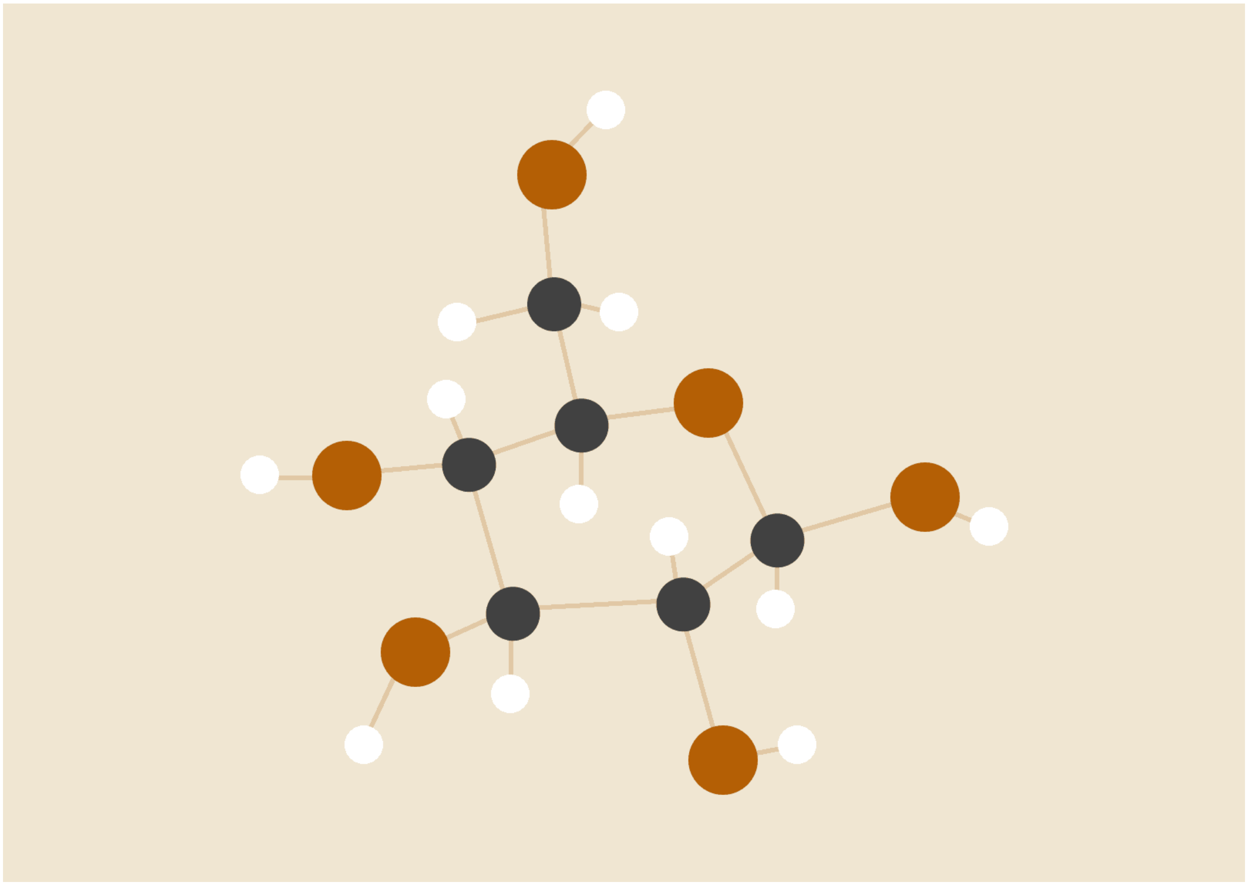
214 PROJECT REPORT

PLANT NURSERY SIMULATOR



Bridge Trolls

RESEARCH BRIEF

A successful plant nursery requires carefully designed structures and workflows to support optimal plant growth and operational efficiency. Core facilities include store-houses for organizing tools, equipment, seeds, and fertilizers, and potting and packing sheds for transplanting, repotting, and preparing plants for sale.

Advanced propagation structures, such as mist chambers, cold frames, and hotbeds, enable precise control of humidity and temperature, improving seed germination and cutting propagation. Greenhouses offer comprehensive protection from pests, extreme weather, and environmental fluctuations, while allowing controlled light and irrigation management. Integrated design and workflow optimization,such as strategically placed water sources, ventilation, and access paths,ensure efficient movement of plants through growth stages and preparation for sale.

Overall, effective nursery management balances environmental control, plant protection, and staff workflow. Centralized storage, flexible propagation structures, and systematic layout contribute to consistent plant quality, efficient operations, and year-round productivity.

Reference:

Agriculture Institute, n.d. *Key structures and components in nurseries*. Available at: https://agriculture.institute/plant-propagation-nursery-mgt/key-structures-components-nurseries/ [Accessed 31 October 2025].

Influence on Design Decisions:

1. **Plant Lifecycle and States:**
   The research emphasized that plants have distinct stages (unplanted, seedling, mature, ready for sale) and require different care at each stage. This inspired the use of the **State Pattern** in the **Plant** class hierarchy to encapsulate stage-specific behavior and transitions. Methods like grow() and endOfDay() reflect environmental interactions and staff actions, simulating realistic growth cycles.

2. **Staff Roles and Workflow:**
   Staff in nurseries perform multiple tasks: watering, fertilizing, inventory management, and assisting customers. This research led to the creation of the **Employee** class as a central mediator between plants and customers, implementing a **Mediator pattern** to manage interactions and enforce workflow logic. This ensures that plant care and customer service actions are coordinated and do not conflict.

3. **Inventory and Plant Management:**
   Proper storage and inventory tracking were identified as critical for nursery operations. The **Inventory** class maintains a collection of all plants, both ready-for-sale and in-progress. The **Iterator pattern** was applied to allow traversal of available plants, while ensuring that plants removed for orders or returned via refunds are correctly tracked. This mirrors real-world inventory management where staff need to know exactly what is available.

4. **Customer Interaction and Transactions:**
   Customers browse plants, make purchases, and request refunds. Using the **Command pattern**, we encapsulate these operations, allowing the system to queue, execute, and reverse transactions while maintaining proper history (TransactionHistory) for each customer. This approach was inspired by the research emphasis on seamless customer interactions and transaction reliability.

