PHOTOSYNTECH

COS 214 Project Practical Assignment 5

Team:
Wilmar Smit u24584216
Johan Coetzer u24564584
Marcel Stoltz u24566552
Michael Tomlinson u24569705
Zamokuhle Zwane u23533413

Executive Summary: Plant Nursery Simulator

Project objective

The objective of the Photosyntech Plant Nursery Simulator is to design and implement a C++ system that models the daily operations of a dynamic plant nursery. The simulator will capture the greenhouse, staff activities, and customer interactions, while managing the complete lifecycle of diverse plant species. The focus is on building a flexible, maintainable, and scalable system through strict adherence to object-oriented principles and the effective application of more than ten Gang of Four (GoF) design patterns. By doing so, the project demonstrates how design patterns can bring structure and extensibility to complex systems, ensuring that new features can be introduced with minimal modification to existing code.

Diagrams

1. **UML Class Diagram**

   Serves as the foundational structural blueprint for the entire system. This comprehensive diagram explicitly defines the full class hierarchy and all **10+ design pattern implementations.** It illustrates **Generalization** (Inheritance) for polymorphism (e.g., Staff/Strategy classes), **Composition** for complex entities (e.g., Inventory containing `Plants`), and its organization into **packages** (colored units) for high maintainability. Crucially, it confirms the structural integration of the **Facade/Command** subsystem and the abstract classes required for the **State and Strategy** patterns.

1. **UML Activity Diagram**

   Models the Core Workflow. Illustrates the "Plant Creation" process using the Builder pattern across distinct roles**.** It uses **Swimlanes** ("`Facade`," "`planDirector`," and ”`plantBuilder`") to separate responsibilities. Actions like "`Build Identity`" and "`Build Pricing`" define the steps, while **Decisions** (e.g., "`Validate Maturity State`" and ”`Handle Invalid State`") show conditional branching within the workflow.

2. **UML State Diagram**

   Formalizes Object Behavior. Models the `Plant` lifecycle, demonstrating the **State** Design Pattern. It includes primary states ("`seed`," "`Mature`," and "`Dead`"), and utilizes a **Composite State** called "`mid`," which further details internal states like

"`NeedCare`" and "`Healthy`." Transitions like "`HealthDrop`" leading from "`Healthy`" to "`NeedCare`" based on unmet conditions are specifically illustrated.

3. **UML Sequence Diagram**

   Models Runtime Interaction. Models the dynamic flow of messages during a key transaction, such as a customer purchase or staff update. It specifically illustrates the **Mediator** and **Observer** patterns by showing how the `SalesFloorMediator` orchestrates communication between the `Customer` and `Staff` objects, ensuring the decoupled `Inventory` (the Subject) correctly notifies all registered `Staff` (Observers) of the change in stock.

