COS 214 FINAL PROJECT

Functional and non-functional requirements for a plant nursery:

Momina and Friends

Functional requirements:

**Decorator:**

**Category:** Structural  
**Function in our design:** It attaches additional responsibilities to our Plant object dynamically providing a flexible alternative to subclassing for Extending functionality.  
**Features:** PotDecorator has been implemented which adds a structural-based responsibility to our object and FertilizerDecorator has been added to provide behavioural-based responsibility to our object.

**Abstract Factory:**

**Category:** Creational **Function in our design:** An interface for GreenHouse dependant objects such as the Plants as products and their different types are subclassed correspondingly. **Features:** It ensures consistency with their corresponding characteristics as the different types of plants(Concrete Products) correspond to their Abstract products and Concrete Factories.

**Prototype:**

**Category:** Creational  
**Function in our design:** It allows a Plant object to be replicated, thereby creating a prototypical instance so that when an object of the same attributes is required, it does not have to be created from scratch again.  
**Features:** The clone() function allows duplicates of the Plant object to be created if needed for bulk orders.

**State:**

**Category:** Behavioural   
**Function in our design:** Growth state manages the different stages of growth in a plant. Health manages the health states of the plant so that we can know if more care is needed and the third one manages the ordering process of the plant. Order state allows us to move through the different steps when making a purchase from our plant shop.  
**Features:** The different states correspond to different actions that need to be done which helps with the plant lifecycle management as well as payment management.

**Inventory System and Iterator Design Pattern**

Our inventory system is implemented using a Binary Search Tree (BST) combined with the Iterator Design Pattern to efficiently manage and access plant data.

>The BST organises the inventory by plant species, where each node in the tree represents a single plant species.

>Each node contains a vector of Plant objects, allowing multiple instances of the same species to be stores together.

>When searching for a specific plant:  
 >The system first locates the relevant node in the BST(based on the plants species name).

>It then iterates through the vector inside that node to find the exact plant object that matches the query.

>Traversal through the BST is performed in order, which naturally organises the plants alphabetically by species name.

To enhance this process, the Iterator Design Pattern is used:  
**Category:** Behavioural  
**Function in our design:** The iterator provides a structured way to traverse the plants stored in the BST and their vectors without exposing the underlying data structures (e.g., BST nodes or vectors).  
**Features:**  
>The PlantIterator class iterates through vecotrs of Plant pointers, allowing for seamless access to each plant object.  
>It simplifies traversal by abstracting away the details of how plants are stored internally, supporting easy collection, listing, and manipulation of inventory items.

**Chain Of Responsibility**

**Category:** Behavioural

**Function in Our Design**: The Chain of Responsibility pattern is used to manage plant care tasks by linking multiple plant-care handlers in a chain, where each handler is responsible for checking and responding to a specific plant need. This pattern works closely with the Observer and Command patterns to automate and coordinate how plants receive care in the nursery.

**How it works:**

>Each Plant object has a tick() function that gradually decreases its care levels- water, sunlight, pruning and fertilizer – simulating natural decay over time.  
>When any of these levels drop below 60%, the GrowthObserver detects this change and moves the plant into a Needs Care State   
>This state change triggers the Chain of Responsibility which checks each plant requirements in sequence to identify what needs attention.  
>Ince the appropriate handles is found , it triggers the corresponding Command- WaterCommand, PruneCommand, FertilizerCommand to instruct a StaffMember to take care of the plant.

**Features:**>Provides a flexible and organized way to handle multiple plant-care responsibilities without hardcoding conditional logic.   
>Decouples the plant’s state from the specific action taken to restore it.  
>Works hand-in-hand with :  
The Observer pattern which detects the problem  
The Command Pattern, which executes the solution.

**Command:**

**Category:** Behavioural

**Function in our design**: a staff member is in charge of taking care of a plant when it is notified of a health change in the observer. The staff member then invokes one of the following commands to simulate “taking care of the plant”: Water, Sun, Fertilizer and Prune. This restores the health status of the plant back to good.