5. **Environmental Controls and Assumptions:**
   While nurseries use temperature, humidity, and light management to optimize growth, the simulator abstracts these factors into numeric environment values (**currentEnvironment and preferredEnvironment**) for simplicity. Similarly, watering and care are abstracted as boolean flags (**isWatered**), with staff actions triggering changes.

Definitions and Assumptions:

- **Plant Lifecycle Stage:** Represents the development of a plant from unplanted to ready-for-sale.
- **Environmental Parameter:** Abstract numeric representation of conditions such as humidity, temperature, and light affecting plant growth.
- **Staff Mediation:** Employees act as intermediaries between the greenhouse and

sales floor, coordinating plant care and customer interactions.

- **Inventory Management:** Centralized tracking of plants, ensuring availability and proper state representation, inspired by real-world storage and workspace practices.

DESIGN PATTERN APPLICATION:

1.State Pattern

Reason for Choice:

The State Pattern was chosen to model the different stages in a plant's lifecycle. Each plant transitions through multiple stages:**unplanted**, **seedling**, **mature**, **dead state**, **ready-for-sale and sold state** and exhibits different behaviors at each stage. Using the State Pattern allows encapsulation of stage-specific behavior in separate classes, making the plant's lifecycle easier to manage, extend, and maintain without complex conditional logic in the **Plant** class.

Implementation in the Project:

- **Context Class:Plant** holds a reference to its current **PlantState**.

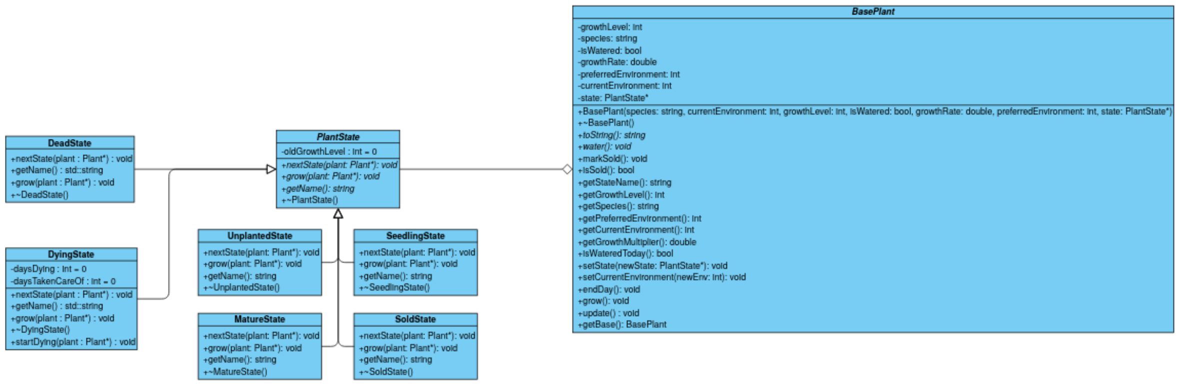
- **State Interface:PlantState** declares the methods that all states must implement, such as **grow()** and **endOfDay()**.

- **Concrete States:** Implement **PlantState** and define stage-specific behavior.

- **State Transitions:** Methods like **grow()** or **endOfDay()**. trigger transitions between states, e.g., a SeedlingState grows into MatureState after sufficient care.

Functional Requirements Addressed:

- Plant lifecycle and growth simulation.
- Plant behavior that changes according to the lifecycle stage.
- Simplified maintenance and future extensibility for new plant stages.


DeadState
+nextState(plant : Plant*) : void
+getName() : std::string
+grow(plant : Plant*) : void
+~DeadState()
DyingState
-daysDying : int = 0
-daysTakenCareOf : int = 0
+nextState(plant : Plant*) : void
+getName() : std::string
+grow(plant : Plant*) : void
+~DyingState()
+startDying(plant : Plant*) : void
PlantState
-oldGrowthLevel : int = 0
+nextState(plant: Plant*): void
+grow(plant: Plant*): void
+getName(): string
+~PlantState()
UnplantedState
+nextState(plant: Plant*): void
+grow(plant: Plant*): void
+getName(): string
+~UnplantedState()
SeedlingState
+nextState(plant: Plant*): void
+grow(plant: Plant*): void
+getName(): string
+~SeedlingState()
MatureState
+nextState(plant: Plant*): void
+grow(plant: Plant*): void
+getName(): string
+~MatureState()
SoldState
+nextState(plant: Plant*): void
+grow(plant: Plant*): void
+getName(): string
+~SoldState()
BasePlant
-growthLevel: int
-species: string
-isWatered: bool
-growthRate: double
-preferredEnvironment: int
-currentEnvironment: int
-state: PlantState*
+BasePlant(species: string, currentEnvironment: int, growthLevel: int, isWatered: bool, growthRate: double, preferredEnvironment: int, state: PlantState*)
+~BasePlant()
+toString(): string
+water(): void
+markSold(): void
+isSold(): bool
+getStateName(): string
+getGrowthLevel(): int
+getSpecies(): string
+getPreferredEnvironment(): int
+getCurrentEnvironment(): int
+getGrowthMultiplier(): double
+isWateredToday(): bool
+setState(newState: PlantState*): void
+setCurrentEnvironment(newEnv: int): void
+endDay(): void
+grow(): void
+update() : void
+getBase(): BasePlant

2.Command Pattern

Reason for Choice:
The Command Pattern was chosen to encapsulate all user-initiated actions (orders, refunds, and inquiries) as discrete objects. This allows the system to queue, execute, and even reverse actions independently of the objects that initiate them. It also provides a clear way to maintain a transaction history, which is essential for tracking customer interactions and supporting refunds.

