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.

**Iterator:**

Category: Behavioural

Function in our design: It allows us to collect and traverse the plants in our inventory, stock and an order without exposing its underlying representation or other details.

Features: PlantIterator iterates through the vector of Plant pointer objects

**Singleton:**

Category: Creational

Function in our design: Logically, we would only need a single instance of Inventory and Stock as this is how the store would be laid out in real life, therefore it would be best practice to enforce this in our code.

Maintaining only one instance of each class object is more secure, and it allows us to reduce the possibility of unpredictable failure due to accidental cloning, as well as more efficient resource management, as we know that our largest data store will not have redundant and wasteful copies.

Features: Implemented in the Inventory and Stock classes, to ensure that only one of each can exist.

**Strategy:**

Category: Behavioural

Function in our design: it allows us to choose a different plant care strategy based on the 4 different climate-based plants that we have such as its water requirement, sunlight and any additional nutrients that the plant might need to grow and mature without dying

Features: we define a family of algorithms that are all encapsulated and interchangeable to what the plant type might need to grow in terms of how often it needs to be watered, and how much sunlight it needs and environmental features for example a carnivorous plant needs insects and such for its nutrition to grow or a succulent has specific watering cycles every week. Also reduces the amount of conditional statements required in our implementation.

**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.