Task 4: Report

Group Members:
Megan Azmanov - u24575292
Gabi De Gouveia - u24594084
Gabriela Berimbau - u24711013
Rachel Clifford - u24647374
Kyle McCalgan - u24648826
Kahlan Hagerman – u24601358
Sofia Finlayson - u24593240

Research Brief:

In designing a realistic Plant Nursery Simulator, we researched real-world nursery management practices, focusing on plant care routines, staff coordination and, customer interaction. The goal was to align our simulator with real business and biological logic, ensuring that design patterns were chosen based on real functional needs.

Plant Care Routines:
Different plant types have widely varying requirements for water, sunlight, nutrients and pruning. A poor execution of a plants needs could result in a very unprofitable Plant Nursery, thus we divided our plant care routines into four main categories (Succulents, Flowers, Vegetables, and Other Plants), each linked to its own CareStrategy class for accurate and efficient maintenance.

- Succulents require 15 units of water, +10 nutrients, 85% sunlight, and occasional pruning of dead leaves.
- Flowers require 50 units of water, +20 nutrients, 70% sunlight, and regular pruning to promote continued flowering.
- Vegetables require 100 units of water, +25 nutrients, 75% sunlight, and routine pruning of old growth.
- Other (General) Plants require 20 units of water, +10 nutrients, 60% sunlight, with light pruning for overall health.

All measurements operate on a relative 0-100 unit scale:
Water: 0-100 units → 0-5 liters/week
Nutrients: 0-100 units → 0-50g fertilizer/week
Sunlight: 0-100 units → % of daylight exposure
This system ensures consistency, accuracy, and controlled testing in the simulation, directly informing the behaviour of each CareStrategy class.

Nursery Operations:
The Mediator Pattern reflects real-world supervision by coordinating communication between greenhouse staff, sales assistants, and the nursery owner. The Chain of Responsibility Pattern ensures that requests are escalated through the appropriate staff hierarchy so that customer service is efficient. Plant lifecycles are modelled using the State Pattern, indicating when a plant is moved to the sales floor during flowering, or removed from inventory once sold or deceased.

Customer Interaction and Sales:
Product personalization enhances customer engagement and satisfaction. The Decorator Pattern enables customers to add optional features to purchases (Ribbons, Decorative Pots, Gift Wrap) dynamically at checkout . Additionally, the Template Method Pattern enforces a consistent transaction workflow while allowing multiple payment types (Cash, Credit Card). This reflects real-world retail processes where the payment sequence is fixed but payment method flexibility improves convenience.

Influence & Assumptions:
Environmental variables such as sunlight exposure, and soil fertility are represented as fixed simulation constants to maintain experimental control. Fertiliser and water units are proportional values, not physical measurements.
Customer interactions in the simulator occur in a shared environment where all staff members can collaborate through the Mediator and Chain or Responsibility patterns.
The simulator operates on a time-scaled model, where one simulated “day” equals approximately one real-time minute, to allow for continuous refreshing of plant growth data, lifecycle transitions and sales floor updates.
Together, these research insights ensured that every aspect of the Plant Nursery Simulator reflects nursery behaviour and system design.

References:
Innovative Metal Solutions (2020). *Quick Nursery Equipment Guide on Propagating, Handling & Growing Plants*
Indeed (2025). *Product Customization: What It Is and Benefits*
Royal Horticultural Society (RHS). (2023). *Plant Care Basics.*
South African Nursery Association (SANA). (2024). *Nursery Management Guide.*

Design Pattern Application:

The Nursery Management System implements 12 design patterns to model a realistic plant nursery operation. Each pattern addresses specific functional requirements and implementations to ensure maintainability and extensibility within the system.

Factory Method: Creational

Purpose:
Factory Method creates different types of plants (Strelitzia, Monstera, VenusFlyTrap, Cactus, Aloe, Potato, Radish, Rose, Daisy) without specifying the exact classes. The system needs to create various plant types without coupling client code to concrete plant classes. The Factory Method pattern provides a consistent interface for plant creation and encapsulates initialization logic, such as assigning appropriate state and strategy.

Implementation:

- Creator: PlantFactory, OtherPlantFactory, SucculentFactory, VegetableFactory, FlowerFactory
- Concrete Creator: StrelitziaFactory, MonsteraFactory, VenusFlyTrapFactory, CactusFactory, AloeFactory, PotatoFactory, RadishFactory, RoseFactory, DaisyFactory
- Product: Plant, Flower, Vegetable, Succulent, OtherPlant
- Concrete Product: Daisy, Rose, Radish, Potato, Aloe, Cactus, VenusFlyTrap, Monstera, Strelitzia

Design Decision:
Each Concrete Creator creates one specific plant type. It assigns the correct care strategy, sets the initial plant state, configures plant-specific attributes, and returns a fully initialised plant.

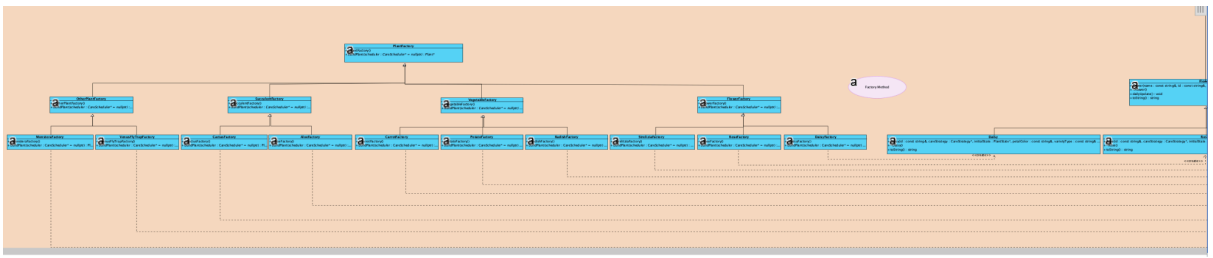
Functional Requirements:

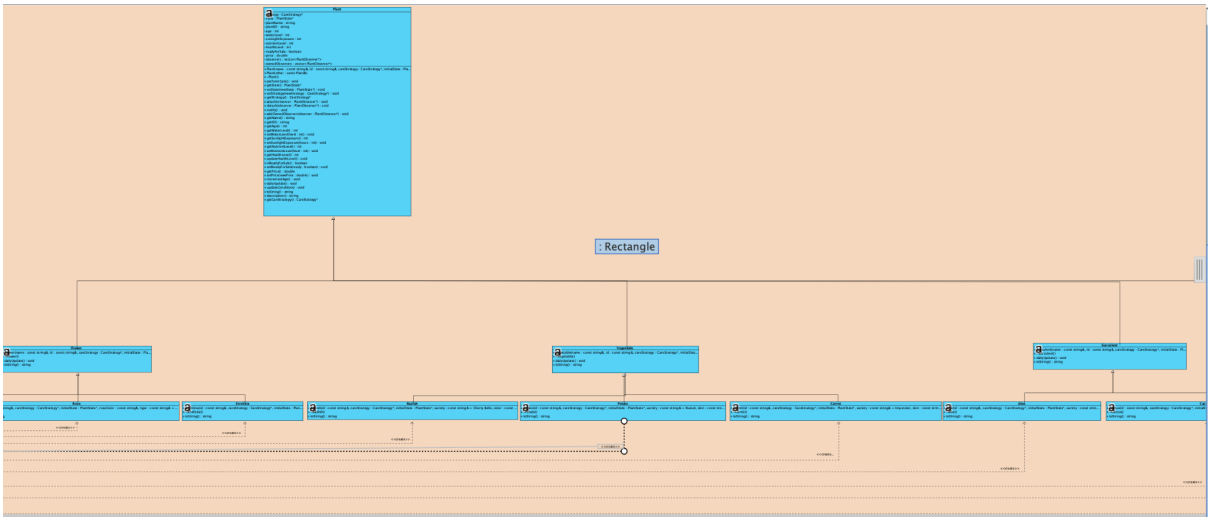
- Factory-Based Plant Creation: Plants are created using the Factory Method, and each class is responsible for instantiating a specific plant type. This ensures consistency in the creation of plants.

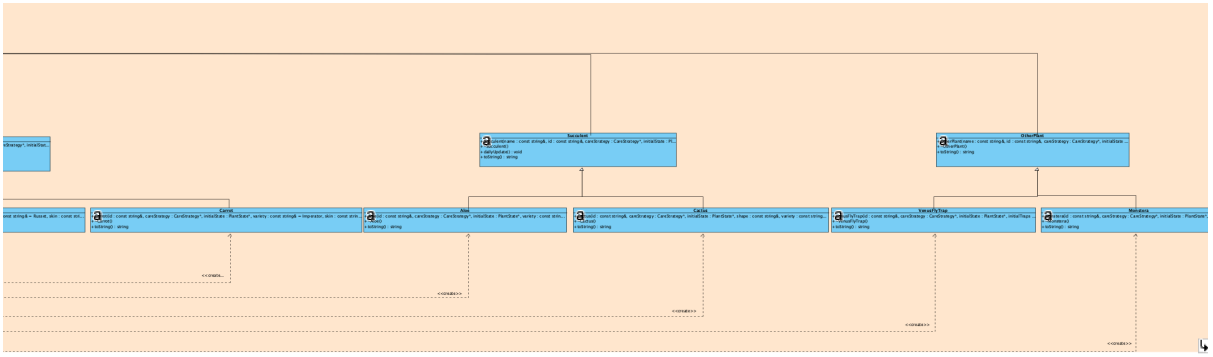
Factory Usage Flow:

1. Client requests a new plant via factory
2. Factory creates a plant object
3. Factory assigns the appropriate attributes to the plant
4. Factory returns the initialised plant

Key Methods:
`Plant* PlantFactory::buildPart()`







State: Behavioural

Purpose:
State manages the plant's lifecycle stages with different behaviours and conditions for each state.
Different states of a plant include stages of seedling, growing, maturing, flowering, and a dead state that signals the end of the plant's lifecycle. Each state has unique behaviours, care requirements, and transition conditions. The State design pattern eliminates complex conditional logic and makes state-specific behaviour explicit.

Implementation:

- Context: Plant
- State Interface: PlantState
- Concrete States: SeedlingState, GrowingState, MatureState, FloweringState, DeadState

Design Decision:
States manage their own transitions based on plant age, health, and environmental conditions. When transitioning, the old state is deleted and is replaced by the new state, which ensures clean memory management. The Plant's readyForSale status updates automatically based on the state mature and flowering.

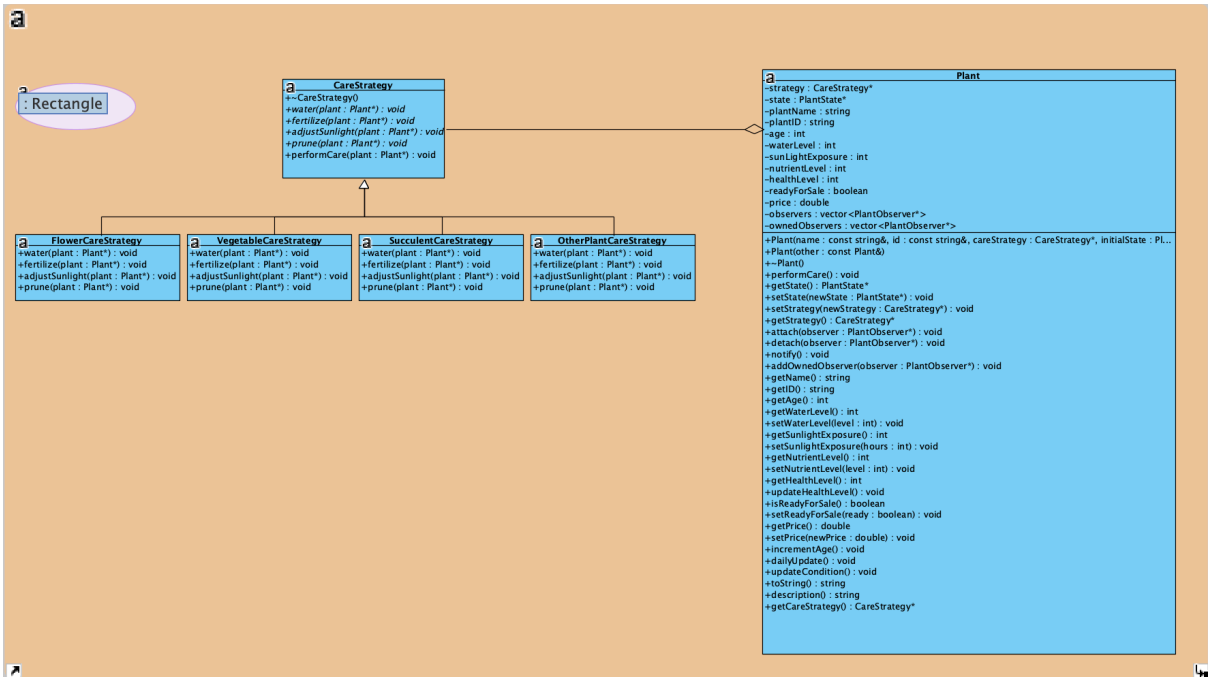
Functional Requirements:

- Plant Lifecycle Management: The system must track each plant's lifecycle stages and allow state transitions automatically based on age and care.

