COS 214 Project

Git it done

# Content:

[Content: 2](#_Toc212728556)

[Group Members: 4](#_Toc212728557)

[A Summary of our Research: 5](#_Toc212728558)

[Functional Requirements: 6](#_Toc212728559)

[Prototype 7](#_Toc212728560)

[Explanation: 7](#_Toc212728561)

[Structure: 7](#_Toc212728562)

[Participants: 7](#_Toc212728563)

[Iterator 8](#_Toc212728564)

[Explanation: 8](#_Toc212728565)

[Structure: 8](#_Toc212728566)

[Participants: 8](#_Toc212728567)

[Decorator 9](#_Toc212728568)

[Explanation: 9](#_Toc212728569)

[Structure: 9](#_Toc212728570)

[Participants: 9](#_Toc212728571)

[Factory 10](#_Toc212728572)

[Explanation: 10](#_Toc212728573)

[Structure: 10](#_Toc212728574)

[Participants: 10](#_Toc212728575)

[Observer 11](#_Toc212728576)

[Explanation: 11](#_Toc212728577)

[Structure: 11](#_Toc212728578)

[Participants: 11](#_Toc212728579)

[Command 12](#_Toc212728580)

[Explanation: 12](#_Toc212728581)

[Structure: 12](#_Toc212728582)

[Participants: 12](#_Toc212728583)

[Strategy 13](#_Toc212728584)

[Explanation: 13](#_Toc212728585)

[Structure: 13](#_Toc212728586)

[Participants: 13](#_Toc212728587)

[State 14](#_Toc212728588)

[Explanation: 14](#_Toc212728589)

[Structure: 14](#_Toc212728590)

[Participants: 14](#_Toc212728591)

[Chain of responsibility 15](#_Toc212728592)

[Explanation: 15](#_Toc212728593)

[Structure: 15](#_Toc212728594)

[Participants: 15](#_Toc212728595)

[Builder (Need To Change to Bridge) 16](#_Toc212728596)

[Explanation: 16](#_Toc212728597)

[Structure: 16](#_Toc212728598)

[Participants: 16](#_Toc212728599)

[Non – Functional Requirement 17](#_Toc212728600)

[Security: 17](#_Toc212728601)

[Usability: 17](#_Toc212728602)

[Scalability: 17](#_Toc212728603)

[Performance: 17](#_Toc212728604)

[Maintainability: 17](#_Toc212728605)

[GUI: 17](#_Toc212728606)

# Group Members:

* u23541475-Joshua Heath
* 24590739-Antony Van Straten
* u23540282-Avuyile Sapula
* u24631494-Hamdaan Mirz
* u22503685-Dylan Hebron
* u24898008-Abdelrahman Ahmed
* u24566179-Abdulrahman Sabah

# A Summary of our Research:

* To run a successful nursery there are many factors that had to be researched. A plant is a fragile living entity that needs to be carefully monitored by staff. This includes:
  + watering at the correct intervals,
  + knowing the various life cycles
    - This is needed because you do not want to sell a customer a plant which is not fully developed, also not sell a customer a plant which is dying and needs to be discarded.
* Our first choice for our nursery was to choose which plants we would be growing and selling, these include:
  + Cactus
  + Lavender
  + Rose
* Our second choice was how we would create these plants; we decided on using the Factory Method design pattern, for its simplicity.
* We then had to decide how we would differentiate each plant’s **care** and **life cycles**; we decided on using Strategy and State design patterns to manage these components.
* We also decided on using the Iterator design pattern to be able to traverse our inventory of plants, so we can easily check each plant and all its components.
* To make our system unique and bring in customers, we decided to add a decorator pattern so we could customize our plants. This includes wrapping the plants, adding pots and customizable cards with plants for a personal touch.
* While researching we realized the importance of monitoring the plants, as they are very sensitive to their environment and each need to be maintained in different ways. Our choice of design patterns to solve this problem was a Observer pattern. It allows for a one-to-many relationship between Staff/Main and each plant in the ***Inventory.***
* Bridge pattern …
* Chain of responsibility …
* We also had to make assumptions for our nursery, these include:
  + Our plants have consistent care and life cycles.
  + Staff are the only connection to the plants for customers. In other words, staff are only able to handle the plants and if a customer wants a plant or information about a plant they must receive it from the staff.

# Functional Requirements:

1. Prototype
2. Iterator
3. Decorator
4. Factory
5. Observer
6. Command
7. Strategy
8. State
9. Chain of responsibility
10. Bridge

# Prototype

## Explanation:

* This will be utilized when we want to replicate an existing plant exactly. Instead of merely constructing an existing one, which takes time for individuals in general, this will help the system simplify things.

## Structure:

A screenshot of a computer

AI-generated content may be incorrect.

