COS 214 Project - Plant Nursery Simulator

Report

Abstract Avengers

Taskeen Abdoola – u22698681
Sabira Karie – u23591481
Karishma Boodhoo – u23538199
Rene Reddy – u23558572
Kiolin Gounden – u22647300

Google Doc Link:
https://docs.google.com/document/d/1Sz6qQVimTvQpHv8gAtQXjmmB9GBPVIA0m_3Q79iLQAc/edit?tab=t.0

Project Overview

A plant nursery is a dynamic environment where growth, care, and customer experience come together to create a thriving ecosystem. The nursery consists of several interconnected areas, primarily the greenhouse, where young plants are cultivated, and the sales floor, where customers browse and purchase plants. Each plant species has unique requirements such as watering frequency, sunlight exposure, and seasonal care. If these needs are not met, plant health declines, which directly affects the nursery's ability to operate effectively.

Customers play a critical role in the nursery’s success. They not only purchase plants but also seek advice, guidance, and personalised arrangements. To maintain customer satisfaction, staff members must coordinate between plant care and customer service. Staff assist customers in selecting suitable plants, personalise orders by adding pots, wrapping, and messages, and help ensure that stock on the sales floor reflects what is available in the greenhouse.

Inventory management is another essential aspect. Plants can be ready for sale, still maturing, or temporarily unavailable. When stock runs low, the greenhouse must replenish the sales area. This interaction requires seamless coordination to maintain availability and avoid customer dissatisfaction. The nursery must also be prepared to handle situations such as payment failures, changes in customer decisions, and staff workload balancing.

Sustainability and growth are core goals for any nursery. Over time, the nursery should expand its stock variety, improve operational processes, and enhance the customer experience. Encouraging eco-friendly practices, such as promoting indigenous or low-water plants, supports environmental responsibility while appealing to customers who value sustainability.

Knowing all of this, we designed our system. The greenhouse area nurtures plants through their life cycle using structured care routines. Staff roles are coordinated to manage plant health, customer queries, and stock control. Customers are modelled with behavioural states that reflect real purchasing decisions. Personalisation is supported through custom plant arrangements, allowing customers to create meaningful and unique products. Through the use of object-oriented principles and software design patterns, the nursery adapts and grows over time while maintaining clarity, scalability, and flexibility in its complex, evolving system.

Functional & Non-Functional Requirements

GREENHOUSE/ GARDEN AREA

Functional Requirements:

Design pattern: **State**
FR1: Plant life cycle -
The system shall transition plants through distinct life cycle states (Seedling, Flowering, Mature, Ready for Sale, Sold) based on age, care received, and environmental conditions.

Design pattern: **Composite**
FR2: Stock -
The system must allow users to work with individual plants and plant groups using identical operations (display, get price, count items).

Design pattern: **Factory Method**
FR3: Stock -
The system must allow users to create plants by selecting a category (Succulent, Flower, or Tree) without knowing the specific plant type that will be created.

Design pattern: **Observer**
FR4: Stock -
The system must automatically notify the inventory tracker and staff when plants are added to or removed from stock.

Non-Functional Requirements:

NFR1: Maintainability -
The system shall be designed such that adding a new plant type requires only the creation of a new factory class and plant subclass, with no modifications to existing code.

THE STAFF

Functional Requirements:

Design pattern: **Command**
FR1: Plant care -
The system shall allow the staff to execute plant care routines such as: watering, pruning and fertilizing, as discrete commands can be queued, scheduled and undone.

Design pattern: **Chain of Responsibility**
FR2: Plant life cycle -
The system shall automatically escalate plant health issues through a chain of staff members based on the severity. The members will range from junior gardeners to plant specialists.

Design pattern: **Mediator**
FR3: Inventory -
The system shall coordinate inventory updates between greenhouse and sales floor through a central mediator which ensures that the staff receives timely stock notifications.

Design pattern: **Chain of Responsibility**
FR4: Customer query handling system -
The system shall route customer queries through a help chain where different staff members will handle questions based on their expertise and authority level.

Design pattern: **Factory Method** -
FR5: The system shall create different staff roles such as: gardeners, sales assistants and managers, which uses the factory method to support future role extensions.

Non-Functional Requirements:

NFR1: Maintainability -
The system shall be designed with modular components which allows new plant types, staff roles and care routines to be added with minimal code changes.

THE CUSTOMERS AND THE SALES FLOOR

Functional Requirements:

Design pattern: **Iterator**
FR1: Customer Browsing -
This system provides traversal of items ready for sale and all items without exposing internal collections.

Design pattern: **Strategy**
FR2: Customer Browsing -
This system will provide the Customer with different recommendations/ strategies based on their inputs or not.

Design pattern: **Builder**
FR3: Personalization -
The system shall assemble multi-component arrangements stepwise via a builder (base plant(s), pot, wrap, tag), validating order and mandatory components.

FR4: Personalization -
The system shall provide pre-defined “recipes” (Director) to construct common arrangements (e.g., Gift) with one call.

Design pattern: **Decorator**
FR6: Personalization -
The system shall allow a customer to apply multiple personalisation layers to an item (e.g., DecorativePot, GiftWrap, Note), such that each layer wraps the previous one and the final personalised item is still usable wherever an Item is expected.

Design pattern: **State**
FR7: Sales and Transactions -
The system shall represent each order with a state machine: CartOpen → PendingPayment → (PaymentAuthorized|PaymentFailed) → Completed with additional terminal state Cancelled.

FR8: Sales and Transactions -
Each state shall enforce allowed transitions and reject invalid actions (e.g., CapturePayment is disallowed unless in PaymentAuthorized).

Design pattern: **Command**
FR9: Sales and Transactions -
Sales operations shall be executed as commands (AddToCart, RemoveFromCart, AuthorizePayment, CapturePayment, CommitOrder, CancelOrder) that mutate the order and may emit domain events.

FR10: Sales and Transactions -
If a command in a multi-step transaction fails, the system shall execute configured compensation commands (e.g., CancelOrder after failed CommitOrder).

Design Pattern: **Strategy**
FR11: Pricing Policy -
The system shall compute price via a pluggable PricingStrategy selected at checkout time (e.g., base, promo code, bulk discount).

FR12: Pricing Policy -
The system shall allow runtime switching of the active pricing policy per order (e.g., apply promotional strategy when a valid coupon is added).

Design Pattern: **Adapter**
FR13: Enable Builder & Decorator Compatibility -
The system shall enable the Builder and Decorator patterns to operate on live Plant instances from the sales floor by adapting them to the Item interface.

FR14: Enable Builder & Decorator Compatibility -
The system shall integrate sales and personalization functionality without modifying or extending the abstract Plant class hierarchy.

Non-Functional Requirements:

NFR1: Maintainability -
Concerns regarding different types of recommendations are separated. This makes modifying and adding recommendations easier.

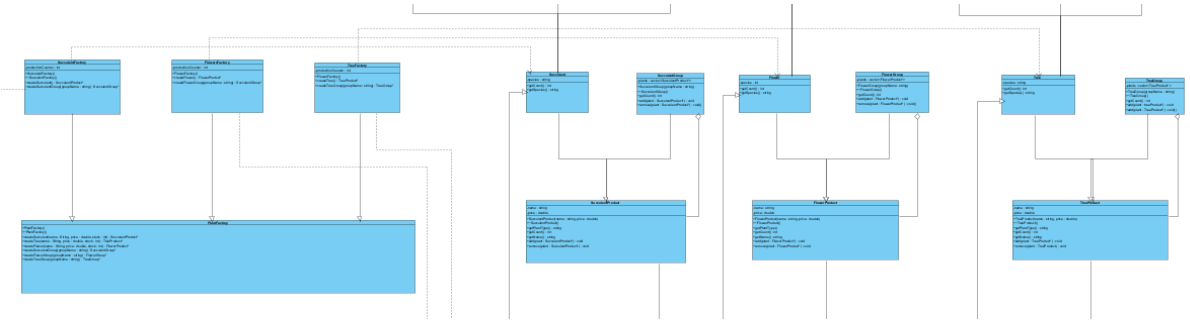
NFR2: Scalability -
The pricing subsystem must support dynamic switching between pricing strategies (e.g., base, bulk, promo) without requiring system downtime or redeployment, maintaining 100% service availability during pricing policy updates.