Transition Logic:

- Seedling to Growing: age >= 7 days and health >= 50
- Growing to Mature: age >= 12 days and health >= 50
- Mature to Flowering: age >= 35 days and health >= 80
- Flowering to Mature: age >= 50 days (transitions from blooming back to the mature state)
- Any state to Dead: health < 10 or 20 depending on the state

Key Methods:
void PlantState::handleChange(Plant* plant)
std::string PlantState::getStateName()
void Plant::setState(PlantState* newState)



Strategy: Behavioural

Purpose:
Strategy encapsulates the different plant care algorithms required for each plant type. Different plant types require distinct care approaches.

Different care strategies are determined by each plant type such as succulents, vegetables, flowers, and other plants. Each care strategy will include watering, fertilizing, adjusting the sunlight, and pruning. These levels are different for each plant type, hence why different care strategies are used.
The Strategy pattern encapsulates these varying care algorithms which allows for runtime flexibility and eliminating conditional logic for plant-specific care.
Implementation:

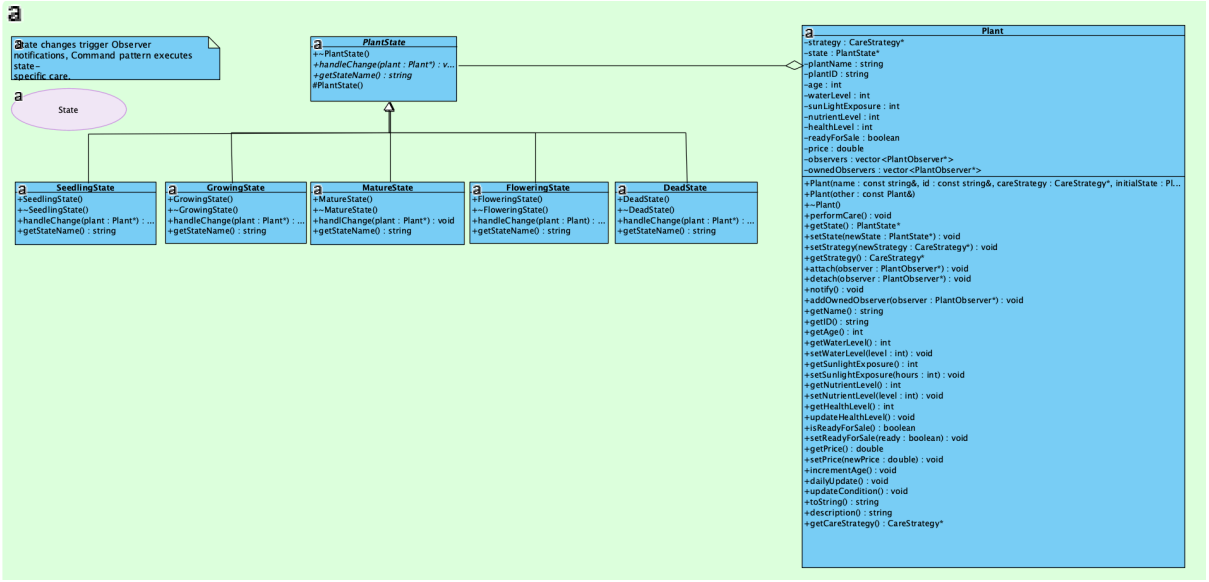
- Context: Plant
- Strategy Interface: CareStrategy
- Concrete Strategies: SucculentCareStrategy, VegetableCareStrategy, FlowerCareStrategy, and OtherPlantCareStrategy

Design Decision:
Each strategy encapsulates the complete care algorithm for a plant type, including watering frequency, fertilization needs, sunlight requirements, and pruning techniques. Strategies can be swapped at runtime if a plant's needs change.

Functional Requirements:

- Dynamic Plant Care System: different plant types (succulents, flowers, vegetables, and other plants) require unique care routines that can change dynamically.

Key Methods:
void CareStrategy::water(Plant* plant)
void CareStrategy::fertilize(Plant* plant)
void CareStrategy::adjustSunlight(Plant* plant)
void CareStrategy::prune(Plant* plant)



Chain of Responsibility: Behavioural

Purpose:
Chain of Responsibility passes the customer's requests through a hierarchy of handlers. Each level in the chain of responsibility, handles a different difficulty of requests. Customer requests vary in complexity from simple purchases to bulk orders and special arrangements. The Chain of Responsibility design pattern allows requests to be passed through a hierarchy of handlers, each deciding whether to handle the request or escalate it.

Implementation:

- Handler: StaffMembers
- Concrete Handlers: SalesAssistant, FloorManager, NurseryOwner

Design Decision:
The Request class parses its own text to determine complexity level. It looks for keywords such as "bulk", "wedding", "special". This self-awareness allows the chain to route efficiently. Staff members inherit from both StaffMembers and Person.

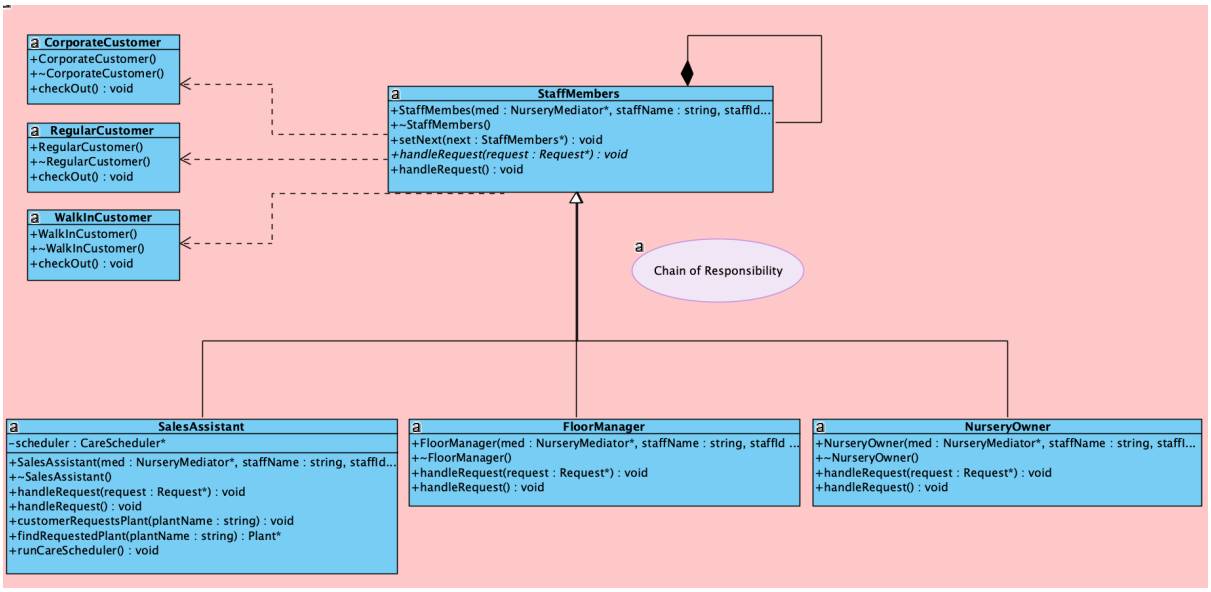
Functional Requirements:

- Coordinating Staff Operations: All communication between greenhouse, sales floor, and customers is handled via a centralized mediator.

