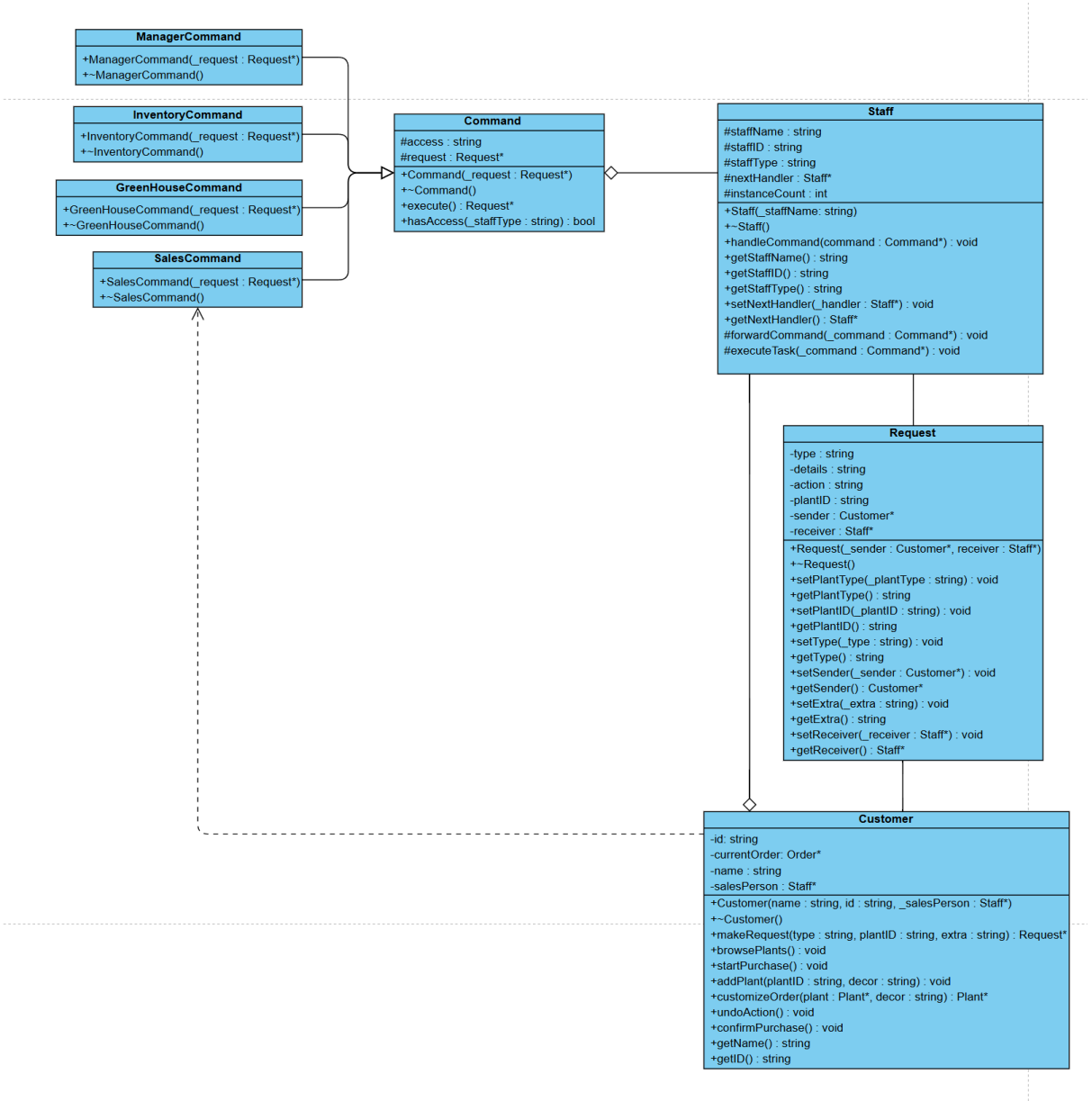


We implemented the Iterator pattern to enable customers to browse available plants through a standardized interface while keeping the inventory implementation encapsulated.

The pattern allows the Inventory Clerk to manage plant traversal for both customers and staff without exposing internal storage details. The clerk utilizes the iterator to efficiently locate specific plants, identify plants needing care based on their state, and facilitate transfers between greenhouse and sales floor.

This maintains a clean separation between inventory access and management responsibilities while supporting flexible browsing and operational needs throughout the system.