NFR3: Usability and Flexibility -
The arrangement-builder component must enable customers to construct customised plant arrangements in user interactions (steps) on the interface, and allow for re-ordering or skipping steps without errors, ensuring high usability and flexibility in the personalisation workflow.

Design Pattern Applications

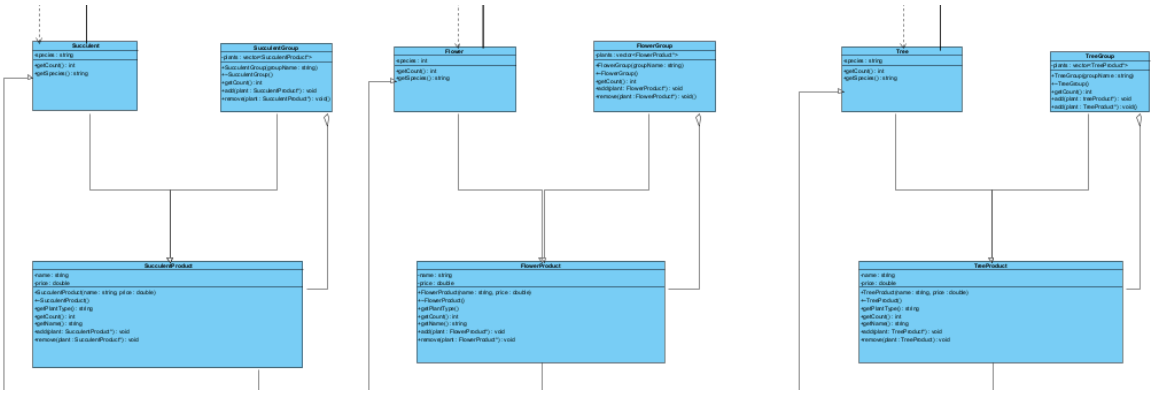
GREENHOUSE/ GARDEN AREA

Factory Method:



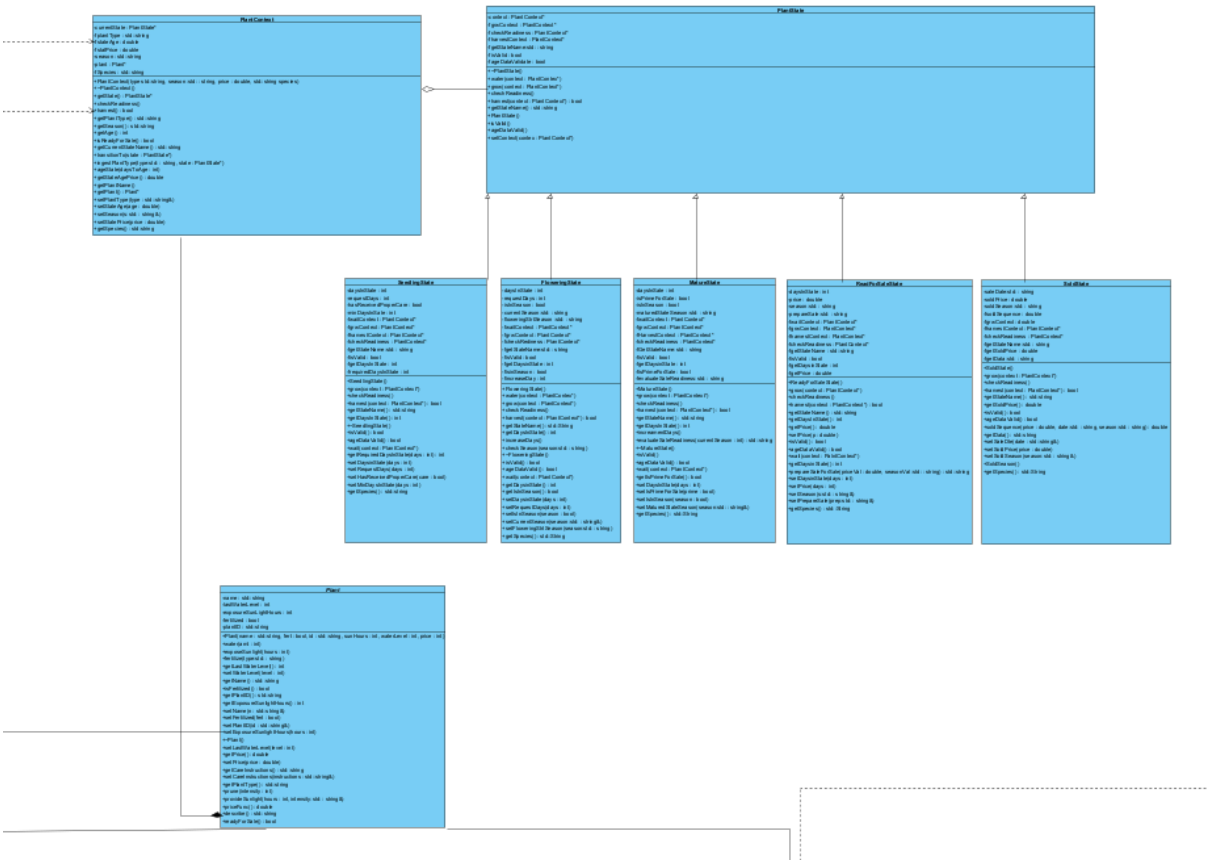
Factory Method was used to create plants by category, like flowers, succulents, and trees. This keeps the creation code clean and makes it easy to add new plant types later. Abstract Factory was not used because sets of related objects don't need to be created, each factory just handles one simple category. Factory Method pattern is a better fit.

Composite:



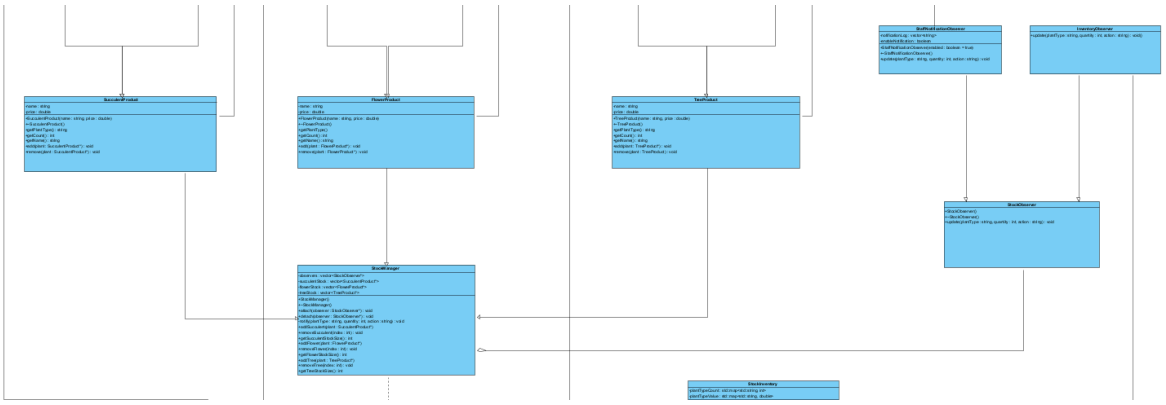
Composite pattern was used to group plants by type so single plants and groups can be treated the same way. For example, a flower group can hold many flowers or other subgroups. Decorator pattern wasn't used because we weren't trying to add extra features to a plant, just needed a way to organize them neatly by category, and we chose Composite over the Iterator pattern because we needed to represent part-whole relationships and manage groups as single entities, not just sequentially access individual plants.