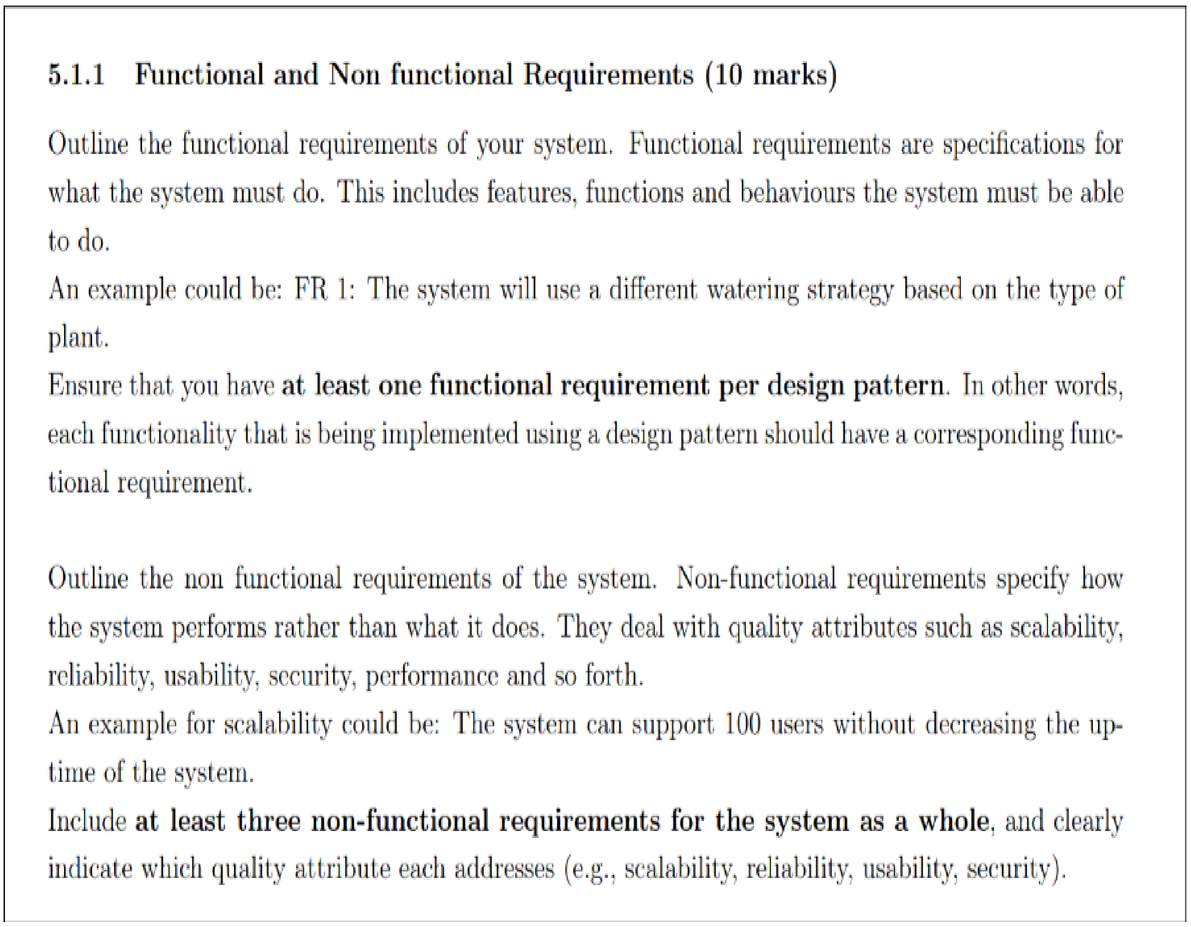
4. **UML Object Diagram**

   Instance Snapshot**.** Provides a snapshot of the system at a specific moment in time. It shows instantiated objects (e.g., specific `Staff` like "Wilmar," and specific `plants` like "Multiflora Rose") and their relationships to the plants (e.g., **state** of the plant, watering, and sun **strategies**) to demonstrate a real-world scenario of the implemented patterns.

5. **UML Communication Diagram**

   Interaction focus **:** Specifically illustrates the Builder pattern during the Plant creation process. The diagram models the client initiating the construction via the `Director`, which then orchestrates the `SunflowerBuilder` to apply specific configurations (including setting the required **Strategy** and **State** objects) before the final `Plant` object is retrieved.

Functional and Non-Functional Requirements



5.1.1 Functional and Non functional Requirements (10 marks)

Outline the functional requirements of your system. Functional requirements are specifications for what the system must do. This includes features, functions and behaviours the system must be able to do.

An example could be: FR 1: The system will use a different watering strategy based on the type of plant.

Ensure that you have **at least one functional requirement per design pattern**. In other words, each functionality that is being implemented using a design pattern should have a corresponding functional requirement.

Outline the non functional requirements of the system. Non-functional requirements specify how the system performs rather than what it does. They deal with quality attributes such as scalability, reliability, usability, security, performance and so forth.

An example for scalability could be: The system can support 100 users without decreasing the uptime of the system.

Include **at least three non-functional requirements for the system as a whole**, and clearly indicate which quality attribute each addresses (e.g., scalability, reliability, usability, security).

Figure: Screenshot of the the project specification

1. Non-Functional Requirements (Quality Attributes)

These specifications define how well the system performs, ensuring quality attributes such as performance, maintainability, usability, scalability, and reliability are met.