Request Routing:

- "I would like a rose" is handled by SalesAssistant
- "I need 50 plants for a wedding" is handled by FloorManager
- "I have a complaint about the quality of my plant" is handled by the NurseryOwner

Key Methods:
void Request::parseRequest()
void StaffMembers::handleRequest(Request* request)
void StaffMembers::setNext(StaffMembers* next)



Mediator: Behavioural

Purpose:
Mediator centralizes complex communication between different areas of the nursery and the staff personnel. Mediator handles different interactions between the nursery, customers, staff, and plants. The nursery involves complex interactions between multiple classes such as GreenHouse, SalesFloor, Customers, and Staff. Direct communication would create tight coupling and make the system difficult to maintain. The Mediator pattern centralizes communication logic.

Implementation:

- Mediator: NurseryMediator
- ConcreteMediator: NurseryCoordinator
- Colleague: Colleague
- Concrete Colleagues: GreenHouse, SalesFloor, Person, Customer, CorporateCustomer, RegularCustomer, WalkinCustomer, StaffMembers, SalesAssistant, FloorManager, NurseryOwner

Design Decision:
All colleagues communicate exclusively through the mediator rather than directly with each other. This prevents a web of interdependencies between classes and makes it easier to add new colleagues or modify interactions within the system without affecting existing classes. The mediator also loosens coupling between classes.

Functional Requirements:

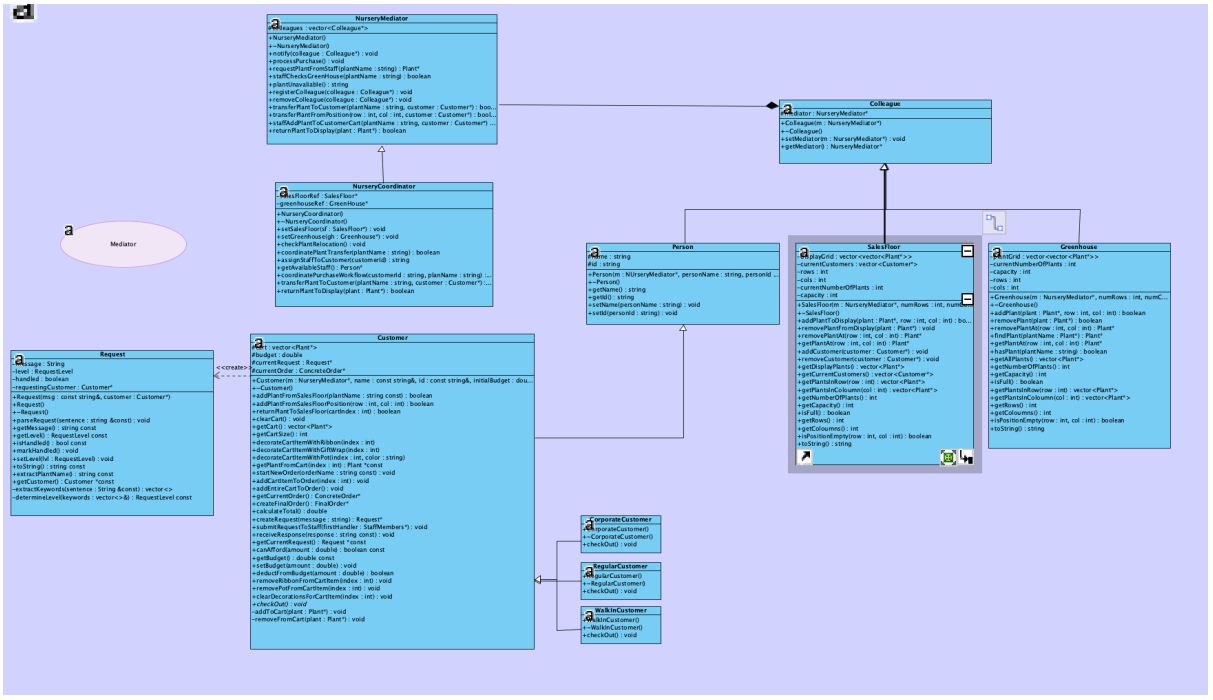
- Coordinating Staff Operations: All communication between the greenhouse, sales floor, and customers is handled via a centralised mediator.

Communication Flow:

1. Customer requests a plant through Mediator
2. Mediator notifies SalesAssistant colleague
3. SalesAssistant asks Mediator to check SalesFloor colleague
4. If the plant is not found, Mediator coordinates with the staff to check the Greenhouse colleague
5. Mediator handles response back to Customer colleague

Key Methods:

```
void NurseryMediator::notify(Colleague* sender, string event)
void NurseryMediator::processPurchase(Customer* customer, FinalOrder* order)
Plant* NurseryMediator::requestPlantFromStaff(string plantName)
bool NurseryMediator::staffChecksGreenHouse(string plantName)
```



Decorator: Structural
Purpose:
Decorator dynamically adds features to individual plants, such as ribbons, gift wraps, and a decorative pot. Customers want to customize individual plants with add-ons such as ribbons, gift wraps, and decorative pots without creating other plant subclasses. The Decorator design pattern dynamically adds functionality whilst maintaining the Plant interface.

Implementation:

- Component: Plant
- Concrete Component: Daisy, Rose, Radish, Potato, Aloe, Cactus, VenusFlyTrap, Monstera, Strelitzia
- Decorator: Decorator
- Concrete Decorators: RibbonDecorator, GiftWrapDecorator, DecorativePotDecorator

Design Decision:
Decorators wrap Plant objects and maintain the same interface, allowing decorated plants to be used anywhere that regular plants can. Multiple decorators can be stacked on a single plant. The price and description methods are overridden to include decoration costs and details.