State:



The State Design Pattern was chosen for the plant lifecycle for the greenhouse because each plant progresses through a distinct life cycle—Seedling → Flowering → Mature → Ready for Sale → Sold—and its behaviour changes depending on its current state. The State pattern allows each PlantState class to encapsulate its specific behaviour, such as growth, care, and sale readiness, keeping the logic clean and maintainable. It also enables dynamic transitions between states based on factors like water, season, or fertilizer, while keeping each state focused on its own job and allowing new states or plant types to be added easily. Other patterns like Strategy, Observer, Factory, or Decorator were less suitable because they do not naturally handle sequential, time-dependent state changes with behaviour tied to each state. Overall, the State pattern provides a clear, flexible, and scalable way to manage plant lifecycles in the system.

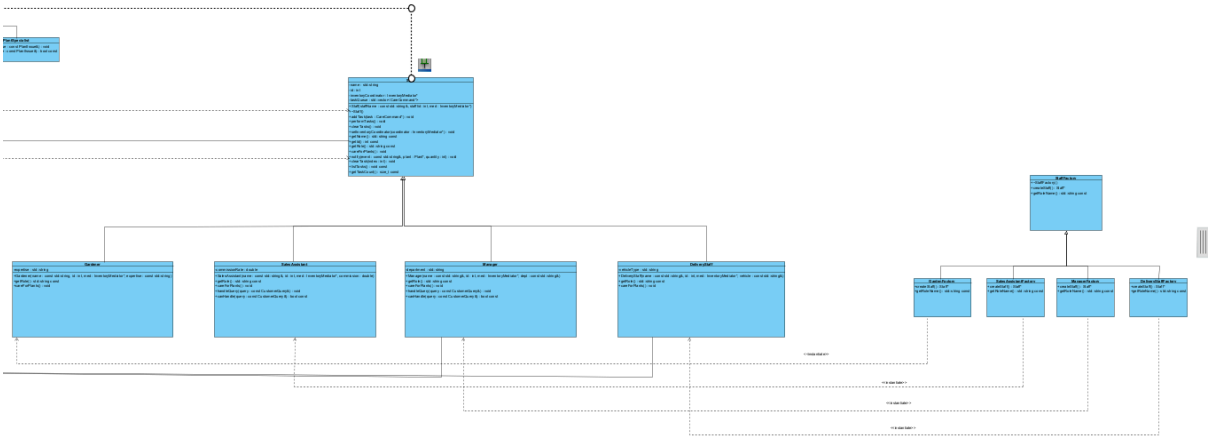
Observer:



The Observer pattern was used so the system could automatically update when stock changes. When plants are added or removed, the StockManager notifies staff and the inventory right away. Mediator was not used because only one object was needed to send updates to many others, not to manage complex two-way communication.

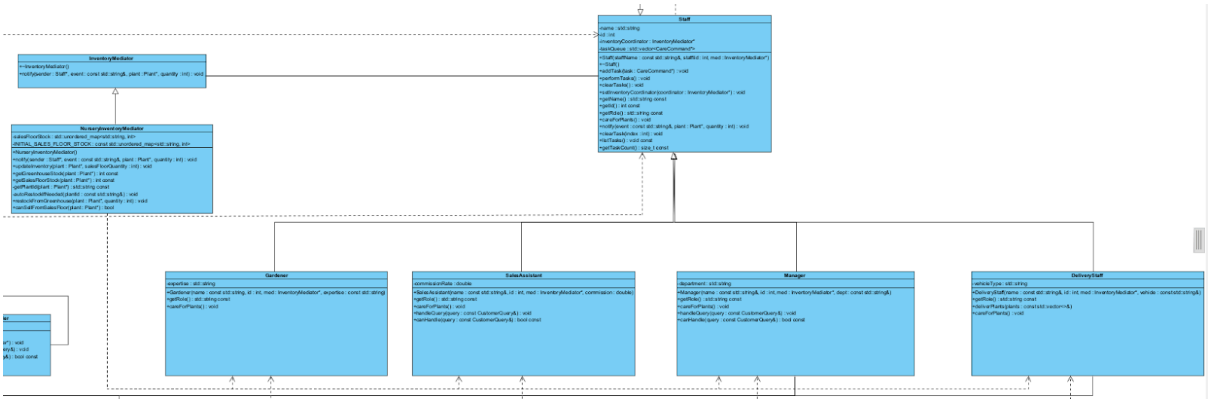
THE STAFF

Factory Method:



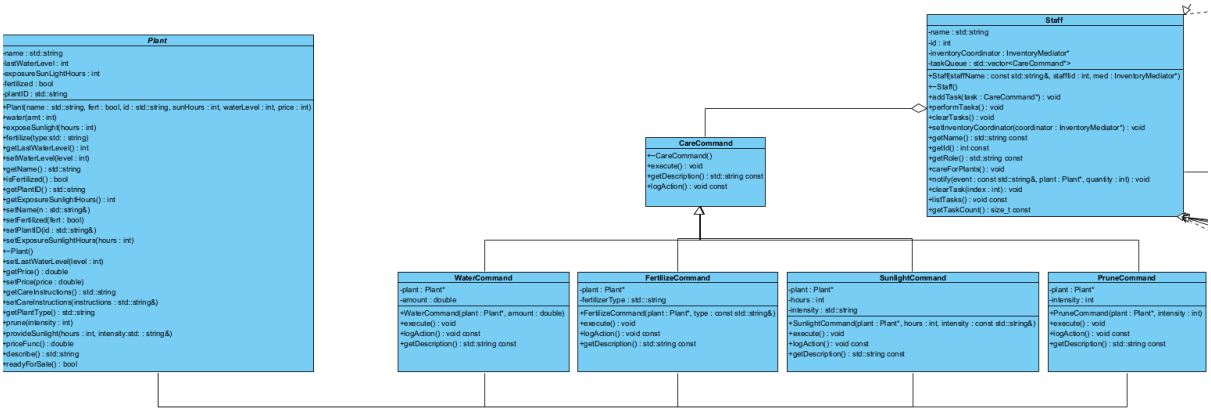
The factory design pattern is well suited for the staff creation for the Nursery because it allows the system to encapsulate the instantiation logic of different staff types, such as greenhouse managers, gardeners and the other staff members, into separate factory classes. This promotes flexibility and extensibility. As the nursery grows and new staff roles need to be created, the factory method enables the system to easily incorporate these without modifying the existing client code. This design pattern also enhances maintainability by centralizing the staff creation logic which makes it easier to manage and update while allowing the subclasses to decide which specific staff type to instantiate based on the nursery's evolving needs. The singleton design pattern was not suitable for this as it restricts instantiation to a single instance which is inappropriate for staff creation since many staff members of the same role are needed. The prototype design pattern would not work as well because while it is useful for cloning objects it is unnecessary here because staff members are not cloned from a prototype but created with specific roles and attributes. The abstract design pattern was also not a good fit because it is designed for creating families of related objects which is not what is required, we simply want to create individual staff members without requiring product families.