Implementation in the Project:

- **Command Interface: Command defines a common interface with an execute() method.**
- **Concrete Commands:**
    - **OrderCommand – handles ordering a plant.**
    - **RefundCommand – processes refunds and updates inventory.**
    - **InquiryCommand – provides information about specific plant types.**

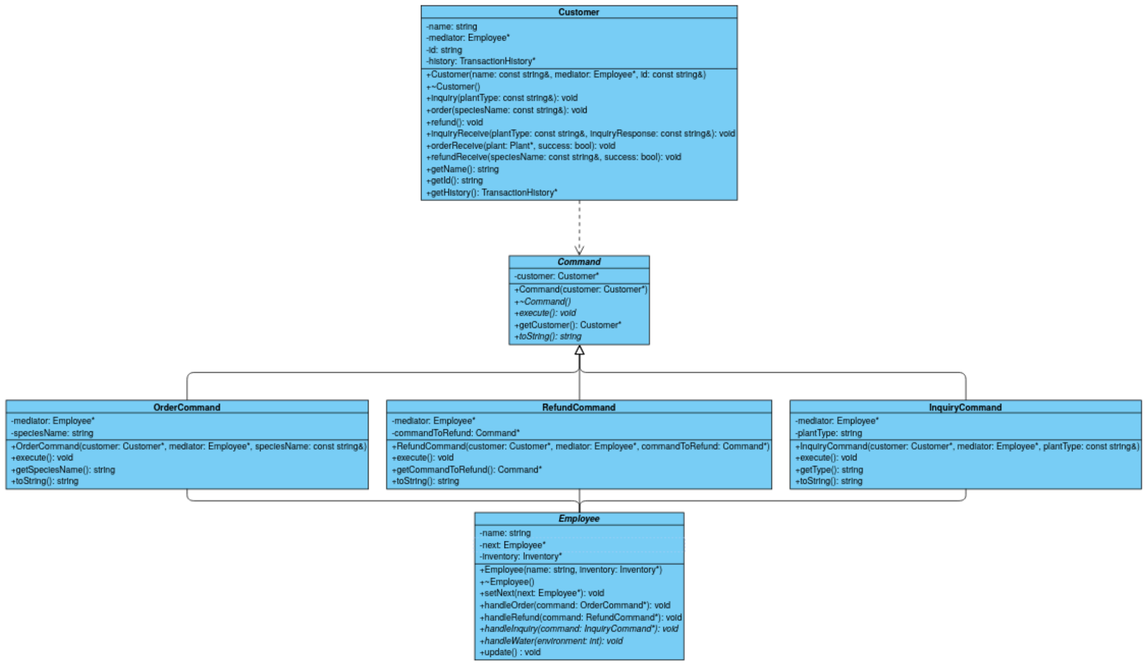
- **Invoker: The Customer class acts as an invoker, creating command objects and executing them.**

- **Receiver: The Employee class and Inventory act on the commands, performing the actual operations such as updating stock or processing refunds.**
- **Transaction History: Each executed OrderCommand is stored in a Transaction History, enabling tracking and potential undoing (refunds).**

Functional Requirements Addressed:

- **Customer browsing and inquiries (InquiryCommand).**

- **Plant purchase and inventory update (OrderCommand).**
- **Refund handling and inventory restoration (RefundCommand).**
- **Transaction tracking and history management.**
- **Decoupling of customer actions from staff processing, enabling flexibility and scalability.**


Customer
-name: string
-mediator: Employee*
-id: string
-history: TransactionHistory*
+Customer(name: const string&, mediator: Employee*, id: const string&)
+~Customer()
+inquiry(plantType: const string&): void
+order(speciesName: const string&): void
+refund(): void
+inquiryReceive(plantType: const string&, inquiryResponse: const string&): void
+orderReceive(plant: Plant*, success: bool): void
+refundReceive(speciesName: const string&, success: bool): void
+getName(): string
+getId(): string
+getHistory(): TransactionHistory*
Command
-customer: Customer*
+Command(customer: Customer*)
+~Command()
+execute(): void
+getCustomer(): Customer*
+toString(): string
OrderCommand
-mediator: Employee*
-speciesName: string
+OrderCommand(customer: Customer*, mediator: Employee*, speciesName: const string&)
+execute(): void
+getSpeciesName(): string
+toString(): string
RefundCommand
-mediator: Employee*
-commandToRefund: Command*
+RefundCommand(customer: Customer*, mediator: Employee*, commandToRefund: Command*)
+execute(): void
+getCommandToRefund(): Command*
+toString(): string
InquiryCommand
-mediator: Employee*
-plantType: string
+InquiryCommand(customer: Customer*, mediator: Employee*, plantType: const string&)
+execute(): void
+getType(): string
+toString(): string
Employee
-name: string
-next: Employee*
-inventory: Inventory*
+Employee(name: string, inventory: Inventory*)
+~Employee()
+setNext(next: Employee*): void
+handleOrder(command: OrderCommand*): void
+handleRefund(command: RefundCommand*): void
+handleInquiry(command: InquiryCommand*): void
+handleWater(environment: int): void
+update() : void

3. Iterator Pattern

Reason for Choice:
 The Iterator Pattern was used to provide a standardized way to traverse collections in the system, such as inventory or transaction history, without exposing the internal

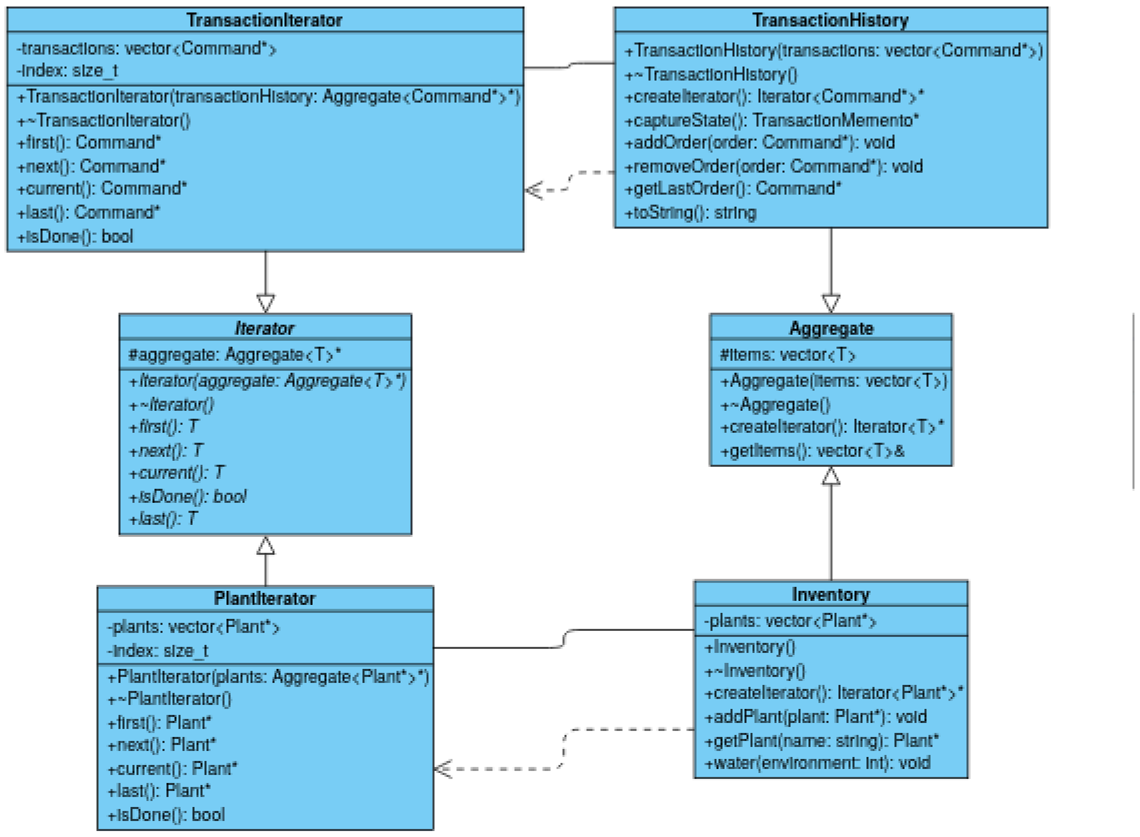
structure of these collections. This allows external classes like **Customer** or **Employee** to view or interact with items (plants or orders) safely and consistently.