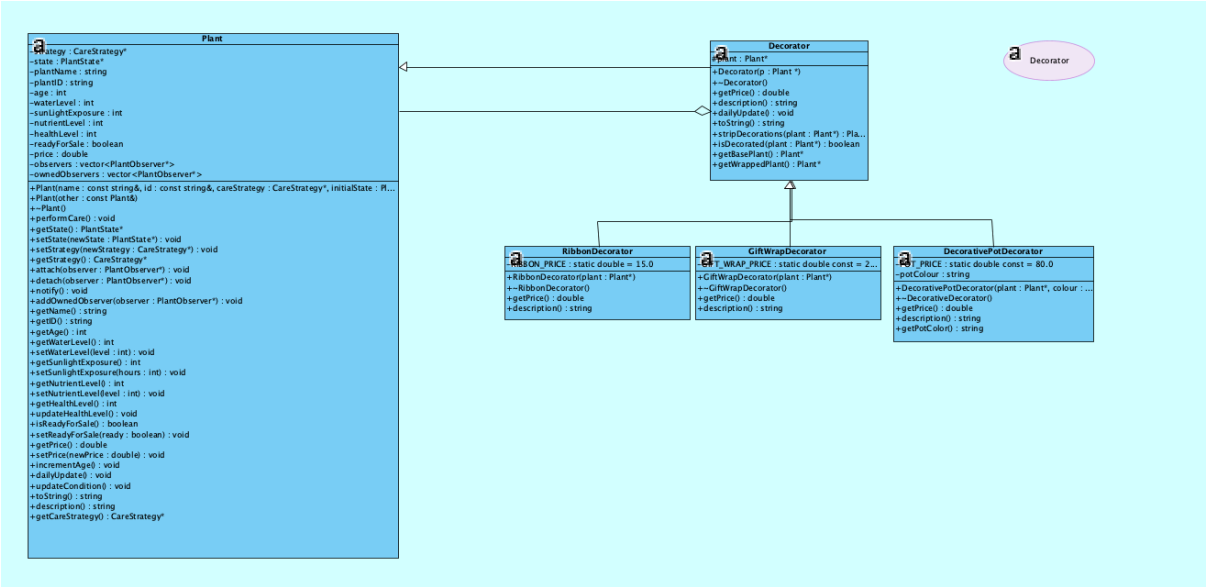
Functional Requirements:

- Product Customisation: Allows customers to enhance their plant purchase with decorations and packaging.

Decoration Examples:

```
Plant* plainRose = new Rose();
Plant* fancyRose = new DecorativePotDecorator(new Rose());
Plant* giftRose = new RibbonDecorator(new GiftWrapDecorator(new Rose()));
```

Key Methods:
double Decorator::getPrice() override
string Decorator::getDescription() override



Composite: Structural

Purpose:
Composite builds complex order structures with individual items and groups. Customers create orders ranging from single plants to complex nested structures. The Composite design pattern treats individual items and groups uniformly hence, simplifying order management and price calculation.

Implementation:

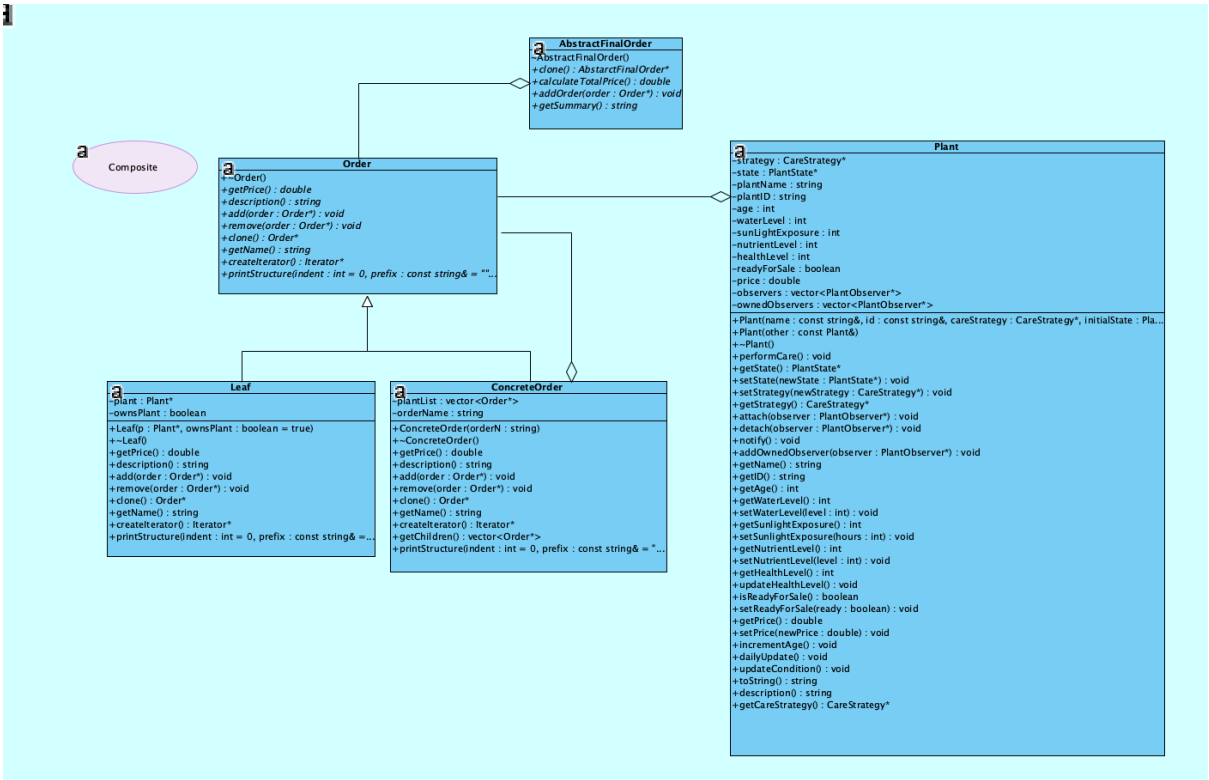
- Component: Order
- Leaf: Leaf
- Composite: ConcreteOrder

Design Decision:
The Leaf wraps a single Plant reference (which may be decorated) and quantity. The ConcreteOrder contains a vector of Order children from Leaf or ConcreteOrder. The recursive price calculation present in ConcreteOrder sums all of the children's prices. It supports both complex and individual order hierarchies.

Functional Requirements:

- Composite Order Structure: The system supports both individual and grouped plant orders using a composite structure.
- System Integration and Consistency: All subsystems (Greenhouse, SalesFloor, Staff, Customer, Payment, Order) interact through a single orchestrated flow managed by the Mediator

Key Methods:
double Order::getPrice()
string Order::getDescription()
void Order::add(Order* order)
void Order::remove(Order* order)



Template Method: Behavioural

Purpose:
Template Method defines the payment processing skeleton, such as credit card or cash payment with variable steps. Payment processing follows a consistent workflow regardless of payment type, but specific payment steps (cash or credit card) differ. The Template Method design pattern defines the payment skeleton whilst allowing subclasses to customise specific steps in the payment process.

Implementation:

- Abstract Class: PaymentProcessor
- Concrete Class: CreditCardPayment, CashPayment

Design Decision:
The template method processTransaction() is to enforce the payment workflow.