Mediator:



The mediator design pattern is well suited for inventory tracking and staff coordination in the nursery because it centralizes complex communication between multiple components such as the greenhouse inventory, sales floor inventory and various staff members into a single mediator object. This prevents these components from having direct references to each other which reduces coupling and makes the system more maintainable and flexible. When inventory changes occur the mediator can automatically notify all relevant staff and update the greenhouse and sales floor record. This ensures consistency across the system without individual objects managing these complex interactions. This approach aligns with the need for coordinated inventory management while allowing easy extension to include new types of staff or inventory categories if needed in the future. The observer design pattern is not suitable for this because it would create dependencies if used for bidirectional coordination between multiple inventory systems and staff members and it does not have the control that the mediator design pattern provides. The command and visitor design pattern is also not suitable as they are not designed for managing real-time communication and synchronization between components.

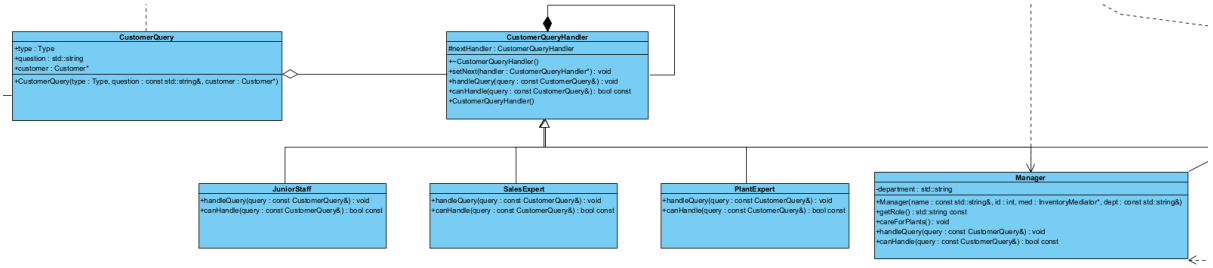
Command:



The command design pattern is well suited for managing plant care routines because it encapsulates each care action such as watering, fertilizing, pruning and providing sunlight which provides a unified interface for executing these operations. This approach allows staff members to invoke plant care actions without needing to understand the specific implementation details of each care type. This promotes loose coupling between the staff and the complex care logic. This design pattern supports the nursery's evolving needs by making it easy to add new care commands without modifying the existing code. This ensures

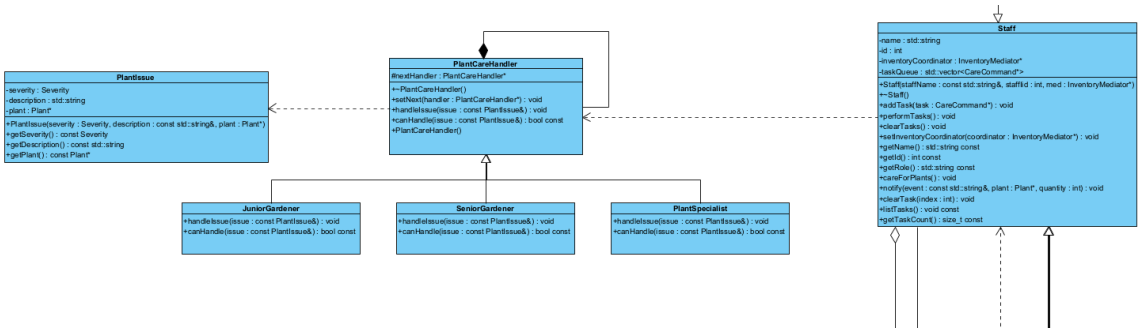
that the system remains flexible and extensible as new plant species with unique care requirements are introduced. The strategy design pattern is not suitable because it focuses on varying a single behaviour rather than encapsulating complete actions that need execution. The mediator and observer design pattern are also not suitable as they do not support the execution control needed for individual care operations. The template method would not be suitable as well because it focuses on algorithm skeletons in base classes but it does not provide the flexibility to treat operations as objects that can be parameterized or queued.

Chain of Responsibility:



The chain of responsibility design pattern is well suited for customer interactions in the nursery because it models the escalation that occurs when staff members help customers with increasingly complex queries. When a customer asks for advice or has questions about plants the requests can be passed through a chain of staff members with different expertise levels. Each handler in the chain can either resolve the request based on their specific knowledge or pass it on to the next more qualified staff member while ensuring that the customers receive the most appropriate assistance without any single staff member needing to know about all possible queries. This approach distributes responsibility and create a flexible system that can easily accommodate new staff roles that can be added to the handler. The command design pattern is not suitable because it does not provide the natural escalation mechanism needed for handling customer queries of varying complexity. The mediator design pattern is also not suitable because it would require the mediator to know about all staff capabilities which would defeat the purpose of its distributed expertise. The observer design pattern is not good as well as it does not provide sequential processing and escalation logic required for customer service interactions.

Chain of Responsibility:

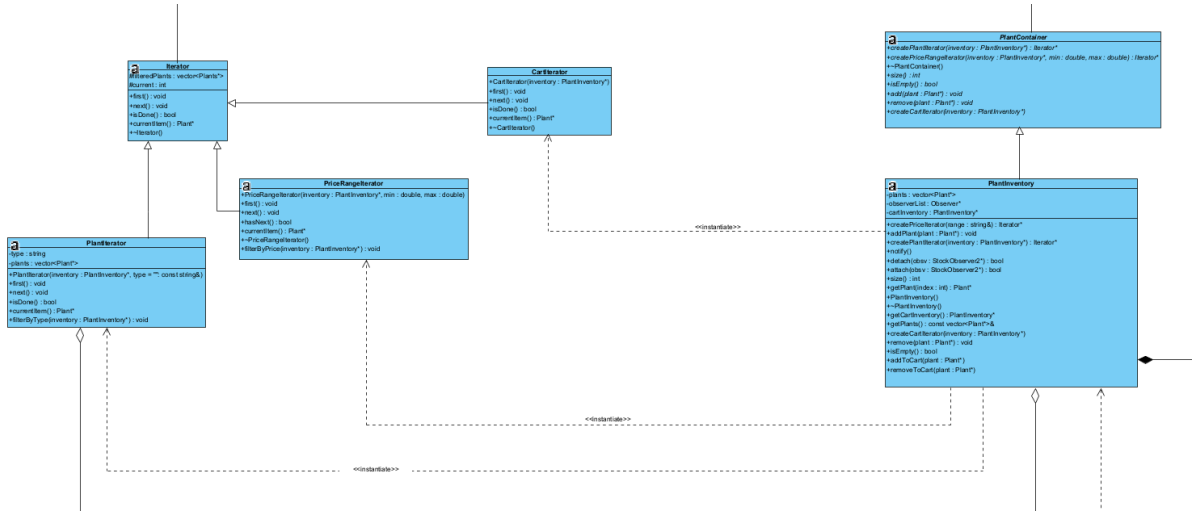


The chain of responsibility design pattern is well suited for monitoring plant health and life cycles because it creates a pipeline of health checks where each handler in the chain can assess specific aspects of a plants condition and either address issues within their expertise

or escalate more serious problems to higher level specialists. This approach ensures comprehensive health monitoring without requiring any single component to understand all possible ailments while allowing flexible additions of new health assessment expertise as the nursery expands its plant varieties. The design pattern effectively distributes responsibility across multiple specialized handlers which ensures the plants receive appropriate attention through a systematic escalation process. The observer design pattern is not suitable as it broadcasts to all observers simultaneously rather than creating the sequential chain needed for systematic health evaluation which is also why the command design pattern is not suitable. The state design pattern is also not suitable because it does not address the need for multiple external handlers to assess and act on plant conditions.