Coordination of System-Staff Interactions (Command)

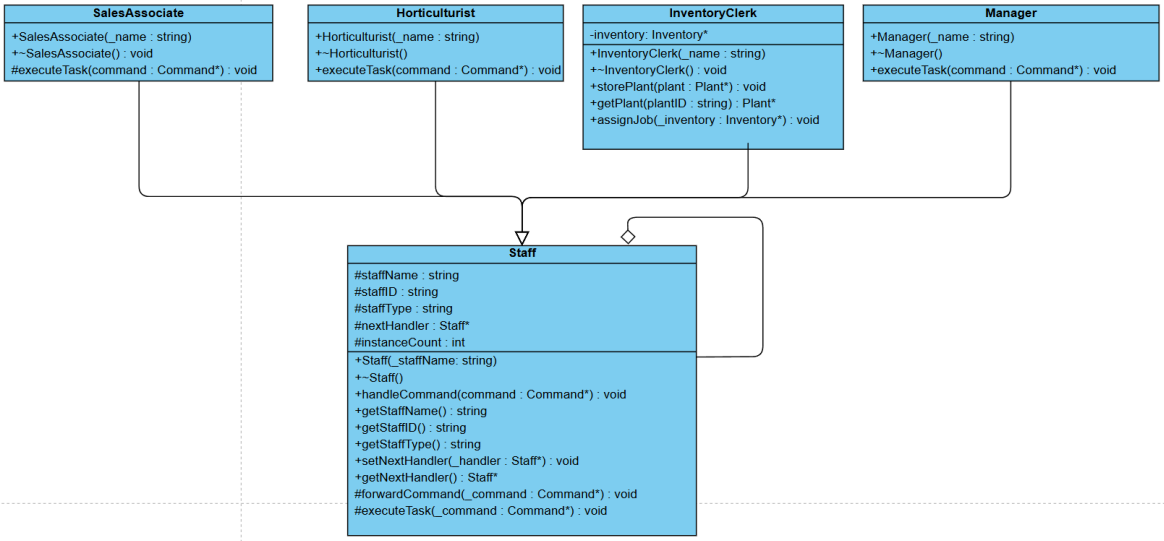


To enable staff to handle various customer and system requests efficiently, we implemented the Command pattern.

The system encapsulates requests as command objects (e.g. SalesCommand, InventoryCommand, ManagerCommand, etc), which are sent to the receiver (Request). For example, when a customer requests a customization, the system generates a SalesCommand that the SalesAssociate executes.

The receiver routes these commands to the appropriate staff member based on their role, ensuring easy and correct handling of tasks like customizations, inventory checks or escalated issues.

Escalation Requests (Chain of Responsibility)