## Participants:

|  |  |
| --- | --- |
| **Participant** | **Class name** |
| Prototype | Plant |
| ConcretePrototype | Rose |
| ConcretePrototype | Cactus |
| ConcretePrototype | Lavender |
| ConcretePrototype | PlantDecorations |

# Iterator

## Explanation:

* For the plant iterator to traverse through the entire inventory and verify every plant that was added by the staff, I will only be able to add a single plant to the iterator.
* Therefore, to properly iterate through the Inventory class, we will include functions like ***first()***, ***next()***, ***isDone()***, and ***current()*** in our methods.

## Structure:

A screenshot of a computer

AI-generated content may be incorrect.

## Participants:

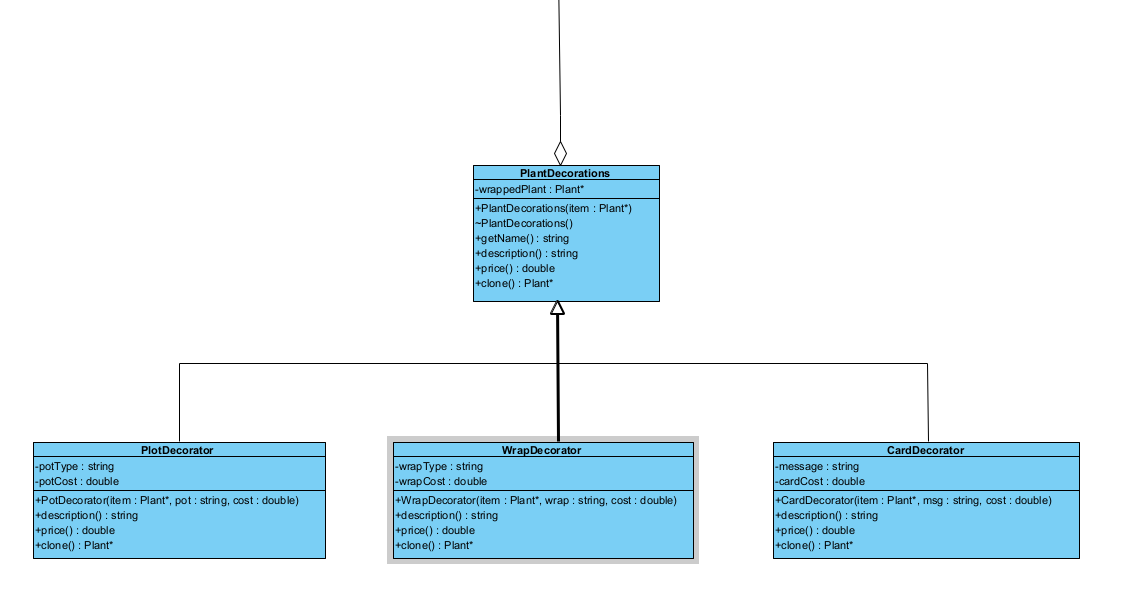
|  |  |
| --- | --- |
| **Participant** | **Class name** |
| Iterator | PlantIterator |
| ConcreteIterator | InventoryIterator |
| Aggregate | InventoryAggregate |
| ConcreteAggregate | Inventory |

# Decorator

## Explanation:

* Thus, when a consumer wants to purchase something, we will provide them with additional features like a plot decorator, a wrap decorator, and a card decorator. They will be able to select the kind of decoration they like to add when they purchase the plant.

## Structure:



## Participants:

|  |  |
| --- | --- |
| **Participant** | **Class name** |
| Component | Plant |
| ConcreteComponent | Cactus |
| ConcreteComponent | Lavender |
| ConcreteComponent | Rose |
| Decorator | PlantDecorations |
| ConcreteDecoratorA | PotDecorator |
| ConcreteDecoratorA | WrapDecorator |
| ConcreteDecoratorA | CardDecorator |

# Factory

## Explanation:

* The primary duty of the plant factory is to produce various plant varieties for our project.
* So, it follows a simple factory Design pattern, where a single factory class is used to produce multiple plant objects depending on the requested type.

## Structure:

A blue rectangular object with black lines

AI-generated content may be incorrect.

## Participants:

|  |  |
| --- | --- |
| **Participant** | **Class name** |
| Creator | Creator |
| ConcreteCreator | CactusCreator |
| ConcreteCreator | LavenderCreator |
| ConcreteCreator | RoseCreator |
| Product | Plant |
| ConcreteProduct | Cactus |
| ConcreteProduct | Lavender |
| ConcreteProduct | Rose |

# Observer

## Explanation:

* An observer pattern is implemented in systems where one-to-many dependencies are prevalent. In Git It Done nursery this relationship is between the class ***Plant*** (Subject) and the class ***PlantObserver*** (Observer). This is done so when the plant changes its state, such as from ***SeedlingState*** to ***GrowingState,*** the observer classes mentioned above will be notified by a command such as ***GrowingStateCmd.***
* This matches real life nurseries where plants are monitored by real time sensors, and when the plants’ environment changes, staff are notified to act.