1. Validate the payment amount
2. Process the payment through the subclasses
3. Generate a receipt
4. Update the inventory

Functional Requirements:

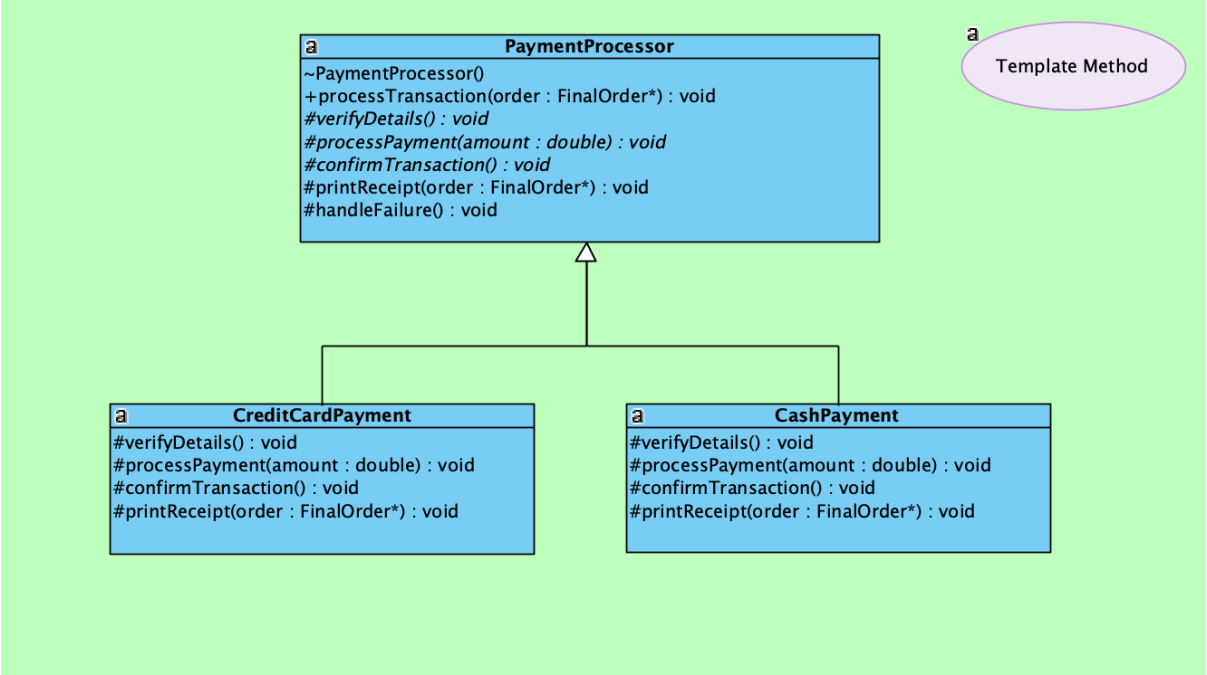
- Checkout and Payment Processing: Payments must follow a consistent process whilst allowing different methods such as cash payment or credit card payment
- System Integration and Consistency: All subsystems (Greenhouse, SalesFloor, Staff, Customer, Payment, Order) interact through a single orchestrated flow managed by the Mediator

Payment Flow:

```
PaymentProcessor* processor = new CashPayment(finalOrder);
processor->processTransaction();
```

Key Methods:

```
void PaymentProcessor::processTransaction(FinalOrder* order)
```



Observer: Behavioural

Purpose:
Observer monitors plant conditions and triggers notifications when care is needed for a plant without tight coupling between plants and care system strategies. It tracks each plant's lifecycle stages as seen in the State design pattern (SeedlingState, GrowingState, MatureState, FloweringState, and DeadState).
This allows the correct care strategy to be used in the different states of each plant type.

Implementation:

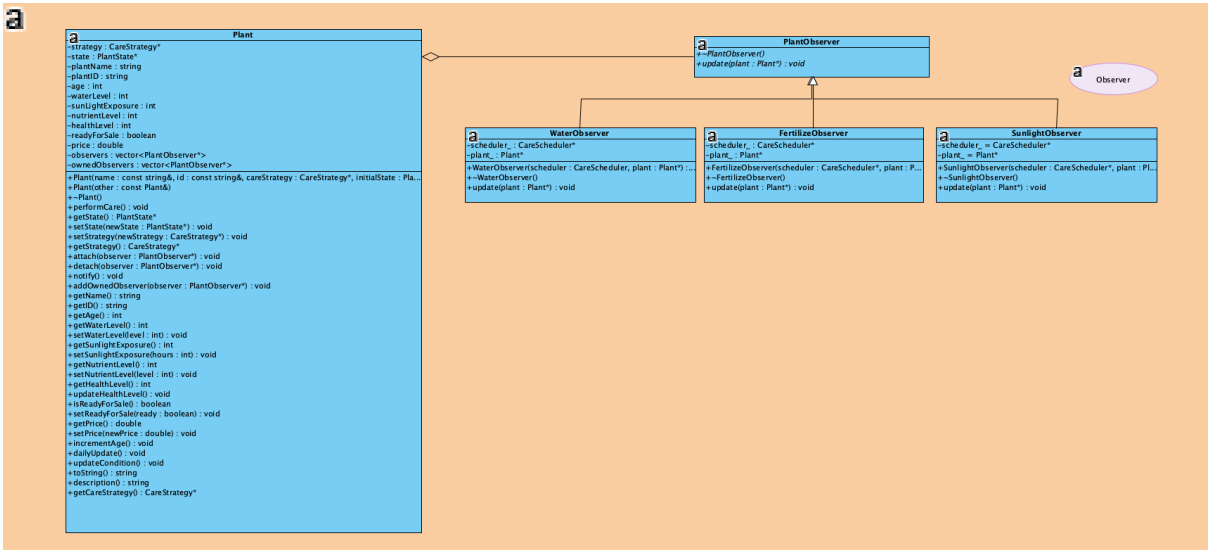
- Subject: Plant
- Observer: PlantObserver
- Concrete Observer: WaterObserver, FertilizeObserver, SunlightObserver

Design Decision:
The Plant class implements the Subject behaviour directly. This simplifies the design by eliminating an abstraction layer whilst maintaining the Observer design pattern.

Functional Requirements:

- Automatic Plant Monitoring: The system automatically detects when a plant needs water, fertilizer, or sunlight and it triggers care commands.

Key methods:
void Plant::attach(PlantObserver* observer)
void Plant::detach(PlantObserver* observer)
void Plant::notifyObservers()
void PlantObserver::update(Plant* plant)



Command: Behavioural

Purpose:
Command encapsulates plant care actions as objects for execution and scheduling. Command calls different care strategies required for a plant type. Plant care actions need to be scheduled, queued, logged, and potentially undone. The Command design pattern encapsulates care actions as objects, enabling flexible execution management and decoupling action requests from execution.

Implementation:

- Command Interface: Command
- Concrete Commands: WaterPlantCommand, FertilizePlantCommand, AdjustSunlightCommand
- Invoker: CareScheduler
- Receiver: Plant

Design Decision:
Observers create and schedule commands when they detect plant needs. The CareScheduler maintains a queue of commands and executes them systematically. This architecture supports future features like command logging, undoing functionality, and batch processing.