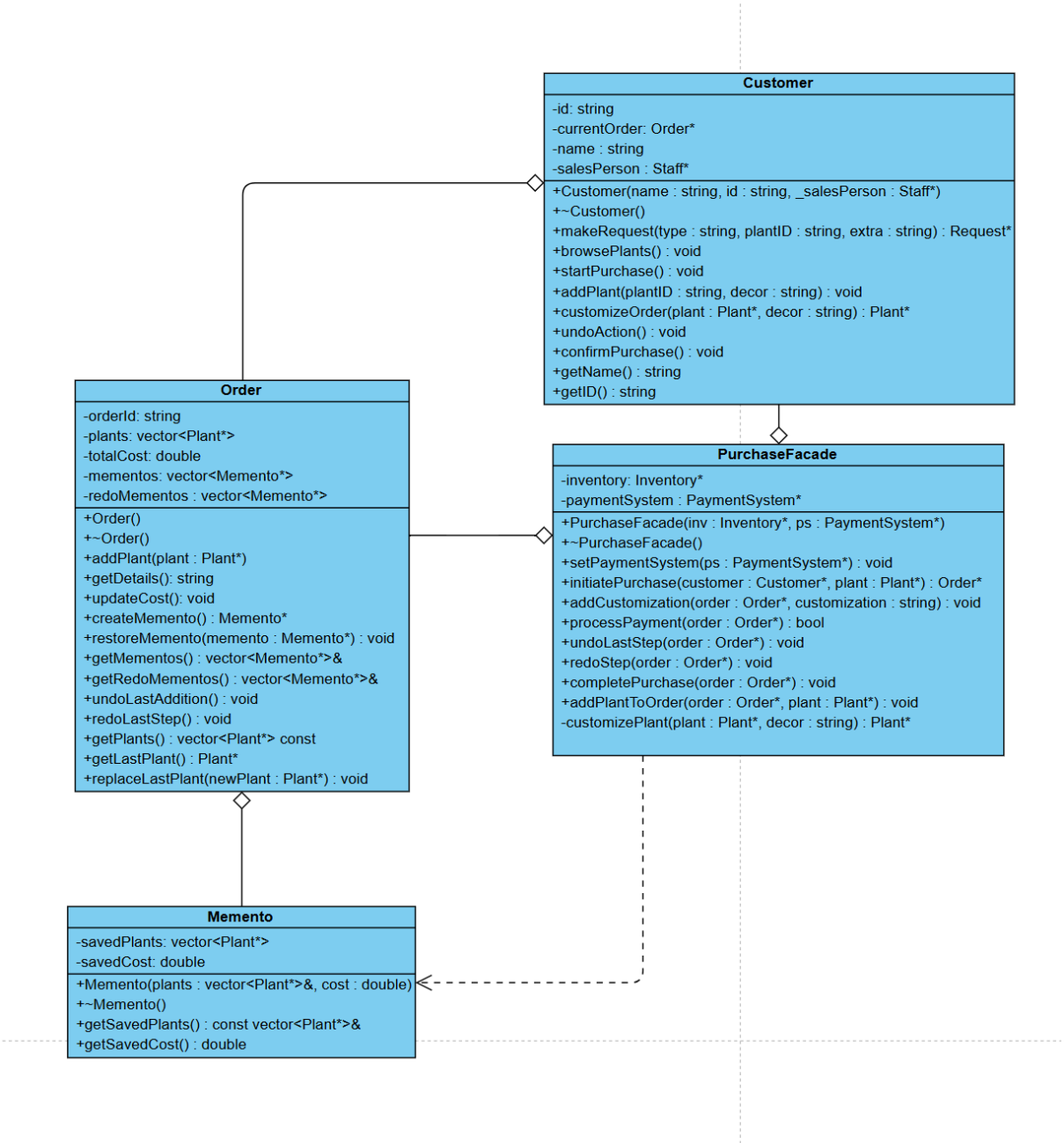


To ensure customer requests are handled efficiently we implemented the Chain of Responsibility pattern for staff interactions.

Requests start with a SalesAssociate, who either processes the request or passes it to the next handler, an InventoryClerk. If the InventoryClerk can't handle it the request moves to a Horticulturist. If no staff member can resolve the issue, it escalates to the Manager as the final handler.

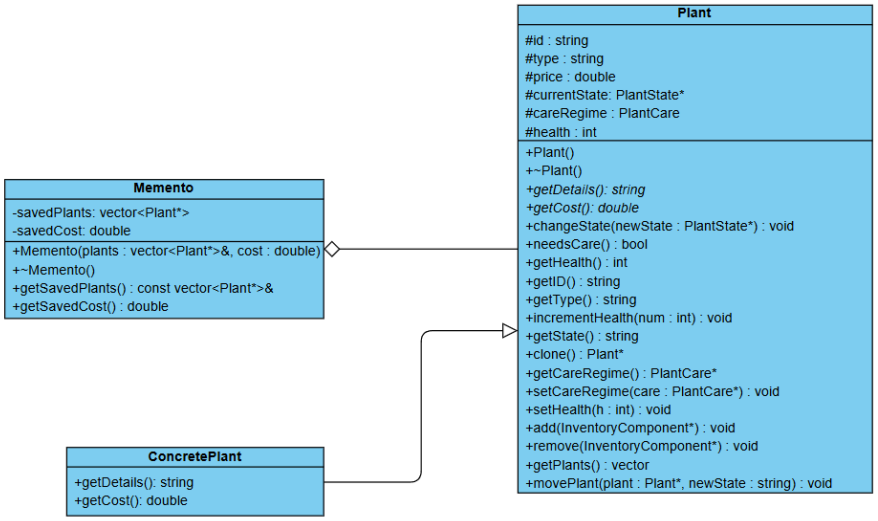
This chain ensures requests are systematically delegated and resolved, with the manager acting as the fallback to guarantee every request is addressed.

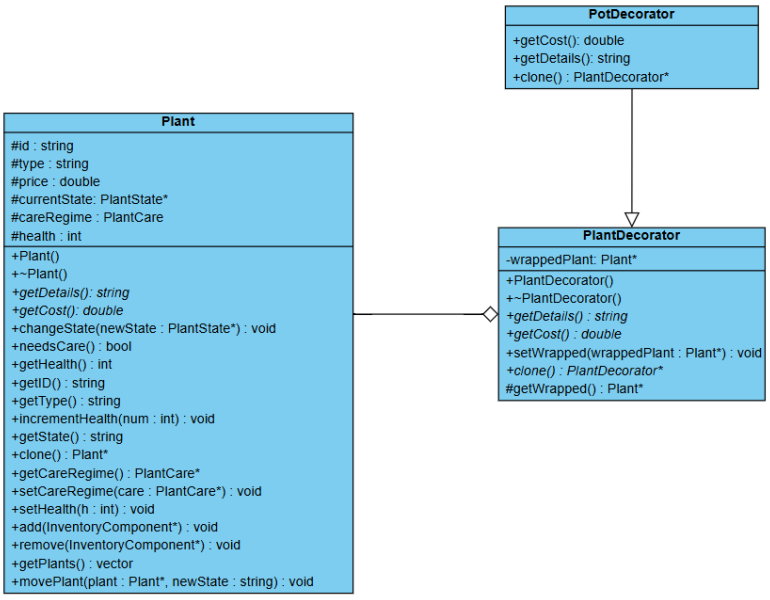
Undo/Redo for Order Modifications (Memento)



To let customers easily undo or redo steps in their orders, like adding or removing a plant, we implemented the Memento pattern. It captures snapshots of the order state at each change, so when someone wants to roll back (or step forward after a rollback), the system just restores the right snapshot without messing up the rest.