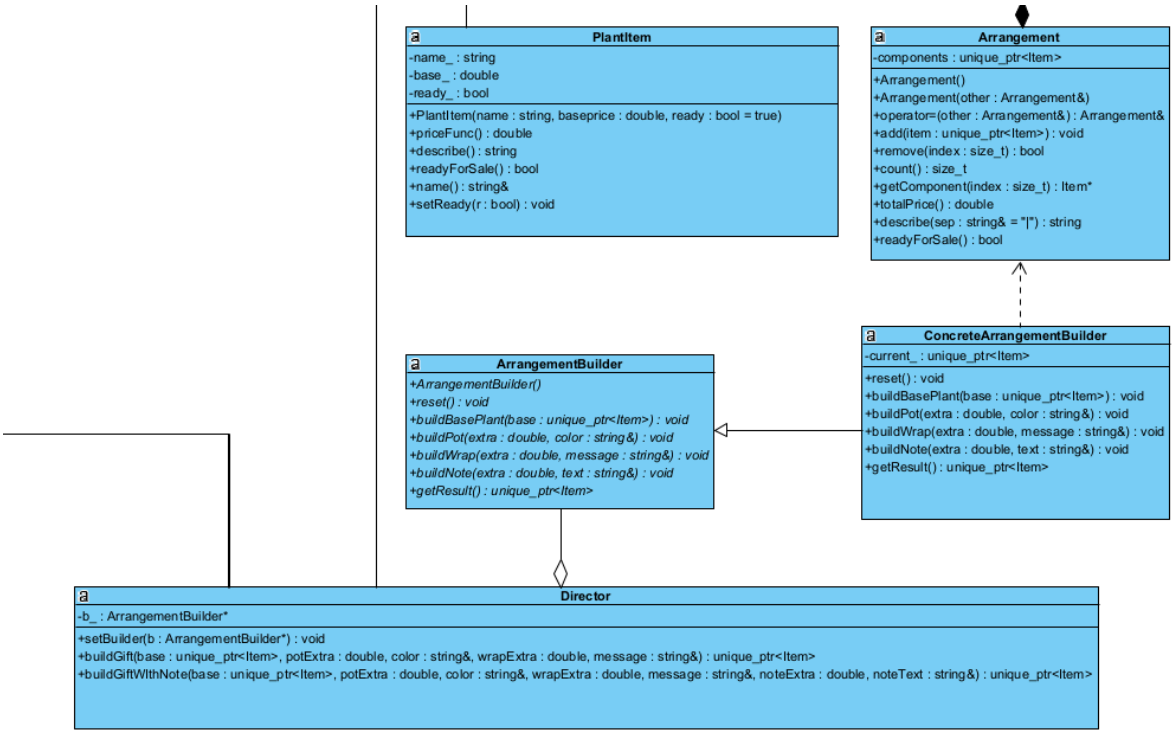
THE CUSTOMERS AND THE SALES FLOOR

Iterator:



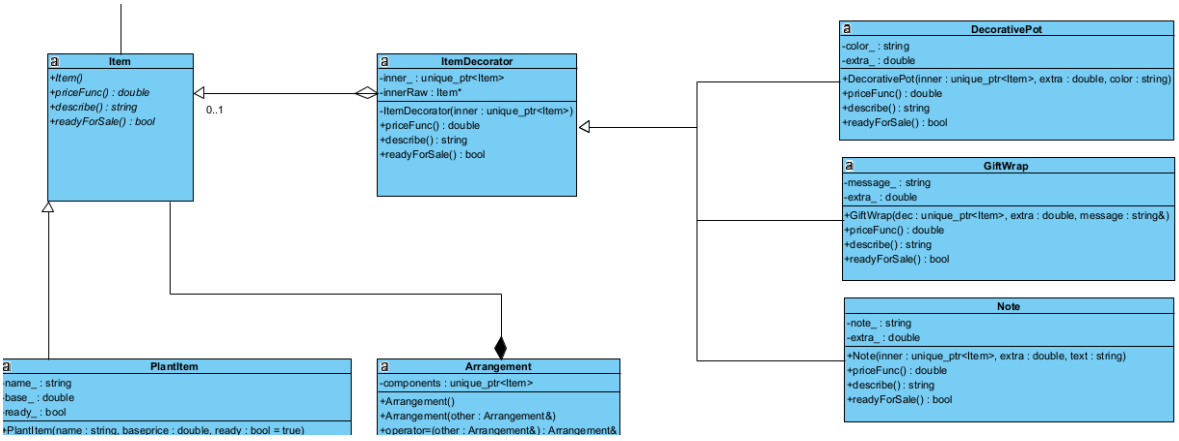
The Iterator design pattern was chosen for customer browsing and has a collection that acts like the sales's floors Inventory. There are three concrete Iterators, PlantIterator - traverses through all the plants in the collection, PriceRangeIterator - iterates through a filtered version of the plants based on the user's desired budget, and lastly the CartIterator - iterates through all the plants that customer/user placed within their carts. This design pattern fits perfectly into the customer browsing scenario as it provides different ways to browse the collection.Why this pattern instead of the Visitor? Using the Visitor pattern would be a total overkill as complex operations are not really required when we're just doing a simple traversal.

Builder:



The Builder pattern was chosen for customer personalization because arrangements are assembled step-by-step, and each step may be optional or vary by order. Builder cleanly separates the construction process from the final Item, enforcing order where needed (e.g., base before decorations) and making it easy to validate required parts. It keeps the construction logic out of UI/state code, supports different build paths, and allows evolving the recipe without breaking clients. Alternatives like Factory Method or Prototype don't capture multi-step assembly rules, and Decorator alone can't manage sequencing/validation of components during creation. Overall, Builder gives a clear, flexible way to construct complex, customizable arrangements.

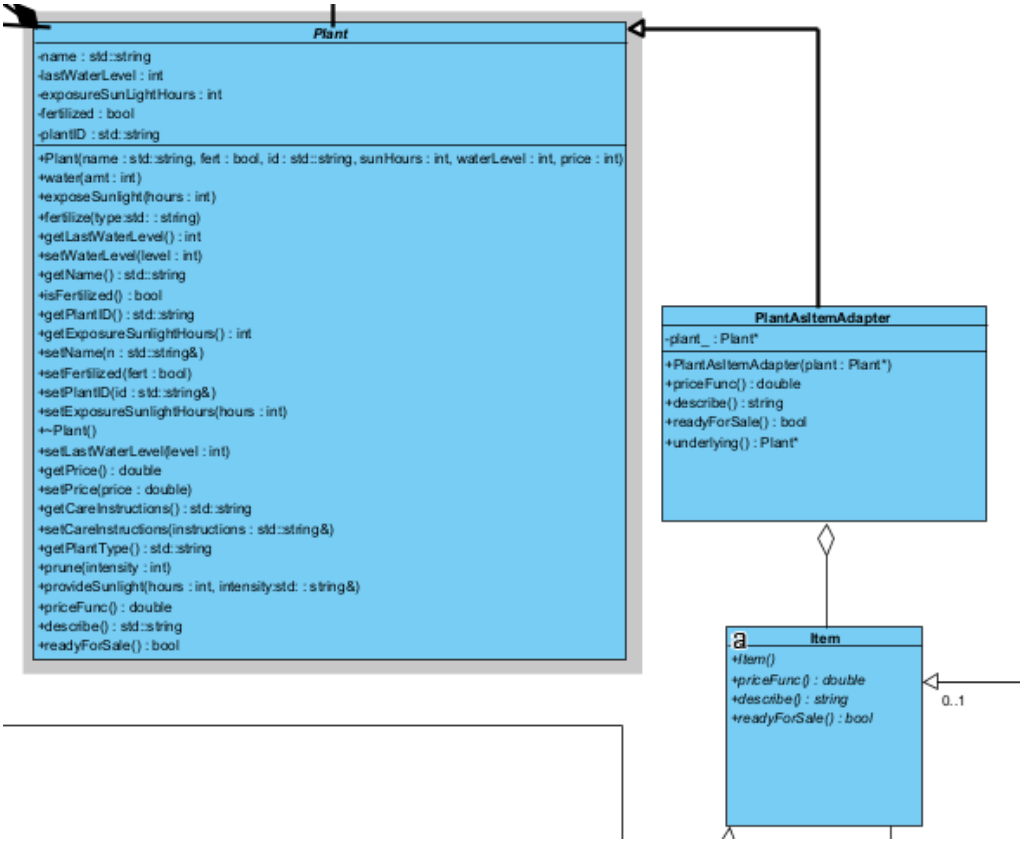
Decorator:



The Decorator pattern was chosen for customer personalization because it allows customers to add multiple optional enhancements—like decorative pots, gift wrapping, and personalized notes—to any base plant or arrangement without modifying its core class. Each decorator "wraps" the item while preserving its original interface, meaning the final decorated

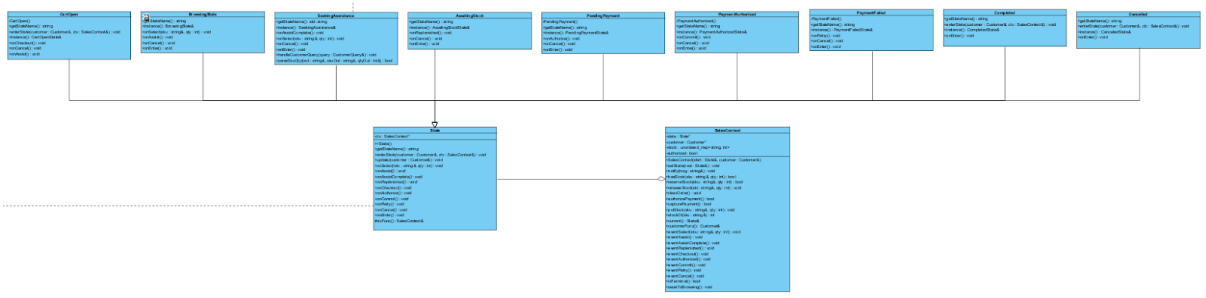
product can still be treated as a single Item by the rest of the system (for pricing, display, or checkout). This pattern provides high flexibility, letting customers mix and match decoration combinations dynamically at runtime. Alternatives like Inheritance or just Builder would either rigidly define all possible combinations or tightly couple construction with appearance. Decorator cleanly supports open-ended personalization, making it ideal for modular add-ons and layered presentation logic on the sales floor.

Adapter:



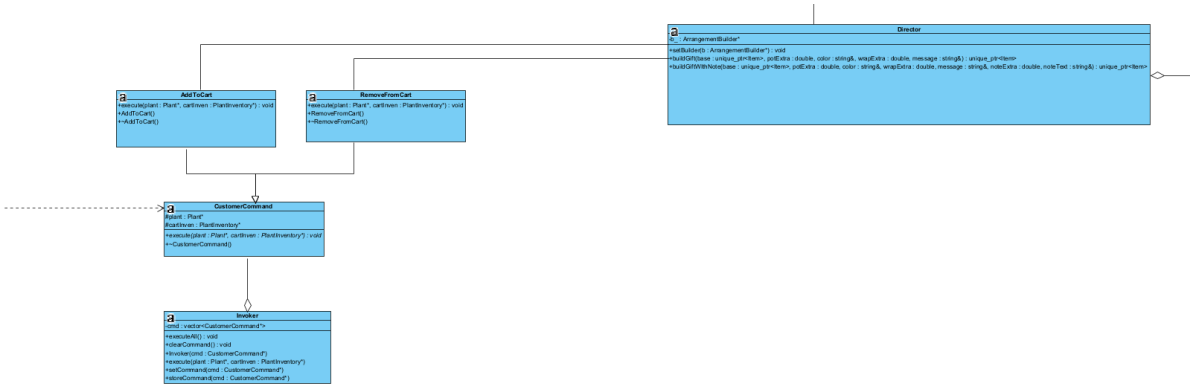
The Adapter pattern was chosen to let the sales/personalization pipeline (which works with the Item interface) operate directly on live, abstract Plant objects without changing Plant. PlantAsItemAdapter wraps a Plant* and exposes Item methods (priceFunc(), describe(), readyForSale()), so Builders and Decorators can treat a plant like any other Item. Because the adapter forwards calls to the underlying Plant, pricing and readiness stay live-linked to inventory (no cloning, no stale copies). This cleanly preserves the abstract Plant hierarchy, and keeps data synchronized at runtime. Alternatives were less suitable: Inheritance/multiple inheritance would force Plant (or all subclasses) to implement Item, violating separation of concerns; Facade doesn't adapt types for polymorphic use; Bridge targets implementation/abstraction decoupling, not interface mismatch; and Strategy alters behavior, not type compatibility. Adapter gives a minimal, non-invasive integration point that keeps domains separate while enabling seamless personalization and checkout flows.

State:



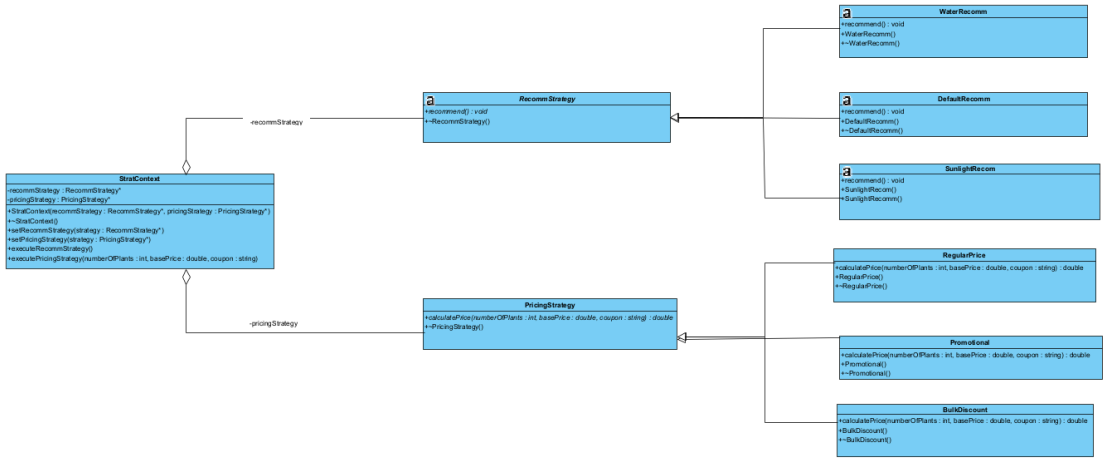
The State pattern was chosen for the order workflow because the cart's permissible actions and behavior change by phase. Each state class encapsulates its own rules (e.g., only PaymentAuthorized may capture payment), producing clear guards and legal transitions while preventing invalid operations. Strategy doesn't model time-ordered transitions; Command executes actions but doesn't gate them by lifecycle; Observer only notifies about changes. State gives a robust, extensible way to enforce business rules across the checkout lifecycle and makes it straightforward to add future states (e.g., Refunded) without tangling logic.

Command:



The Command design pattern was chosen here so it's easier for the user to add and remove from the cart. There are two commands present , AddToCart and RemoveFromCart, CustomerCommand classes that is abstract and an invoker that lets users invoke the command they'd like to use.Why was the command pattern perfect for this scenario? It handles decoupling. It separated the invokes operation from the one that performs it. The undo/redo capability! Command can be stored and reversed.