**Features:** we encapsulate a request from a plant as an object, letting us parametrize different commands with different requests or plant care to perform undoable operations because a plant cannot water or fertilize itself so a staffMember does it for us.

**Memento:**

**Category:** Behavioural

**Function in our design:** Memento works hand-in-hand with the Command pattern, allowing an entire queue of commands to be saved and executed at will. This allows for plant care routines, making it easy to carry out the same or similar care for numerous plants, reducing explicit function calls as the size of the system grows

**Features:** Memento stores a vector<Command\*> allowing for easy rerunning of command sets, preserving order.

**Observer:**

**Category:** Behavioural

**Function in our design:** It monitors the plant health state and notifies a staff member that the plant needs care. Works with the command so that a staff member can execute the necessary care to take care of the plant.

**Features:** we attach an observer to each plant growing in our nursery to monitor the growth and health of the plant. This is a one-to-many relationship between objects so that when a plant changes its health state, a growth monitor will notify staff members to update the health state of the plant.

**Mediator:**

**Category:** Behavioural

**Function in our design:** growthMediator mediates between the plant and staff members when a change in the plant occurs. The commMediator facilitates communication between the customer and staff members such as asking for recommendations and creating orders.

**Features:** By having 2 mediators we can encapsulate how a set of objects interact (how a customer interacts with a staffMember and how a staffMember interacts with a plant by using a growthMonitor.

**Composite:**

**Category:** Structural

**Function in our design:** Composite is used to model our clientele. Customers can either be individuals or be one of multiple registered with a company. This allows us to add specific behaviour to members of a company, as well as categorise and store our transactions more efficiently and descriptively

**Features:** Easy categorisation of users, easy maintenance of orders and payments, allows for exclusive discounts, features, etc.

Non-functional requirements (at least 3):

1. **The system needs to remain performant under a large number of Plant objects**

This is achieved with the use of efficient data structures, like the std::vector<Plant\*>. Such a data structure allows for O(1) addition, as well as O(N) searching and deletion. The use of Plant\* avoids unnecessary copies, allowing the system to be space efficient. Another benefit of this, is that when combined with the Plant class’ built-in categorisation, searching and deleting can be further optimised through the use of binary search for a sorted vector O(log N), or reducing the search space by filtering out unwanted categories. Efficient iterator patterns will implement this functionality.

1. **Reliability and security, the system needs to prevent exploitation and silent errors**

This is achieved through strict enforcing of the Singleton pattern in Inventory and Stock. This ensures that no duplicate instances of these structures can be created, ensuring that the system remains consistent throughout all reference to these classes. Silent failures where incomplete/inconsistent records are erroneously used are eliminated entirely from the system.

1. **Ease of use. The client should not need much knowledge of the inner system to perform tasks**

We ensure that the process is as automated as possible, with only high level methods are available to the user. Processes such as adding a new plant to the inventory are handled using 1-2 function calls at most, since most of the complex logic and instantiation is obfuscated from the user. This functionality is heavily aided by efficient use of patterns such as Observer, Abstract Factory, and Mediator. Processes such as growing a plant are handled gracefully and realistically through the use of a real-time notification system, where staff are notified when a plant is neglected for too long and requires maintenance. This heavily reduces the user’s ‘involvement’ in the system and thus leaves less room for user error

1. **Scalability. We need to be able to easily expand our catalog of plants.**

The use of an Abstract Factory as well as a multi-layered Plant hierarchy applies the ‘Open-Closed Principle’ and ensures that new factories and plants can be easily added by plugging in new classes through a common, abstract interface that is used throughout the system.

1. **Maintainability, The code needs to be easy to read and work on**

Doxygen will be used extensively to elucidate otherwise unintuitive and difficult functionality. Code will be written using consistent naming conventions. Coupling is unavoidable due to the nature of some patterns but needs to be kept to the minimum .

1. **Fault Tolerance and Robustness**

The system will perform validation at public entry points, and use try-catch blocks to handle errors. Fallback behaviour will be implemented to reduce the number of crashes from recoverable errors.