Cloning for Plant and Decorator Snapshots (Prototype)

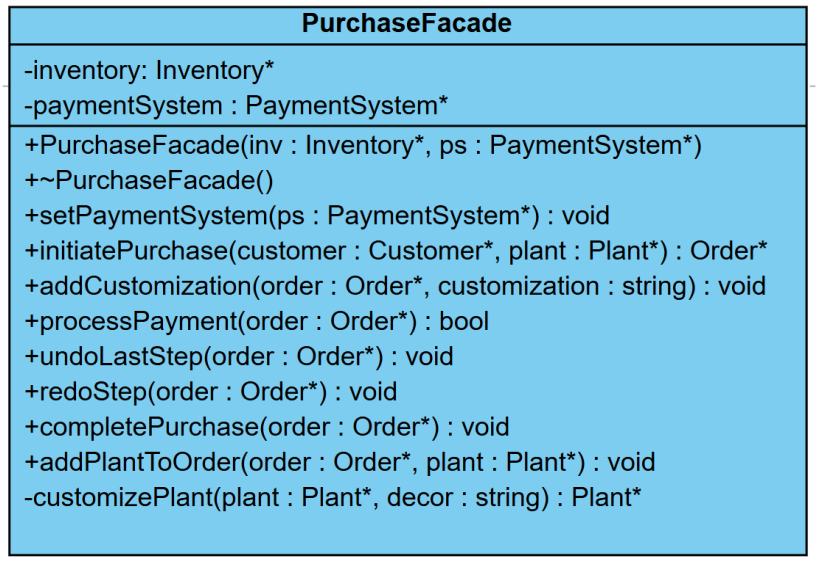




As we were developing the system we realized that Memento needed additional support in order to cater to saving snapshots for plants and their decorators. To improve the Memento pattern's ability to handle undo and redo functionality for customer orders, particularly for plants and their decorators, we integrated two Prototype patterns. One for managing the cloning of plant objects, while a separate decorator prototype handles cloning of decorative elements (e.g. pots and wrapping).

This ensures that snapshots saved by the Memento pattern accurately capture the state of both plants and their customizations.

Simplified Customer Interface (Facade)



To make it easier for customers to navigate the system, build their orders and pay, we used the Facade pattern. The PurchaseFacade class acts like a unified user interface. It hides the complex inner workings of inventory, payment and customization systems. It ties everything together smoothly so that customers can create and check out their orders easily.

Functional Requirements

1. **Greenhouse & Inventory**

    a. The system must create different plant types (succulents, flowers, shrubs) using the Factory Method pattern

    b. Plants must transition through life cycle states (Seeding, Growing, Mature, Moult, Dead) using the State pattern

    c. Different plant types must have customisable care strategies (Sunlight, Watering, Fertilizing) using Strategy pattern

    d. The inventory must organise plants using Composite pattern for hierarchical management

    e. The system must provide inventory traversal using Iterator pattern for plant browsing

2. **Staff Responsibilities**

    a. Staff must assist customers by handling customer requests through Chain of Responsibility pattern (SalesAssociate → InventoryClerk → Horticulturist)

    b. Staff must execute plant care operations according to their unique care routines (watering, fertilizing, sunlight management, pruning, etc) using the Command pattern (GreenHouseCommand)

    c. Staff must process sales transactions and maintain plant displays using the Command pattern (SalesCommand)

    d. Staff must provide inventory information to the customer using the Command pattern (InventoryCommand)

    e. Staff must handle escalated requests using the Command pattern (ManagerCommand)

3. **Customer Interactions**

    a. Customers must be able to filter plants by type, price range, or care level

    b. Customers must be able to request information about plants

    c. Customers must be able to browse plants using the Iterator pattern through inventory

d. Customers must be able to customise plants with decorations (Pots, Wrapping, Arrangements) using the Decorator pattern

e. Customers must be able to undo their order modifications using the Memento pattern

f. Customers must be able to receive assistance from staff through a structured request system

g. Customers must complete purchases through a simplified interface using the Facade pattern