## Structure:

## Participants:

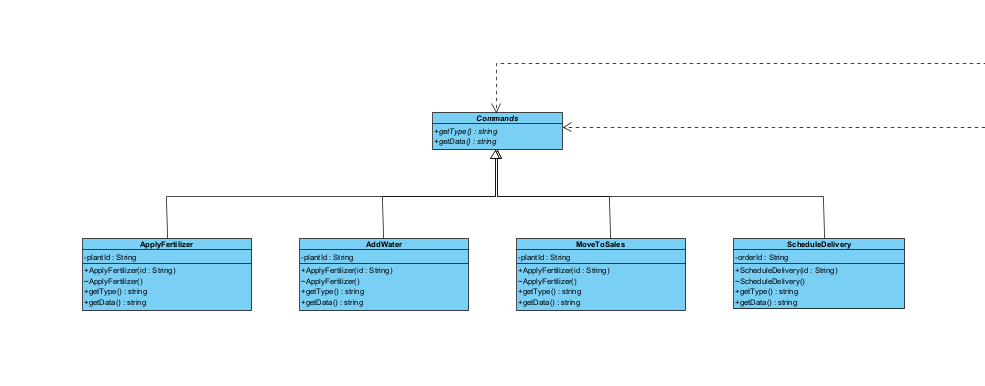
|  |  |
| --- | --- |
| **Participant** | **Class name** |
| Subject | Plant |
| ConcreteSubject | Cactus |
| ConcreteSubject | Lavender |
| ConcreteSubject | Rose |
| Observer | PlantObserver |
| ConcreteObserver | CareScheduleObserver |

# Command

## Explanation:

Our commands, which are distinct objects, will encapsulate actions or requests.   
The following courses, ***ApplyFertilizer,*** will be utilized when a plant requires fertilizer. When water is needed for the plants, we also have ***AddWater***. Additionally, there will be a command called ***MoveToSales*** that may be used to move a plant to sales so that customers can buy it. Additionally, we will have ***ScheduleDelivery*** so that the employees are aware that the customer needs this plant supplied.

## Structure:



## Participants:

|  |  |
| --- | --- |
| **Participant** | **Class name** |
| Command | Commands |
| ConcreteCommand | ApplyFertilizer |
| ConcreteCommand | AddWater |
| ConcreteCommand | MoveToSales |
| ConcreteCommand | ScheduleDelivery |
| Client | Main |
| Invoker | PlantObserver |
| Receiver | Plant |

# Strategy

## Explanation:

This class shows how different plants have different ways to be cared for

The project uses the Strategy Pattern, where each type of plant has its own "care plan" or strategy, rather than writing all the care logic inside a single large class.   
Thus, we'll have the following We will have RoseCare, a specific care type for roses, CactusCare, a specific care type for cacti, and SucculentCare, a specific care type for succulents.

## Structure:

A screenshot of a computer

AI-generated content may be incorrect.

## Participants:

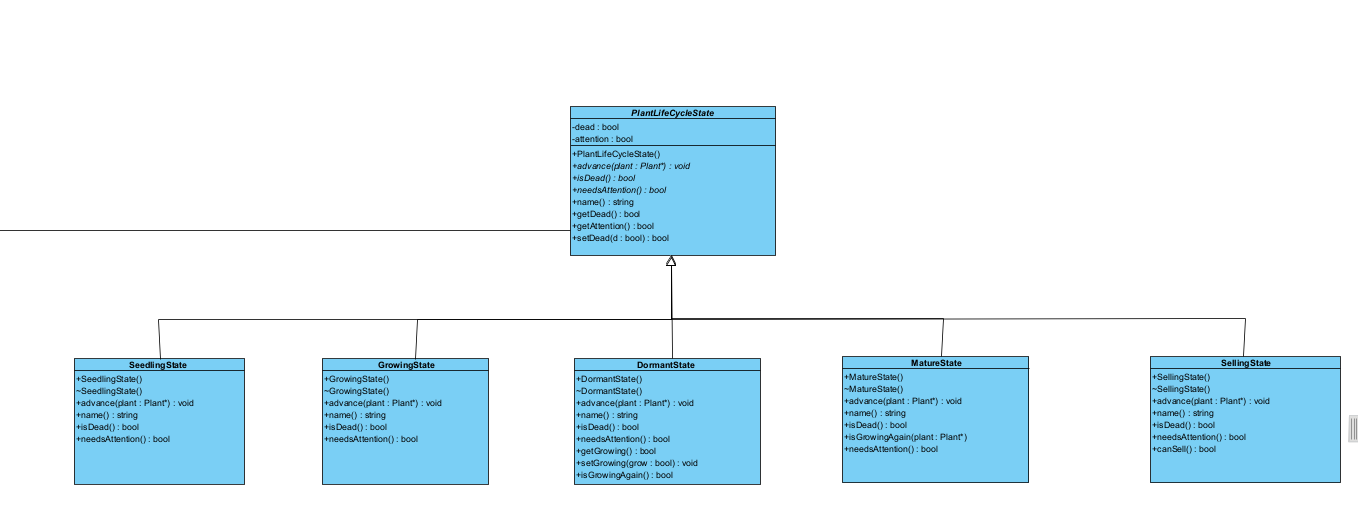
|  |  |
| --- | --- |
| **Participant** | **Class name** |
| Strategy | CareStrategy |
| ConcreteStrategy | CactusCare |
| ConcreteStrategy | LavenderCare |
| ConcreteStrategy | RoseCare |
| Context | Plant |