Strategy:

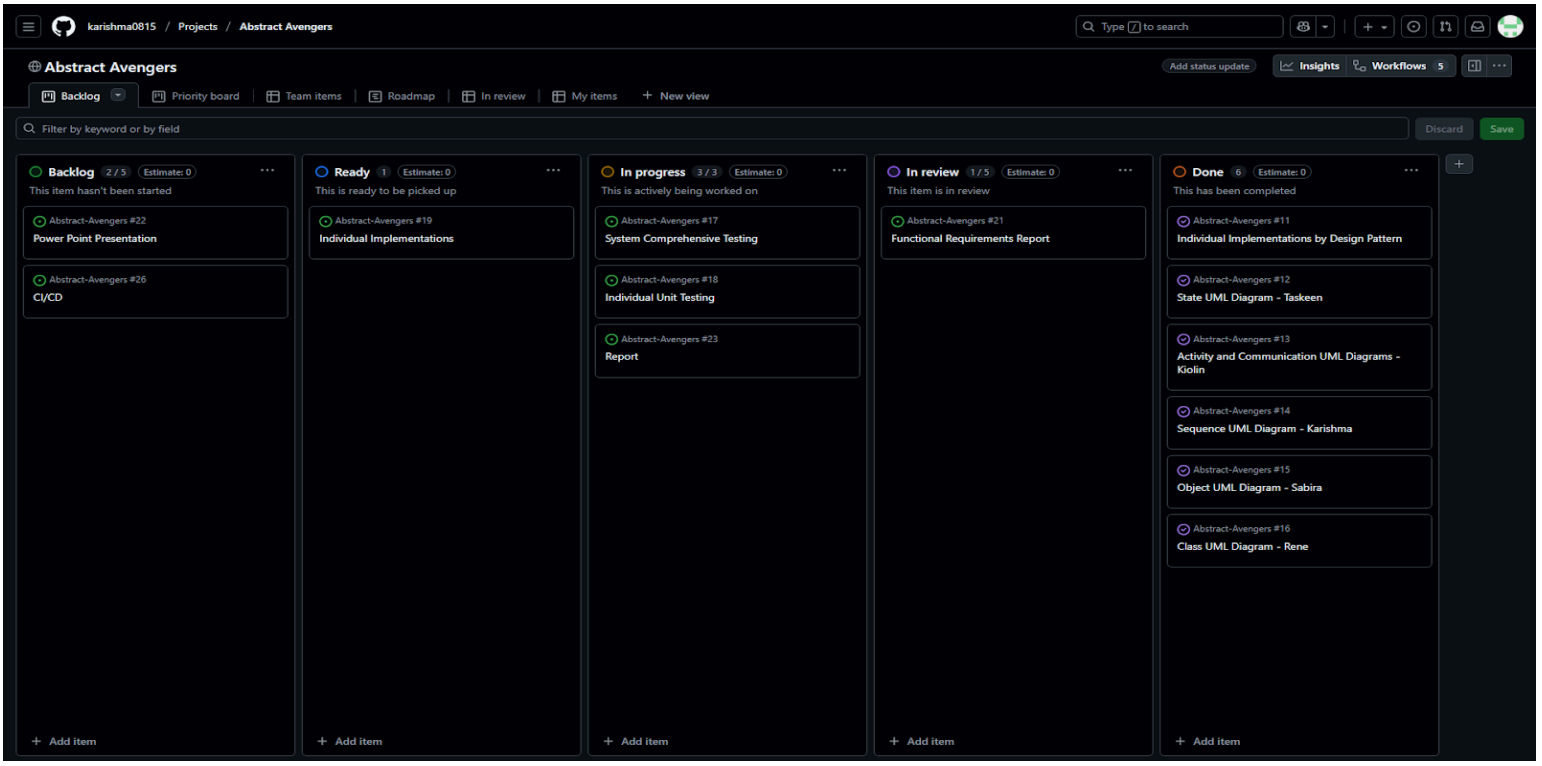


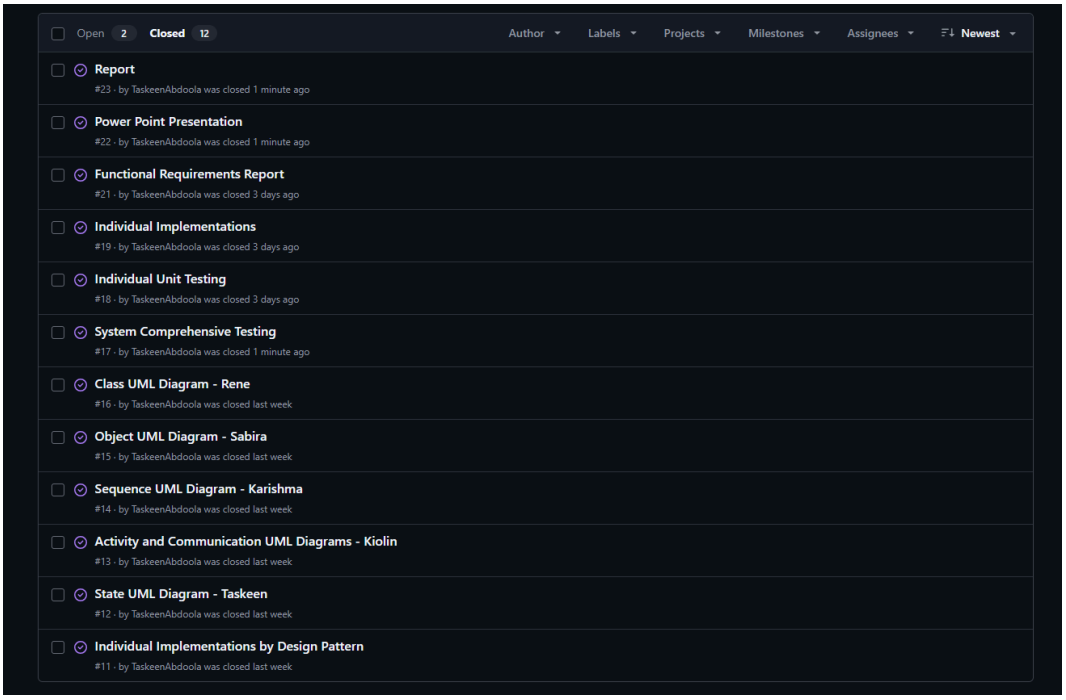
The Strategy design pattern was chosen for both the Recommendations and Pricing. For the Pricing strategy we have three strategies, RegularPrice - provides users/Customers with the price that plant is being sold for, BulkDiscount - gives users 10% for purchasing more than 10 plants, and finally Promotional - this compares the coupon code(so that we know its a valid code) that users have and gives them 90% off on their purchase. For the Recommendation strategy, we have three strategies, DefaultRecomm,WaterRecomm, and SunlightRecomm. These strategies give users recommendations based on what type of plant they'd want to buy.Why is Strategy design pattern the best option in this case? The nursery needs multiple interchangeable algorithms for both the pricing and plant recommendations, allowing dynamic switching based on the customers needs.

Project Management

We used an **Agile Kanban approach** to manage our plant nursery simulation project. Tasks were organized into workflow stages from “Backlog” to “Done,” ensuring continuous progress and clear visibility. Each task was tracked individually (e.g., UML diagrams, testing, reports), reviewed before marking as complete, and assigned to team members based on their roles.

This visual project management method enhanced collaboration, accountability, and productivity, helping us deliver structured outputs efficiently.