Implementation in the Project:

- **Iterator Interface:** Defines traversal methods (**first(),next(),isDone(),current()**).
- **Concrete Iterators:** Implemented for both **PlantIterator** and **PlantIterator**.
- **Concrete Aggregates :Inventory** and **TransactionHistory** classes provide the **createInventory()** method.
- **Clients:** Customers browsing available plants, and staff reviewing transaction history, use these iterators to loop through items.

Functional Requirements Addressed:

- **Customer browsing and plant selection (traversing inventory).**
- **Staff and system management of transaction records.**
- **Inventory and history traversal without exposing internal collection structure.**


TransactionIterator
-transactions: vector<Command*>
-index: size_t
+TransactionIterator(transactionHistory: Aggregate<Command*>*)
+~TransactionIterator()
+first(): Command*
+next(): Command*
+current(): Command*
+last(): Command*
+isDone(): bool
TransactionHistory
+TransactionHistory(transactions: vector<Command*>)
+~TransactionHistory()
+createIterator(): Iterator<Command*>*
+captureState(): TransactionMemento*
+addOrder(order: Command*): void
+removeOrder(order: Command*): void
+getLastOrder(): Command*
+toString(): string
Iterator
#aggregate: Aggregate<T>*
+Iterator(aggregate: Aggregate<T>*)
+~Iterator()
+first(): T
+next(): T
+current(): T
+isDone(): bool
+last(): T
Aggregate
#items: vector<T>
+Aggregate(items: vector<T>)
+~Aggregate()
+createIterator(): Iterator<T>*
+getItems(): vector<T>&
PlantIterator
-plants: vector<Plant*>
-index: size_t
+PlantIterator(plants: Aggregate<Plant*>*)
+~PlantIterator()
+first(): Plant*
+next(): Plant*
+current(): Plant*
+last(): Plant*
+isDone(): bool
Inventory
-plants: vector<Plant*>
+Inventory()
+~Inventory()
+createIterator(): Iterator<Plant*>*
+addPlant(plant: Plant*): void
+getPlant(name: string): Plant*
+water(environment: int): void

4.Memento Pattern

Reason for Choice:
The Memento Pattern was chosen to manage transaction history and provide a rollback mechanism. This allows the system to record each order or refund operation while keeping it separate from the core logic, making it possible to undo actions or restore previous states critical for supporting refunds.

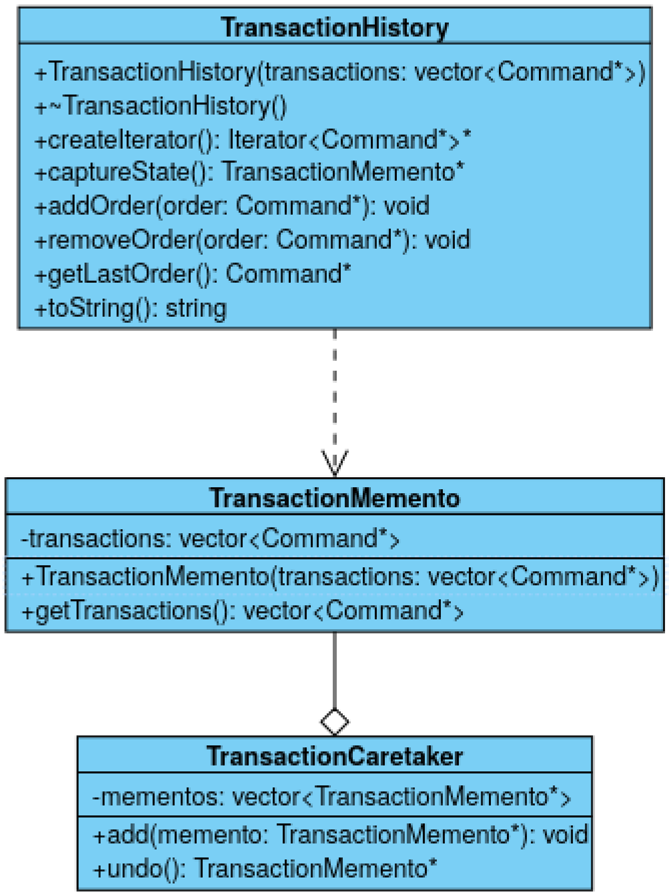
Implementation in the Project:

- **Originator: TransactionHistory** maintains the current state of transactions and creates/restores mementos.
- **Memento: TransactionMemento** stores a snapshot of a transaction.
- **Caretaker: TransactionCaretaker** manages a collection of mementos, providing rollback support.
- **Clients:** Refund commands use mementos to restore previous transaction states

when an order is canceled or refunded.

Functional Requirements Addressed:

- Tracking customer orders and refunds.
- Supporting reversible transactions for refunds.
- Maintaining reliable transaction history while decoupling it from customer actions.