4. System Coordination

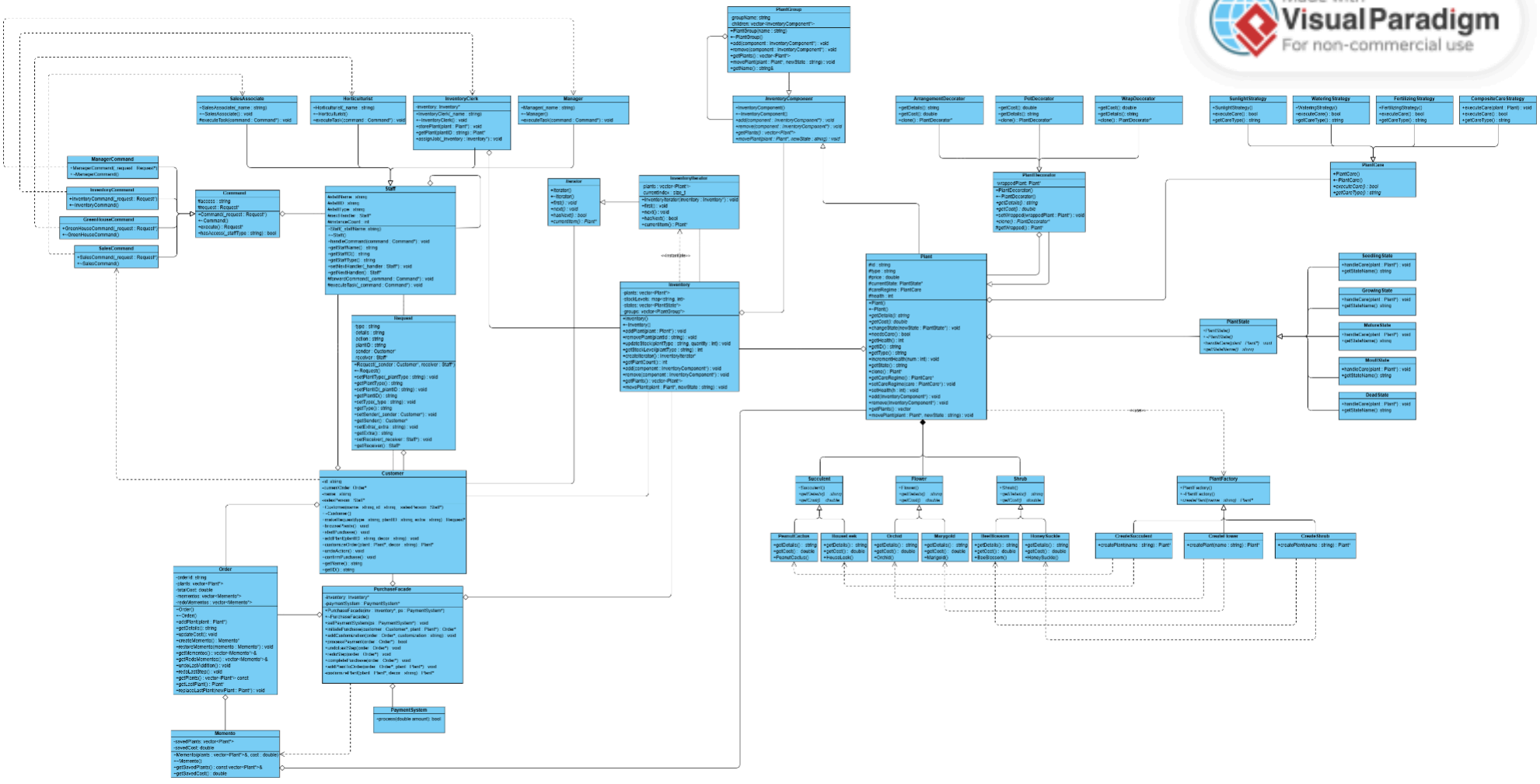
a. The system must coordinate purchase processes through the Facade pattern

b. The system must maintain order history for undo functionality using the Memento pattern