# State

## Explanation:

* Therefore, when an object's behaviour modifies its present state, the state pattern will be applied.
* For our project, the plant will go through multiple phases, like a life cycle:
  + Seedling → Growing → Dormant → Mature → Selling

## Structure:



## Participants:

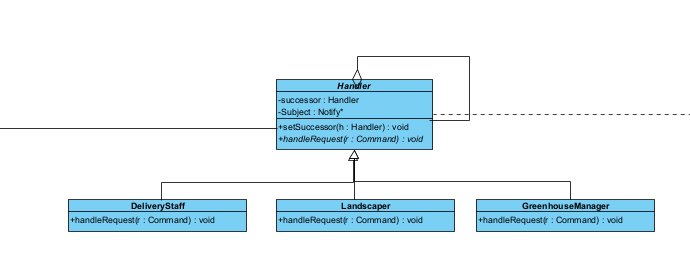
|  |  |
| --- | --- |
| **Participant** | **Class name** |
| State | PlantLifeCycleState |
| ConcreteState | SeedlingState |
| ConcreteState | GrowingState |
| ConcreteState | DormantState |
| ConcreteState | MatureState |
| ConcreteState | SellingState |
| Context | Plant |

# Chain of responsibility

## Explanation:

As a result, the requests, which are commands, will be able to move up the chain of command until one of the following classes (***DeliveryStaff***, ***Landscaper***, or ***GreenhouseManager***) can handle them. All handlers will use the handler class as their base class.

## Structure:



## Participants:

|  |  |
| --- | --- |
| **Participant** | **Class name** |
| Handler | Handler |
| Concrete Handler | DeliveryStaff |
| Concrete Handler | Landscaper |
| Concrete Handler | GreenhouseManager |
| Client | Main |

# Builder (Need To Change to Bridge)

## Explanation:

So, the builder will be used to construct complex plant object step by step- for example, building a Rose or a Succulent with specific care strategies and life cycle states.

## Structure:

A screenshot of a computer flowchart

AI-generated content may be incorrect.

## Participants:

|  |  |
| --- | --- |
| **Participant** | **Class name** |
| Builder | PlantBuilder |
| Concrete Builder | CactusBuilder |
| Concrete Builder | LavenderBuilder |
| Concrete Builder | RoseBuilder |
| Director | GreenhouseDirector |
| Product | Plant |
| Product | Cactus |
| Product | Lavender |
| Product | Rose |

# Non – Functional Requirement

## Security:

* Only employees are permitted to add plants; clients are not permitted to do so.
* Only the items (Plants) will be accessible from the greenhouse.
* To prevent customers from entering, we will provide the staff with a short password.
* Data about customers, staff, and plant inventory should be securely stored and protected from unauthorized access.

## Usability:

* Both customers and employees will find the user interface to be tidy and pleasant.
* Both customers and employees will find the user interface to be tidy and pleasant.
* When a system error occurs, the consumer should receive clear instructions and feedback.

## Scalability:

* There will be a limit on how many plants the greenhouse can hold.

## Performance:

* If the consumer makes a mistake, like inputting the incorrect password, the response will be returned to him.
* Additionally, the client will be added to view the update after the data is added from the stuff, and the user logs out.

## Maintainability:

* For them to know whether to add or remove a function in the event of a merge dispute, we will be documenting our code.
* In our project, unit testing will be used. Every individual will conduct their own unit testing.
* We will also be adding GitHub Actions Linter and Tester for bonus marks.

## GUI:

* We will utilize our C++ code as our API and use HTML and JavaScript to construct a graphical user interface.
* To connect and view our pages online, we will also need a server.cpp file.