TransactionHistory
+TransactionHistory(transactions: vector<Command*>)
+~TransactionHistory()
+createIterator(): Iterator<Command*>*
+captureState(): TransactionMemento*
+addOrder(order: Command*): void
+removeOrder(order: Command*): void
+getLastOrder(): Command*
+toString(): string
TransactionMemento
-transactions: vector<Command*>
+TransactionMemento(transactions: vector<Command*>)
+getTransactions(): vector<Command*>
TransactionCaretaker
-mementos: vector<TransactionMemento*>
+add(memento: TransactionMemento*): void
+undo(): TransactionMemento*

5. Builder Pattern

Reason for Choice:
The Builder pattern was chosen because the system must construct complex, variant

products (bundles composed of multiple plant items + optional decorations) where the same construction process (sequence of steps) should be able to produce different representations (gift bundle, frost-ready bundle, desk terrarium). This keeps construction logic separate from representation and lets the Director facilitate fixed build sequences while multiple concrete builders provide different internal product representations and decorations.

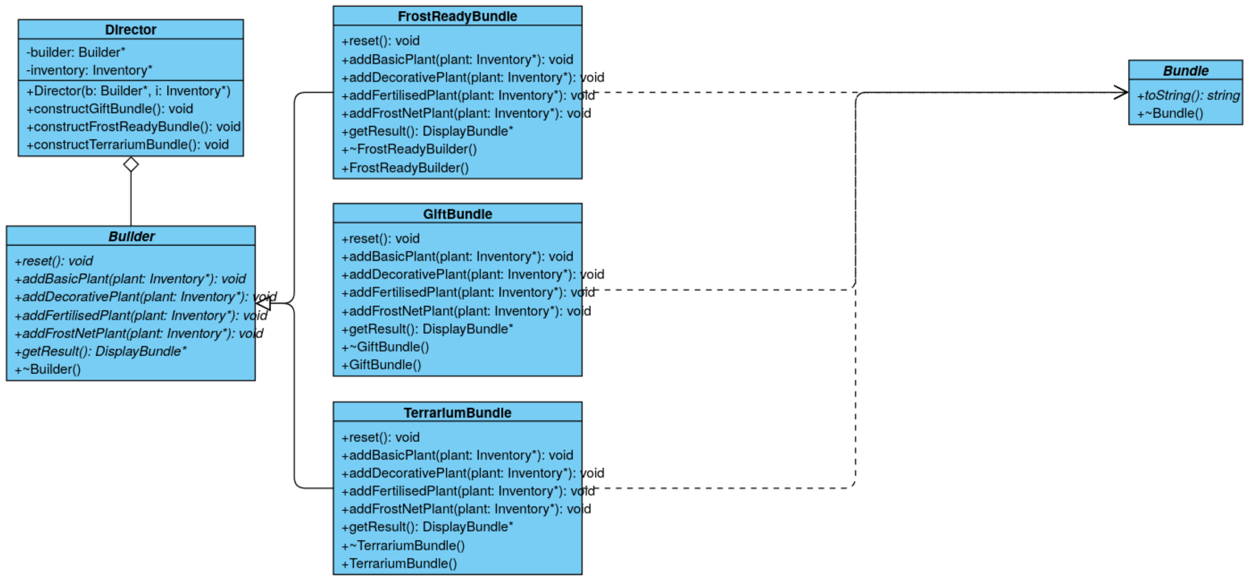
Implementation in the Project:

- **Builder: Builder** defines a full set of build steps for the creation of the product.
- **ConcreteBuilder: GiftBuilder,FrostReadyBuilder and TerrariumBuilder** implement the Builder interface to assemble a particular representation.

- **Director: Director** encapsulates the building algorithm (various construct methods), calls builder steps in proper order, and is builder-agnostic.
- **Product: DisplayBundle** the complex object produced (in our project a composite).

Functional Requirement Addressed:

- **Automation and creation of special arrangements for customers personalisation.**


Director
-builder: Builder*
-inventory: Inventory*
+Director(b: Builder*, i: Inventory*)
+constructGiftBundle(): void
+constructFrostReadyBundle(): void
+constructTerrariumBundle(): void
Builder
+reset(): void
+addBasicPlant(plant: Inventory*): void
+addDecorativePlant(plant: Inventory*): void
+addFertilisedPlant(plant: Inventory*): void
+addFrostNetPlant(plant: Inventory*): void
+getResult(): DisplayBundle*
+~Builder()
FrostReadyBundle
+reset(): void
+addBasicPlant(plant: Inventory*): void
+addDecorativePlant(plant: Inventory*): void
+addFertilisedPlant(plant: Inventory*): void
+addFrostNetPlant(plant: Inventory*): void
+getResult(): DisplayBundle*
+~FrostReadyBuilder()
+FrostReadyBuilder()
GiftBundle
+reset(): void
+addBasicPlant(plant: Inventory*): void
+addDecorativePlant(plant: Inventory*): void
+addFertilisedPlant(plant: Inventory*): void
+addFrostNetPlant(plant: Inventory*): void
+getResult(): DisplayBundle*
+~GiftBundle()
+GiftBundle()
TerrariumBundle
+reset(): void
+addBasicPlant(plant: Inventory*): void
+addDecorativePlant(plant: Inventory*): void
+addFertilisedPlant(plant: Inventory*): void
+addFrostNetPlant(plant: Inventory*): void
+getResult(): DisplayBundle*
+~TerrariumBundle()
+TerrariumBundle()
Bundle
+toString(): string
+~Bundle()

6. Composite Pattern

Reason for Choice:

Bundles themselves are hierarchical objects: a bundle contains plants. The Composite pattern models this elegantly by presenting a uniform interface (Bundle) so clients treat individual Plants and DisplayBundles the same. This supports nesting (terrariums with sub-bundles), uniform display, traversal, and recursive destruction (ownership semantics).