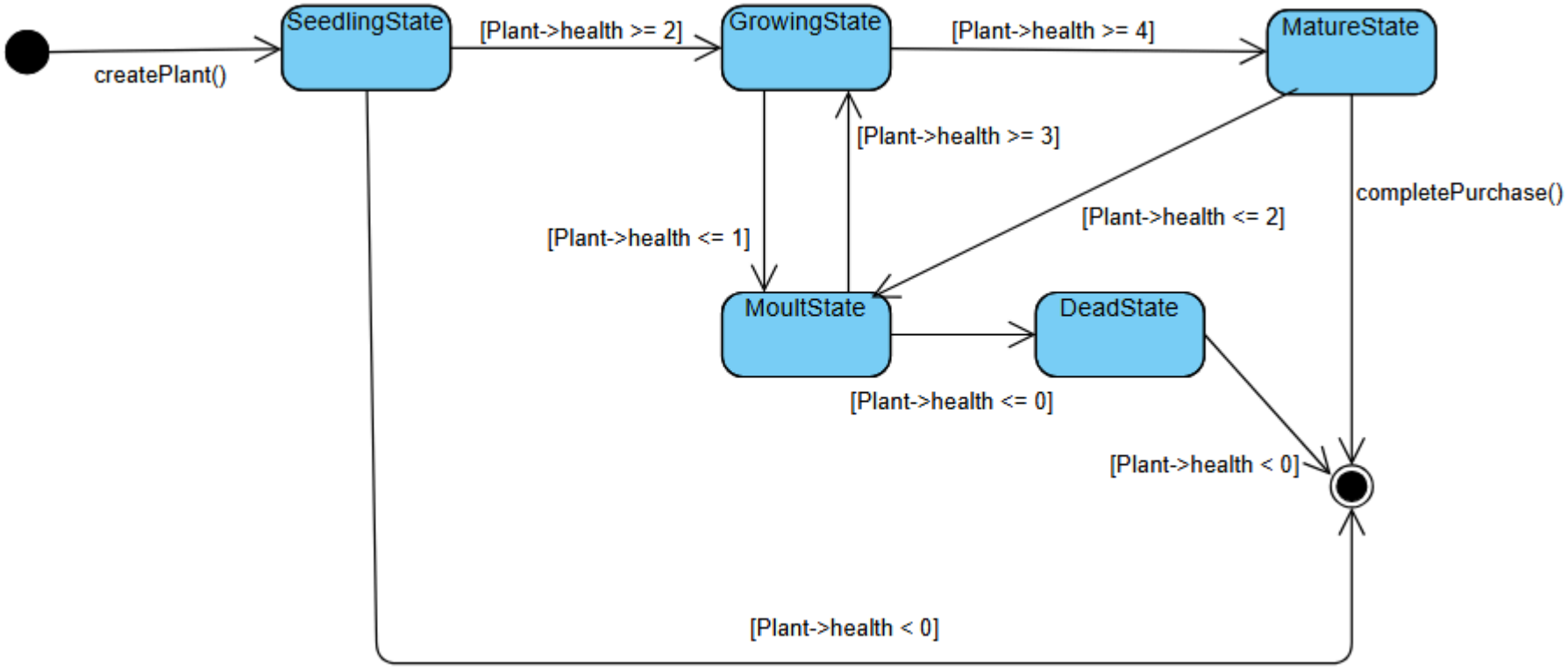
Non-Functional Requirements (NFR)

No.	Requirement	Quality Attribute
NFR 1	The system must support up to 50 plants and 10 users without performance degradation.	Scalability
NFR 2	The system must demonstrate clear separation of concerns through pattern responsibilities	Modularity
NFR 3	The interface should respond to user commands within 1 second.	Performance
NFR 4	Design patterns must allow extension without modifying existing core classes (Open/Closed Principle)	Extensibility / Maintainability
NFR 5	Doxygen generated documentation must describe all public classes and methods.	Documentation / Usability
NFR 6	The system must compile without errors using the provided Makefile and support modular development	Reliability