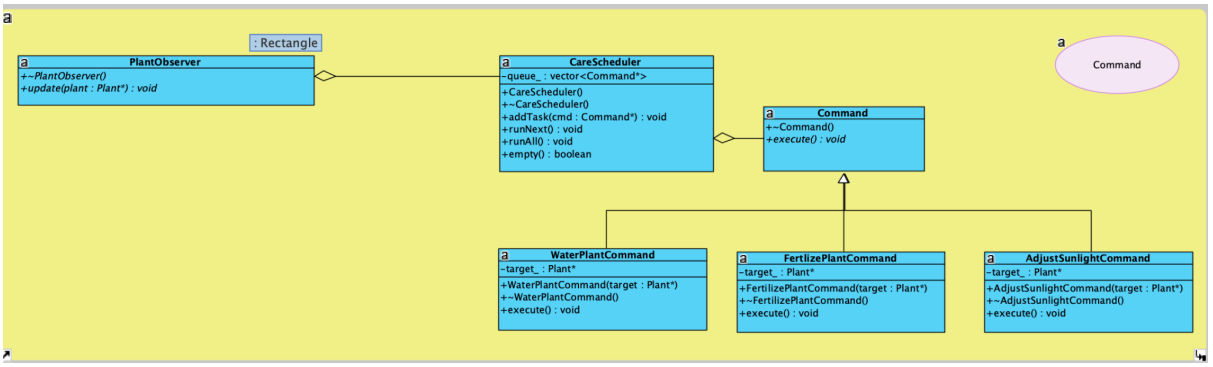
Functional Requirements:

- Automatic Plant Monitoring: The system automatically detects when a plant needs water, fertilizer, or sunlight and it triggers care commands.

Key Methods:
void Command::execute()
void CareScheduler::scheduleCommand(Command* command)
void CareScheduler::executeNext()
void CareScheduler::executeAll()

Workflow:

1. Observer detects a low water level
2. Observer creates WaterPlantCommand
3. Observer schedules a command with CareScheduler
4. CareScheduler executes command when appropriate
5. Command calls plant's strategy's water method



Prototype: Creational

Purpose:
Prototype clones an entire order structure for repeat customers. Repeat customers frequently reorder the same complex order structures. The Prototype design pattern allows cloning entire FinalOrder objects with all nested ConcreteOrders and Leafs, hence saving time and effort by cloning the entire order hierarchy.
Implementation:

- Prototype: AbstractFinalOrder
- Concrete Prototype: FinalOrder

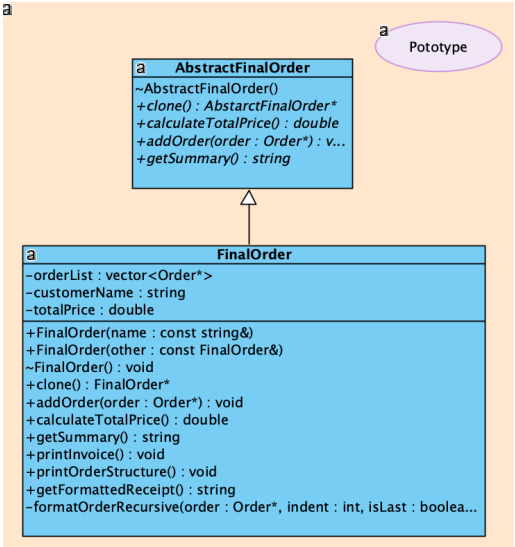
Design Decision:
FinalOrder acts as a container for the complete order structure. The clone() method performs a deep copy of all Order components, preserving the entire hierarchy including all

decorated plants and nested structures. Customers can modify the cloned order before purchasing.

Functional Requirements:

- System Integration and Consistency: All subsystems (Greenhouse, SalesFloor, Staff, Customer, Payment, Order) interact through a single orchestrated flow managed by the Mediator
- Order Cloning: Customers can quickly reorder previous purchases through deep cloning of complex and simple order structures

Key Methods:
AbstractFinalOrder* FinalOrder::clone()
double FinalOrder::calculateTotalPrice()
void FinalOrder::addOrder(Order* order)



Iterator: Behavioural

Purpose:
Iterator iterates through the Composite order structure and accesses each individual order, regardless of its complexity.

Implementation:

- Aggregate: Order
- Concrete Aggregate: ConcreteOrder
- Iterator: Iterator
- Concrete Iterator: ConcreteIterator

Design Decision:
The Iterator design pattern allows the traversal of complex order structures without exposing its underlying, internal representation. It supports both simple and complex order hierarchies.

Functional Requirements:

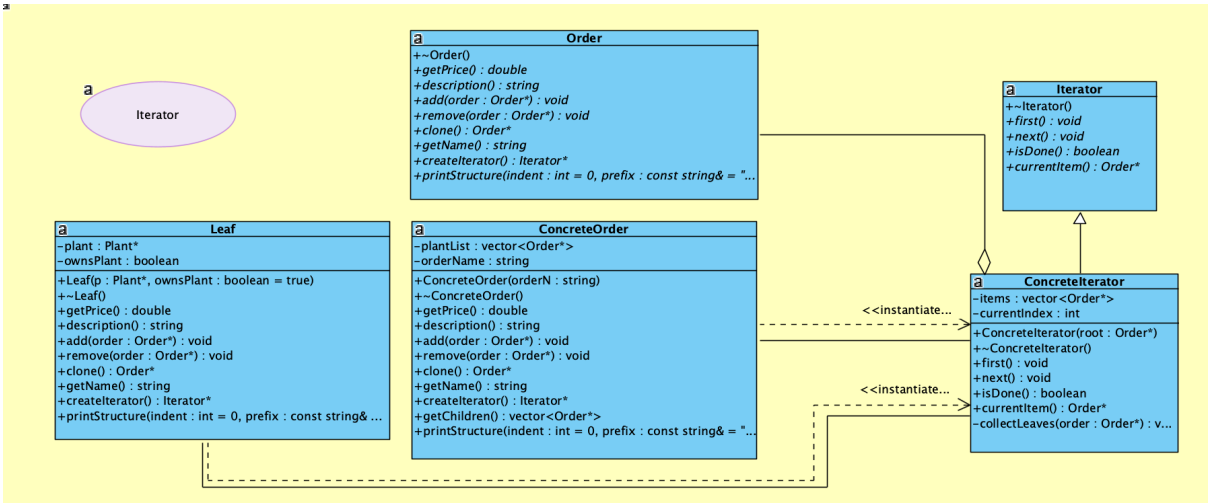
- Composite Order Structure: The system supports both individual and grouped plant orders using a composite structure.
- Order Cloning: Customers can quickly reorder previous purchases through deep cloning of complex and simple order structures