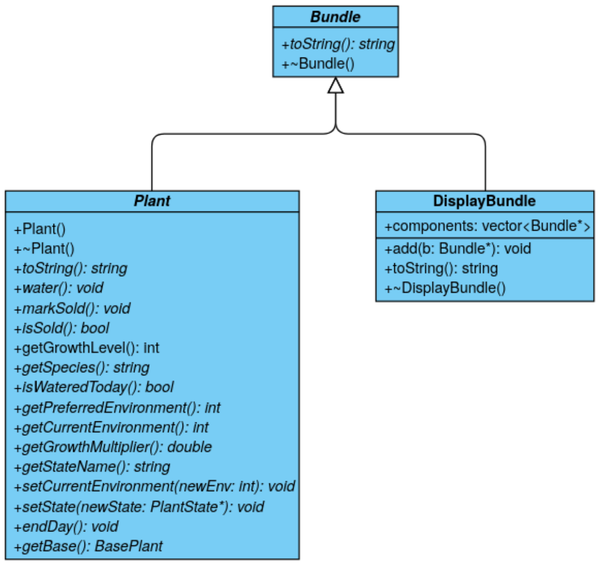
Implementation in the Project:

- **Component:** **Bundle** common interface for both leaves and composites.
- **Leaf:** **Plant** concrete individual objects (no children).
- **Composite:** **DisplayBundle** contains children that are either composites or leaves and iterates children and calls their toString() functions.

Functional Requirement Addressed:

- Supports creation of nested plant bundles.
- Enables uniform treatment of individual plants and composite bundles.

- Simplifies display and traversal of complex bundle structures.



7. Factory Method Pattern

Reason for Choice:
The Factory Method was chosen to create the different subclasses of Plant. It provides an interface that allows the user to specify the attributes that they can or want to and provides default parameters otherwise. The Factory Method was chosen over the other candidate patterns that are also used to create objects because:

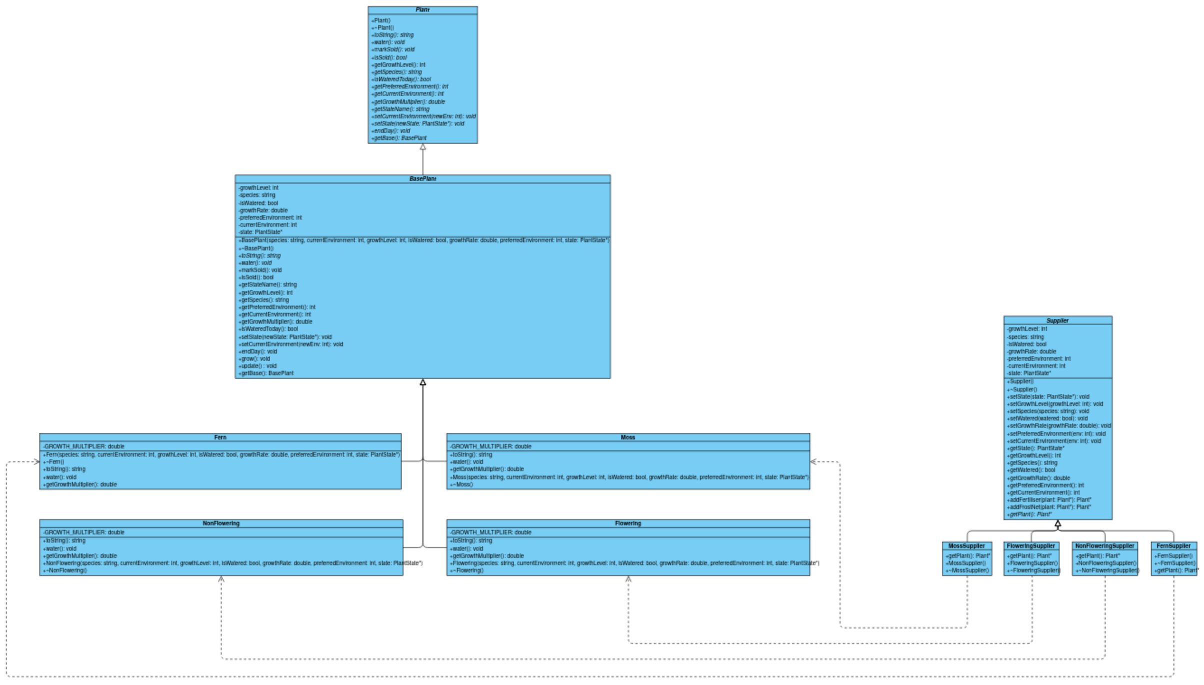
- The Plant does not have a hierarchical structure ruling out the Builder pattern(unless it becomes part of a bundle)
- The Plant is not a member of a family related objects per plant ruling out an Abstract Factory pattern.
- It does not make sense for the Plant to clone itself to create a new one because almost nothing is in common from one plant to the next, ruling out the Prototype pattern.
- The operations to create each plant are the same but the actual concrete plant that is created is different so Template is less well suited than the Factory Method.

Implementation in the Project:

- **Creator:** Supplier
- **Concrete Creators:** MossSupplier, FloweringSupplier, NonFloweringSupplier, FernSupplier
- **Product:** Plant
- **Concrete Products:** Moss, Fern, Flowering, NonFlowering

Functional Requirements Addressed:

- Provides a uniform way to create Plant objects.
- Provides an interface for recording Plants in our system that are purchased from suppliers.