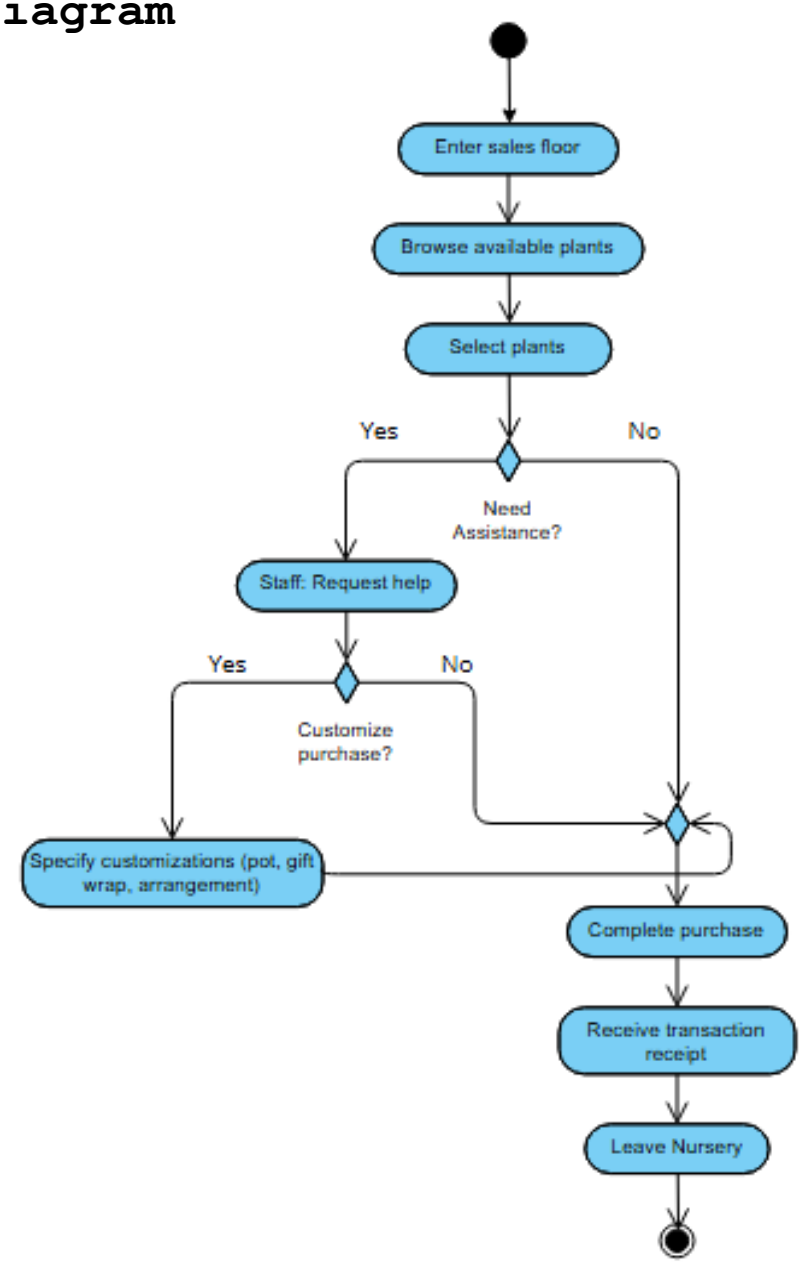
UML Class Diagram



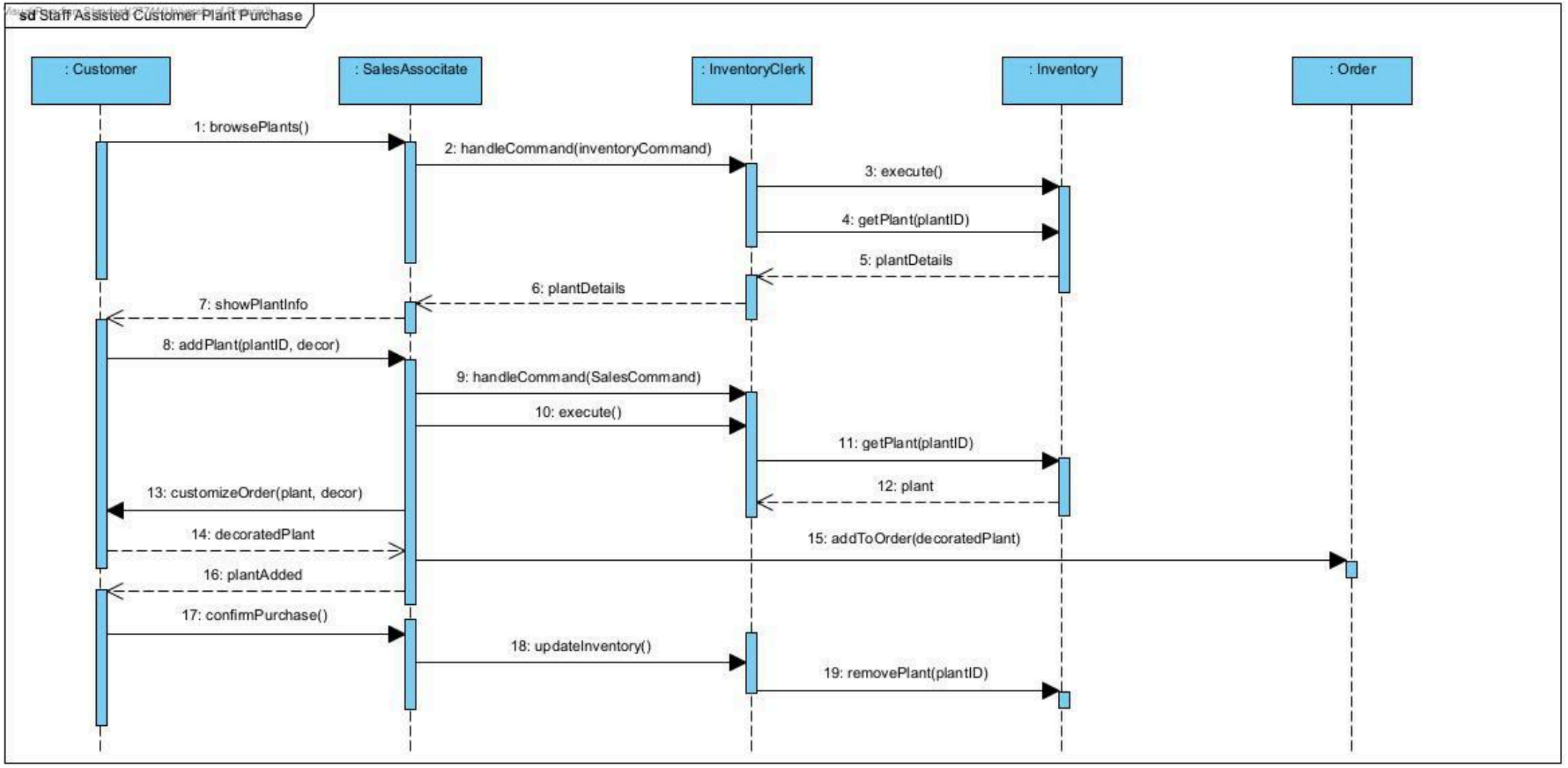
UML State Diagram