Testing Framework

The system was thoroughly tested using the **doctest** C++ testing framework to ensure the correct interaction and functionality of all major components — the **Greenhouse/Garden Area**, **Staff Operations**, and **Customer & Sales Floor** subsystems. Each module was validated through targeted unit and integration tests designed to confirm the intended behavior of implemented design patterns.

Testing verified correct plant lifecycle transitions (Seedling → Flowering → Mature → Ready for Sale → Sold), accurate command execution for staff care actions (watering, pruning, fertilizing), synchronized stock updates between departments via the mediator, and smooth customer interactions across browsing, personalization, and checkout processes.

The State pattern tests confirmed proper transitions in both the Plant Lifecycle and Sales Transactions, including success, failure, cancellation, and retry paths. Builder and Decorator pattern tests validated the flexible creation of multi-layered arrangements, while Factory Method and Observer tests ensured maintainability and automated notifications when new plants or staff were added.

Overall, doctest provided an efficient and lightweight testing framework to validate pattern-specific logic, module interactions, and non-functional aspects such as maintainability, scalability, and usability. The results confirmed that the system behaves consistently across all pattern-driven modules and meets its functional and non-functional requirements.

UNIT TEST: PERSONALIZATION

```
TEST_CASE("Builder + Decorator end-to-end") {
    // base product
    std::unique_ptr<Item> base(new PlantItem("Rose", 45.99, true));

    ConcreteArrangementBuilder builder;
    Director director;
    director.setBuilder(&builder);

    std::unique_ptr<Item> built = director.buildGiftWithNote(
        std::move(base),
        25.0, "Brown",          // pot
        10.0, "Happy Birthday", // wrap
        5.0,  "Enjoy!"          // note
    );

    REQUIRE(built.get() != nullptr);
    CHECK(built->readyForSale());
    CHECK(built->describe().find("Rose") != std::string::npos);
    CHECK(built->describe().find("Brown") != std::string::npos);
    CHECK(built->describe().find("Happy Birthday") != std::string::npos);
    CHECK(built->describe().find("Enjoy!") != std::string::npos);
    CHECK(doctest::Approx(built->priceFunc()) == (45.99 + 25 + 10 + 5));
}
```

```
[doctest] doctest version is "2.4.12"
[doctest] run with "--help" for options
===============================================================================
[doctest] test cases: 1 | 1 passed | 0 failed | 0 skipped
[doctest] assertions: 7 | 7 passed | 0 failed |
[doctest] Status: SUCCESS!
```

UNIT TEST: STATE TRANSITIONS IN GREENHOUSE

```cpp
TEST_CASE("Full lifecycle simulation") {
    PlantContext context("Flower", "Dummy", 12.0, "Sunflower");

    // Seedling -> Flowering
    SeedlingState* seedState = dynamic_cast<SeedlingState*>(context.getState());
    REQUIRE(seedState != nullptr);
    seedState->setMinDaysInState(2);
    seedState->setHasReceivedProperCare(true);
    context.ageState(2);
    CHECK(context.getCurrentStateName() == "Flowering");

    // Flowering -> Mature
    FloweringState* fState = dynamic_cast<FloweringState*>(context.getState());
    REQUIRE(fState != nullptr);
    fState->setIsInSeason(true);
    fState->setRequestDays(3);
    context.ageState(3);
    CHECK(context.getCurrentStateName() == "Mature");

    // Mature -> Ready for Sale
    MatureState* mState = dynamic_cast<MatureState*>(context.getState());
    REQUIRE(mState != nullptr);
    mState->setIsInInSeason(true);
    mState->setIsPrimeForSale(true);
    context.harvest();
    CHECK(context.getCurrentStateName() == "Ready for Sale");

    // Ready for Sale -> Sold
    context.harvest();
    CHECK(context.getCurrentStateName() == "Sold");
}
```

```
root@HPPY:/mnt/c/Users/rener/OneDrive/Documents/2025-Sem2/COS 214/FINAL Project/Final-project-214/SystemFiles# ./plant_tests
[doctest] doctest version is "2.4.12"
[doctest] run with "--help" for options
Unknown plant type: Flower. Defaulting to Protea.
Transitioning from Seedling to Flowering
Unknown plant type: Flower. Defaulting to Protea.
Transitioning from Seedling to Flowering
Transitioning from Flowering to Mature
Unknown plant type: Flower. Defaulting to Protea.
Transitioning from Seedling to Mature
Transitioning from Mature to Ready for Sale
Unknown plant type: Flower. Defaulting to Protea.
Transitioning from Seedling to Ready for Sale
Plant has been sold! Transitioning to Sold state.
Transitioning from Ready for Sale to Sold
Plant sold for $100 on 2025-10-27 during Spring season
Unknown plant type: Flower. Defaulting to Protea.
Transitioning from Seedling to Flowering
Transitioning from Flowering to Mature
Transitioning from Mature to Ready for Sale
Plant has been sold! Transitioning to Sold state.
Transitioning from Ready for Sale to Sold
===============================================================================
[doctest] test cases:  6 |  6 passed | 0 failed | 0 skipped
[doctest] assertions: 13 | 13 passed | 0 failed |
[doctest] Status: SUCCESS!
```

UNIT TEST : ITERATOR DESIGN PATTERN

```
TEST_CASE("Iterator Edge Cases") {
    SUBCASE("Empty inventory for all iterator types") {
        PlantInventory emptyInventory;

        PlantIterator plantIt(&emptyInventory);
        PriceRangeIterator priceIt(&emptyInventory, 0.0, 100.0);
        CartIterator cartIt(&emptyInventory);

        CHECK(plantIt.isDone() == true);
        CHECK(priceIt.isDone() == true);
        CHECK(cartIt.isDone() == true);
    }

    SUBCASE("Single plant inventory") {
        PlantInventory singleInventory;
        Plant* singlePlant = new Jade("Single", 10.0);
        singleInventory.add(singlePlant);

        PlantIterator it(&singleInventory);
```

```
[doctest] doctest version is "2.4.12"
[doctest] run with "--help" for options
===============================================================================
[doctest] test cases:  7 |  7 passed | 0 failed | 0 skipped
[doctest] assertions: 28 | 28 passed | 0 failed |
[doctest] Status: SUCCESS!
```

UNIT TEST : FACTORY METHOD DESIGN PATTERN

```
TEST_CASE("testing the edge cases and error conditions") {
    SUBCASE("Factory creates staff with valid state") {
        GardenerFactory factory;
        Staff* staff = factory.createStaff();

        CHECK(staff != nullptr);
        CHECK_FALSE(staff->getName().empty());
        CHECK(staff->getId() > 0);

        delete staff;
    }

    SUBCASE("testing the staff operations with null mediator") {
        Gardener gardener("Null Mediator Gardener", 1013, nullptr, "General");

        CHECK_NOTHROW(gardener.careForPlants());

        Orchid testPlant("Test Plant", 10.0, "T005");
        CHECK_NOTHROW(gardener.notify("test event", &testPlant, 0));
    }

    SUBCASE("testing the coordinator switching mid operation") {
        NurseryInventoryMediator mediator1;
        NurseryInventoryMediator mediator2;
        DeliveryStaff delivery("Switch Delivery", 4006, &mediator1, "Truck");

        Jade testPlant("Switch Plant", 10.0, "SP001");

        CHECK_NOTHROW(delivery.notify("customer delivery", &testPlant, 10));
        CHECK_NOTHROW(delivery.setInventoryCoordinator(&mediator2));
        CHECK_NOTHROW(delivery.notify("customer delivery", &testPlant, 5));
    }
}
```

```
===============================================================================
[doctest] test cases:  36 |  36 passed | 0 failed | 0 skipped
[doctest] assertions: 297 | 297 passed | 0 failed |
[doctest] Status: SUCCESS!
```