8. Decorator Pattern

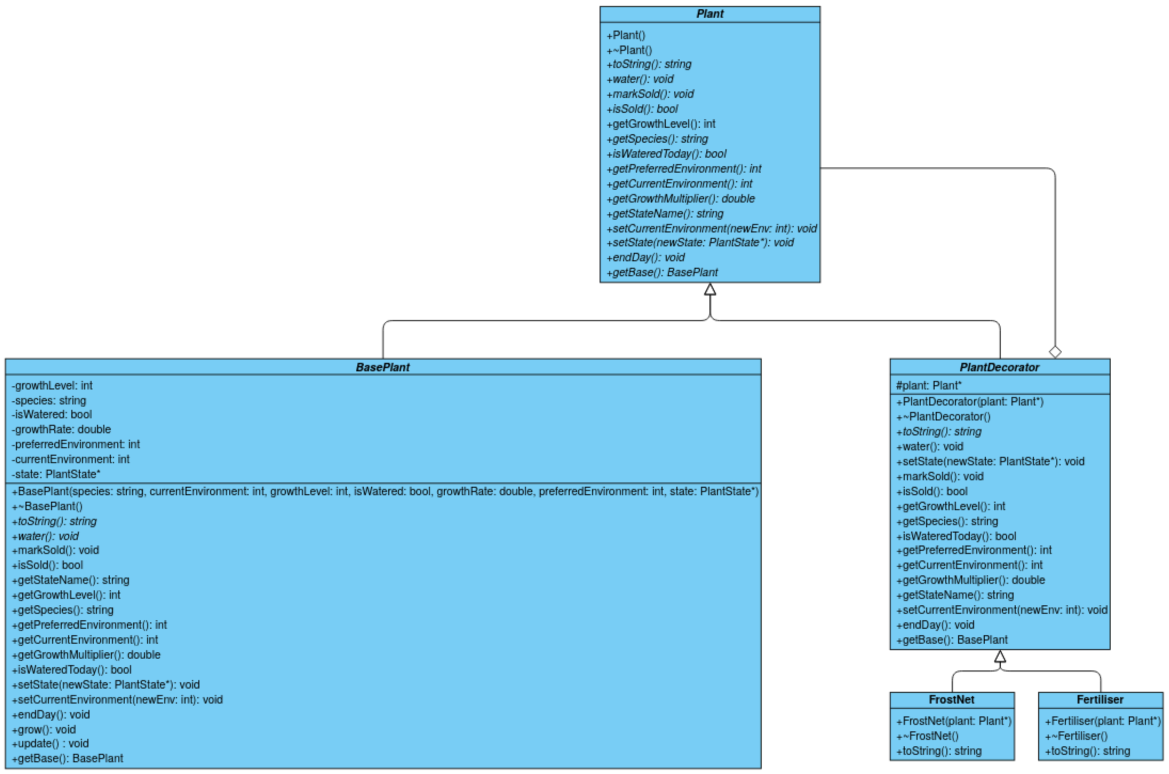
Reason for Choice:
Adds customisation to the representation of the Plant. It allows the plant to exist with no extras or to have cosmetic or functional modifications such as a Bow or Fertiliser that the customer may want to add to their purchase. The Decorator pattern allows multiple customisations to be applied to a plant without adding complexity to Plants that exist in their base state in our gardens before they are ready for purchase.

Implementation in the Project:

- **Component:** Plant
- **Concrete Component:** BasePlant
- **Decorator:** PlantDecorator
- **Concrete Decorators:** Fertiliser, Frostnet

Functional Requirements Addressed:

- Allows users to customize their plants.
- Allows extra customisations to be added easily.



9. Mediator

Reason for Choice:

The mediator was chosen to encapsulate the interactions between the customers and the inventory. The customers should not be able to directly access the inventory or director to order plants or bundles of plants. Instead the responsibility falls on the staff at the nursery being the employee. Our current implementation is similar to the facade, however future implementation where customers may want to send gift plants or bundles to other customers would make the mediator more suitable.

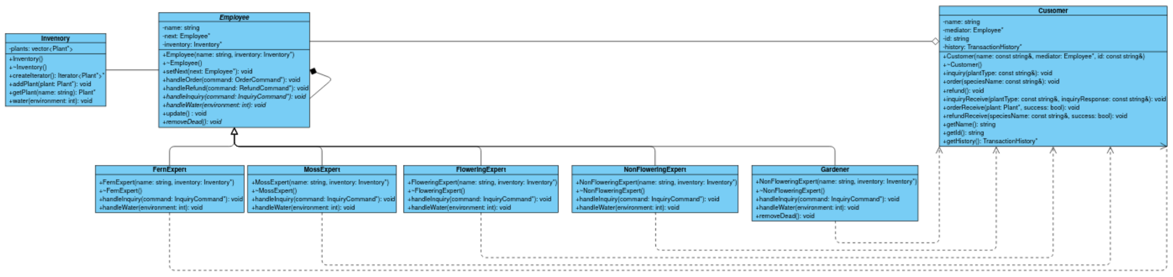
Implementation in the Project:

- Mediator : Employee
- ConcreteMediator : FernExpert, MossExpert, FloweringExpert, NonFloweringExpert, Gardener
- Colleague : Customer, Inventory

Functional Requirements Addressed:

- Prevents a many to many relationship between customers for future implementations

- Provides loose coupling between customers and the inventory as well as the director to create bundles increasing code reusability



10. Chain of Responsibility

Reason for Choice:

The chain of responsibility was chosen because there are employees who specialise in different sub-kingdoms of plants as well as a gardener. The different employee specialists may be unable to help fulfill a specific request and must give the job to another employee who may be better suited to complete it. This is because it would not make sense to ask the **gardener** to answer questions about mosses nor would it make sense for a **Moss Expert** to water the plants.

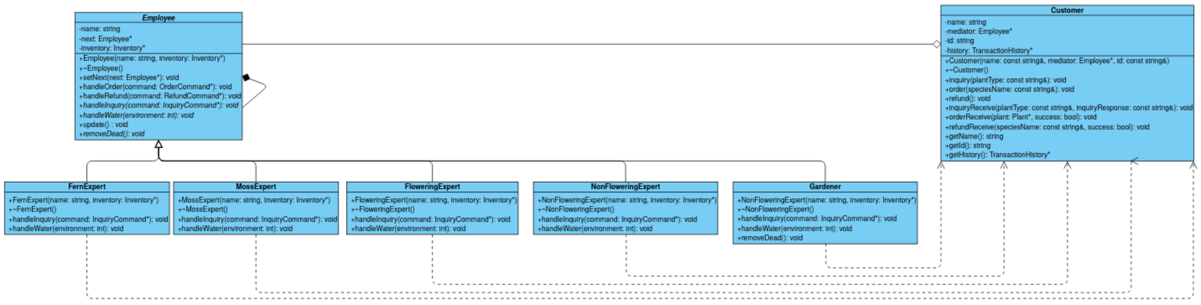
Implementation in the Project:

- Handler : Employee

- Concrete Handler : FernExpert, MossExpert, FloweringExpert, NonFloweringExpert, Gardener
- Client : Command / Customer

Functional Requirements Addressed:

- Increases extensibility by allowing for the addition of more specialist employees
- Reduces coupling by not needing the have the client separately send a request to each specialist