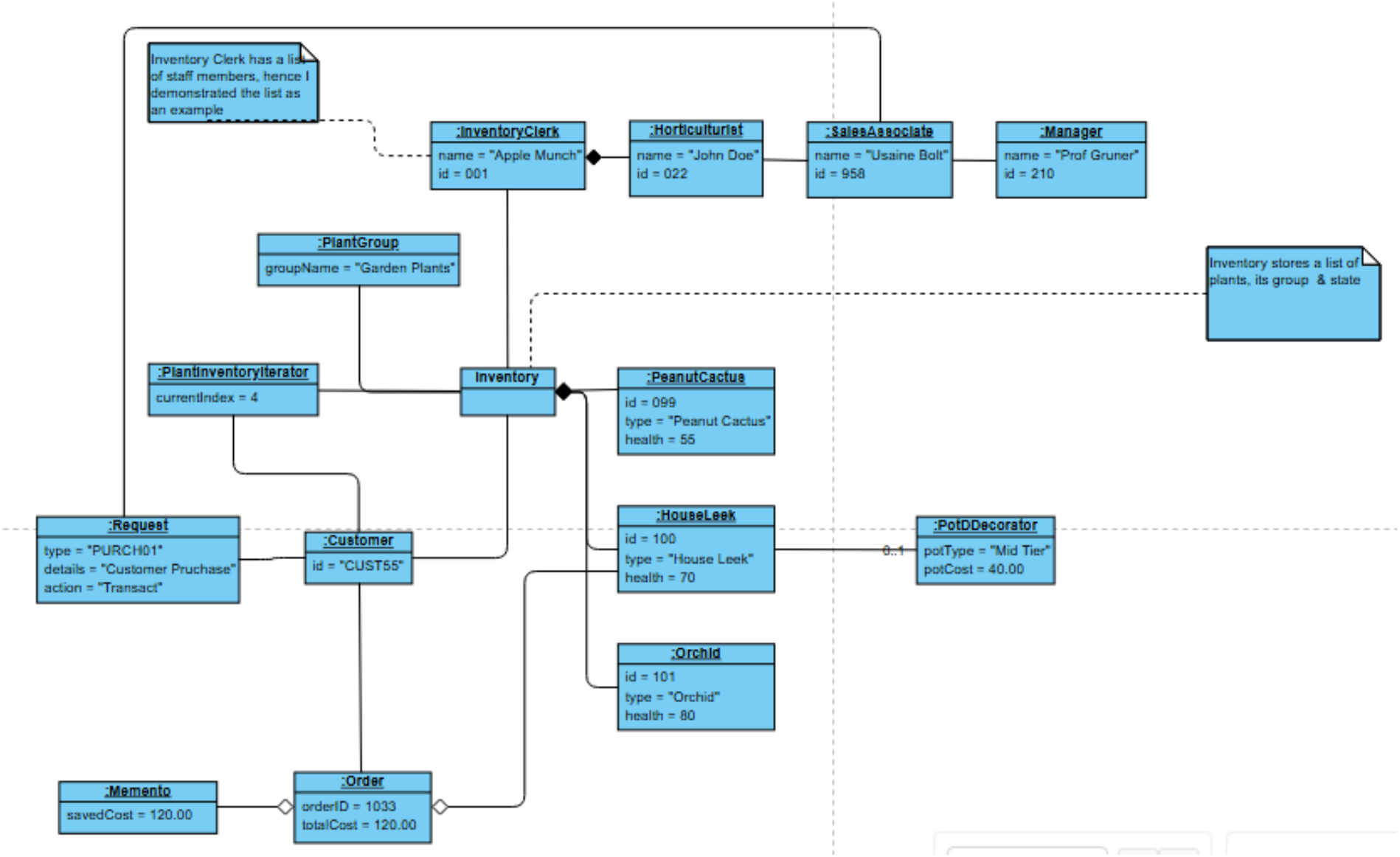
UML Activity Diagram



UML Sequence Diagram



UML Object Diagram



UML Communication Diagram