Iterator Pattern Integration:

```
Iterator* it = order->createIterator();
for (it->first(); !it->isDone(); it->next()) {
Order* item = it->currentItem(){
}
```

Key Methods:

```
Iterator* Order::createIterator()
```



System Flow Example: Complete Purchase Workflow

1. Factory Method creates a Rose with FlowerCareStrategy and in SeedlingState
2. The plant is placed in the Greenhouse (Mediator colleague)
3. Observer monitors the plant daily, and creates Commands when care is needed
4. CareScheduler executes commands using the plant's Strategy
5. State transitions automatically from SeedlingState to GrowingState to MatureState to FloweringState
6. When Mature, the Mediator coordinates the plant to move to the SalesFloor
7. A Customer (colleague) requests the plant through the Mediator
8. Request object parses text, and enters the Chain of Responsibility
9. The SalesAssistant handles a simple request via the Mediator
10. The Customer customises the plant using the Decorator (adding a ribbon and gift wrap)
11. The decorated plant is wrapped in Leaf, and added to a ConcreteOrder (Composite)
12. Multiple items are added, creating a nested structure, which is traversable by the Iterator
13. The ConcreteOrder is wrapped in the FinalOrder (Prototype-ready)
14. The Template Method processes the type of payment (either CashPayment of CreditCardPayment)
15. For repeat customers, Prototype clones the previous FinalOrder

Pattern Integration Benefits

1. Maintainability: Each pattern addresses specific concerns, making it easier to understand and modify
2. Extensibility: New plant types, care strategies, states, and decorations can be added without modifying existing code
3. Testability: Patterns create clear boundaries, allowing us to create isolated unit testing
4. Reusability: Components like factories, strategies, and commands can be reused across different situations
5. Flexibility: runtime behaviour modification through strategy swapping, decoration, and state transitions of the plant
6. Scalability: Mediator and Composite patterns support system growth

Subsystem 1: Customer and Nursery Interactions

- Design patterns: Mediator, Chain of Responsibility, Factory, Decorator, Composite, Iterator, Prototype, Template Method

Flow:

1. Customer creates a Request with a text description
2. Request parses itself and enters the Chain of Responsibility
3. Appropriate staff handlers request via Mediator coordination
4. Plants are retrieved via Factory Method or from inventory
5. Individual plants can be optionally Decorated with add-ons
6. Decorated or plain plants are wrapped in Leafs
7. Leafs are added to a ConcreteOrder (Composite structure)
8. The Order is traversable via the Iterator
9. A complete order is wrapped in a FinalOrder
10. The FinalOrder is able to be cloned via Prototype for repeat orders
11. Payment process via Template Method

Subsystem 2: Employees and Plants Interaction

- Design Patterns: Factory Method, State, Strategy, Observer, Command

Flow:

1. Plant created by Factory Method in a SeedlingState with a CareStrategy
2. Plant is placed in a GreenHouse grid
3. Observers monitor plant conditions continuously
4. When care is needed, Observers trigger Commands via the CareScheduler
5. Commands execute using the plant's CareStrategy
6. The plant progresses through different States
7. When MatureState is reached, the plant is moved to the SalesFloor grid

Subsystem 3: Employees and Customers Interaction

- Design Patterns: Mediator, Chain of Responsibility

Flow:

1. Customer requests a plant (Customer colleague communicates via the Mediator)
2. Mediator notifies a SalesAssistant colleague
3. SalesAssistant requests Mediator to check SalesFloor colleague
4. If not found, Mediator coordinates with staff to check the GreenHouse colleague
5. Complex requests are escalated via the Chain of Responsibility

Grid Management System:

GreenHouse Grid:

- Purpose: Cultivation area for plants in growth stages (SeedlingState, GrowingState)
- Structure: 2D array of Plant pointers
- Organisation: By Plant type, growth stage, or care requirements

SalesFloor Grid:

- Purpose: Retail area for mature plants that are ready for sale
- Structure: 2D array of Plant pointers
- Organisation: For customer browsing and easy access

Customer Types:

RegularCustomer:

- Standard customers with a purchase history
- Access to previous FinalOrders for cloning

WalkInCustomer:

- First-time or casual customer
- No purchase history

CorporateCustomer:

- Bulk orders and recurring purchases
- Special arrangements such as weddings and events
- May use Prototype Pattern extensively for weekly orders

State Transitions:

Transition:	Age Requirement:	Health Requirement:	Information:
Seedling to Growing	>= 7 days	>= 50%	Vulnerable stage, needs frequent care
Growing to Mature	>= 12 days total	>= 50%	Ready for sale, move to SalesFloor
Mature to Flowering	>= 35 days	>= 80%	Premium pricing applied
Flowering to Mature	>= 50 days	Any	Flowering period ends
Any State to Dead	Any age	< 20%	Terminal state, remove from inventory

Care Strategy Specifications:

SucculentCareStrategy:

- Water: Minimal (increases by a small amount)
- Fertilise: Light feeding
- Sunlight: High exposure preferred
- Prune: Remove dead leaves only

VegetableCareStrategy:

- Water: Regular schedule (moderate increase)
- Fertilise: Nutrient-rich feeding
- Sunlight: Full sun exposure
- Prune: Remove old growth and suckers

FlowerCareStrategy:

- Water: Optimal moisture maintenance
- Fertilise: Bloom-boosting nutrients
- Sunlight: Optimal light for blooming
- Prune: Deadheading spent blooms

OtherPlantCareStrategy:

- Water: Standard amount
- Fertilise: General purpose
- Sunlight: Moderate
- Prune: Basic maintenance

Decorator Pricing:

Decorator:	Price:	Description:
RibbonDecorator	+R15	Decorative ribbon around pot
GiftWrapDecorator	+R20	Full gift wrapping
DecorativePotDecorator	+R80	Premium decorative pot upgrade

Memory Management Considerations:

State Pattern:

- States delete themselves during transitions
- Plant destructor deletes the final state when the plant is destroyed

Strategy Pattern:

- Plant owns its strategy and deletes in the destructor

Observer Pattern:

- Plant does not own observers
- Observers must be cleaned up externally
- Use careful lifecycle management

Composite Pattern:

- ConcreteOwner owns its children
- Deleting ConcreteOrder recursively deletes all children
- Leaf does not own the plant

Decorator Pattern:

- Decorator owns the wrapped plant
- Deleting decorator deletes the wrapped plant

Testing Strategies:

Unit Tests:

- Individual pattern components (states, strategies, commands)
- Factory plant creation
- Decorator piece calculations
- State transitions

Integration Tests:

- Observer and Command workflow
- Mediator coordination

Google Doc Link:
https://docs.google.com/document/d/19OLgRupq3-crgAOELdSHIXyBhpyM64h0tUNlKTewJq0/edit?usp=sharing