Employee
-name: string
-next: Employee*
-inventory: Inventory*
+Employee(name: string, inventory: Inventory*)
+~Employee()
+setNext(next: Employee*): void
+handleOrder(command: OrderCommand*): void
+handleRefund(command: RefundCommand*): void
+handleInquiry(command: InquiryCommand*): void
+handleWater(environment: int): void
+update() : void
+removeDead(): void
Customer
-name: string
-mediator: Employee*
-id: string
-history: TransactionHistory*
+Customer(name: const string&, mediator: Employee*, id: const string&)
+~Customer()
+inquiry(plantType: const string&): void
+order(speciesName: const string&): void
+refund(): void
+inquiryReceive(plantType: const string&, inquiryResponse: const string&): void
+orderReceive(plant: Plant*, success: bool): void
+refundReceive(speciesName: const string&, success: bool): void
+getName(): string
+getId(): string
+getHistory(): TransactionHistory*
FernExpert
+FernExpert(name: string, inventory: Inventory*)
+~FernExpert()
+handleInquiry(command: InquiryCommand*): void
+handleWater(environment: int): void
MossExpert
+MossExpert(name: string, inventory: Inventory*)
+~MossExpert()
+handleInquiry(command: InquiryCommand*): void
+handleWater(environment: int): void
FloweringExpert
+FloweringExpert(name: string, inventory: Inventory*)
+~FloweringExpert()
+handleInquiry(command: InquiryCommand*): void
+handleWater(environment: int): void
NonFloweringExpert
+NonFloweringExpert(name: string, inventory: Inventory*)
+~NonFloweringExpert()
+handleInquiry(command: InquiryCommand*): void
+handleWater(environment: int): void
Gardener
+NonFloweringExpert(name: string, inventory: Inventory*)
+~NonFloweringExpert()
+handleInquiry(command: InquiryCommand*): void
+handleWater(environment: int): void
+removeDead(): void

11. Observer

Reason for Choice:

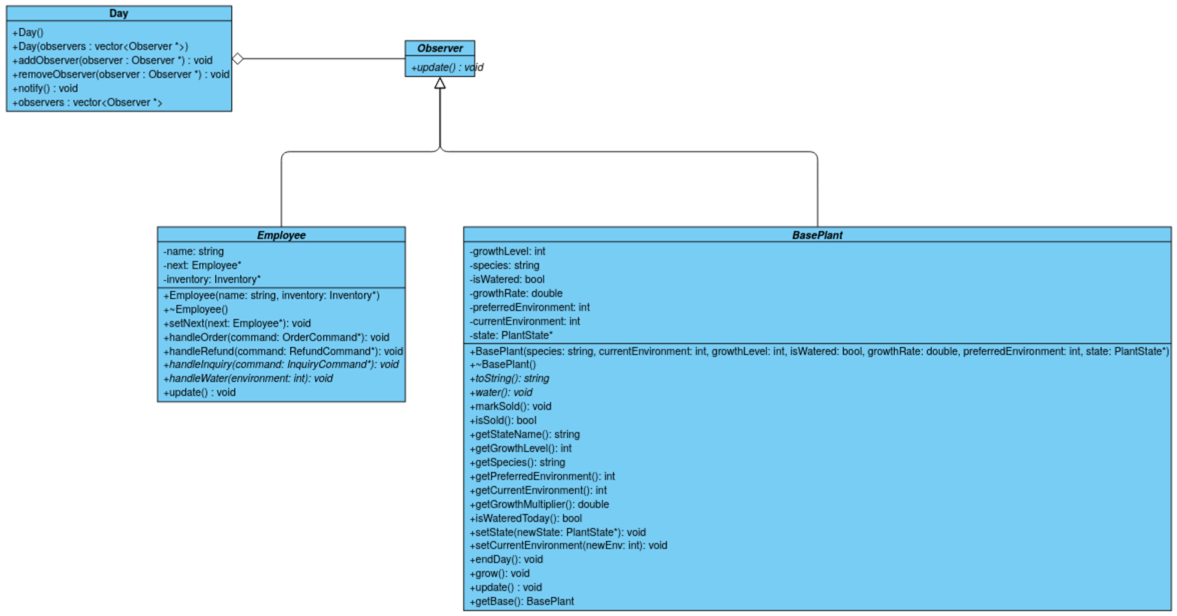
The observer was chosen because both the plants and the employees must update themselves as time passes. The plants may change in growth level and state at the end of each day and the employees need to clock out and go home for the night. Since both of these changes are dependent on the same object, they will observe a shared subject Day that will notify them when a day has passed. The iterator was not used because the employee is also part of the chain of command and each employee maintains a reference to the next employee, meaning having an aggregate over employees would be overly complex.

Implementation in the Project:

- ConcreteSubject : Day
- Observer : Observer
- ConcreteObserver : BasePlant, Employee

Functional Requirements Addressed:

- Increases extensibility because new classes may simply inherit from the Observers class and be notified when a day has passed
- Allows observers to stop observing that a day has passed for some reason (e.g. an employee has been placed on leave or a plant has died)



Day
+Day()
+Day(observers : vector<Observer *>)
+addObserver(observer : Observer *) : void
+removeObserver(observer : Observer *) : void
+notify() : void
+observers : vector<Observer *>
Observer
+update() : void
Employee
-name: string
-next: Employee*
-inventory: Inventory*
+Employee(name: string, inventory: Inventory*)
+~Employee()
+setNext(next: Employee*): void
+handleOrder(command: OrderCommand*): void
+handleRefund(command: RefundCommand*): void
+handleInquiry(command: InquiryCommand*): void
+handleWater(environment: int): void
+update() : void
BasePlant
-growthLevel: int
-species: string
-isWatered: bool
-growthRate: double
-preferredEnvironment: int
-currentEnvironment: int
-state: PlantState*
+BasePlant(species: string, currentEnvironment: int, growthLevel: int, isWatered: bool, growthRate: double, preferredEnvironment: int, state: PlantState*)
+~BasePlant()
+toString(): string
+water(): void
+markSold(): void
+isSold(): bool
+getStateName(): string
+getGrowthLevel(): int
+getSpecies(): string
+getPreferredEnvironment(): int
+getCurrentEnvironment(): int
+getGrowthMultiplier(): double
+isWateredToday(): bool
+setState(newState: PlantState*): void
+setCurrentEnvironment(newEnv: int): void
+endDay(): void
+grow(): void
+update() : void
+getBase(): BasePlant

CONCLUSION

The Plant Nursery Simulator showcases a realistic and efficient system that models nursery operations from plant growth to customer interactions. By leveraging design patterns like State, Command, Builder, and Decorator, we created a flexible, maintainable, and scalable system that handles plant lifecycles, inventory management, transactions, and customizable bundles. The project demonstrates how thoughtful software design can replicate real-world processes, streamline operations, and deliver a satisfying experience for both staff and customers.

Link to Google Docs: Report

Link to Github Repo: https://github.com/u24574547/COS214Prac5.git