Quality Attribute	**Description:**
Performance	The system must be able to execute all staff care routines for a simulation of up to 10000 active plants within a single simulated day cycle to ensure smooth progression.
Maintainability/ Extensibility	The system must be implemented such that adding a new type of care routine or a new plant species requires modification to only a maximum of two existing classes or files.
Usability	The system must utilise a clear,text/visual based interface where all available actions and input parameters are displayed to the user upon request. A new user without any prior knowledge of the system should be able to navigate the system using only the text based system.
Scalability	The system must be able to handle a simulation with an inventory of up to 5000 unique plant instances without memory exhaustion or a noticeable decrease in simulation speed.
Reliability	The whole system must be designed with distinct modules for the staff,greenhouse and customer interactions.This also needs to ensure that a change in one area does not break any functionality in any other area.

2. Functional Requirements (Design Pattern Implementation)

The functional requirements specify what the system must do. Each major functionality is tied to a specific Design Pattern to ensure a flexible and robust object structure.

2.1 Creational Patterns

Design Pattern	Functional Requirement
Builder (Plant Creation)	· The system will provide builder classes that create plants with specific configurations such as water strategy, sun strategy and maturity state. · Director will be used to coordinate the construction of custom plants through the ConcreteBuilders, ensuring all required attributes are initialised.
Prototype (Plant Cloning)	· Cloning will allow new instances of plants to be created with copied attributes through the clone() method · Will be used to recreate very specific versions of plants multiple times.
Singleton (Inventory)	· A single global inventory will be shared to ensure that customers and staff access the same inventory. · The creation of multiple inventories will be prohibited.

2.2 Structural Patterns

Design Pattern	Functional Requirement
Composite (Inventory Structure)	· Plants will be organised into a structure using Composite, where these structures can be individual plants or groups of plants · The system will implement operations that travel through the structure and get the total amount of the contained plants. · Plants will be grouped by categories but will ensure that individual plants will be handled in the same manner.
Decorator (Plant Customisation)	· Plants will be customisable, such as adding a charm or decorating the pot. · Multiple decorators on a single plant will be possible, allowing wrapping on Plant objects. · The Plant interface will be maintained such that decorated plants are treated the same as regular plants.

2.3 Behavioural Patterns

Design Pattern	Functional Requirement
State (Plant Lifecycle)	· The system will manage maturity states, which in turn specifies minimum ages, growth rates and price increases/decreases. · Automatic transitions between states based on the age of the plant will be implemented
Strategy (Watering Strategy)	· Different watering strategies will be implemented that define the water amount and frequency for different types of plants · Watering strategies will change based on the lifecycle of the plant.

Strategy (Sun Strategy)	· Different sun strategies will be implemented that define the intensity and hours of sun needed for different types of plants
Composite (Inventory Structure)	· Plants will be organised into a structure using Composite, where these structures can be individual plants or groups of plants · The system will implement operations that travel through the structure and get the total amount of the contained plants. · Plants will be grouped by categories but will ensure that individual plants will be handled in the same manner.
Observer (Staff)	· Staff members will observe assigned plants' states to receive notifications about the health changes and lifecycle changes of the observed plant. · The system will notify the assigned staff of state changes.
Observer (Inventory Tracking)	· Staff and customers will be notified of inventory changes from sales and stock increases and decreases. · Multiple staff will be able to observe the same inventory, ensuring awareness of stock levels.
Mediator (Interactions Between Staff and Customers)	· A mediator will coordinate communication between staff and customers on the sales floor, allowing decoupling of the staff.
Iterator (Traversal for Seasonal Plants)	· The Iterator will traverse through the Composite Plant structure based on the season the plant is assigned to
Command	· Actions such as watering plants, selling plants and hiring/firing staff will be executed through Command objects.

Implementation Plan for COS 214 Plant Nursery Simulator

Step 1: Project Setup & Foundation

1. **Environment Setup:** Set up the C++ and header files and initialise the local repository.
2. **Define Core Enumerations:** Potentially create enumerations for seasonal plants and plant types

Step 2: Strategy Pattern - Plant Care

1. **Water Strategy Implementation:** Create the waterStrategy abstract class and implement the concrete strategy classes: minWater, midWater and highWater.
2. **Sun Strategy Implementation:** Create the sunStrategy abstract class and implement the concrete strategy classes: minSun, midSun and highSun.

Step 3: State Pattern - Plant Lifecycle

1. **Maturity State Implementation:** Create the maturityState abstract class and implement the concrete state classes: seed, mid, mature and dead.
2. **State Context Integration:** Add state management to the Plant base class.

Step 4: Plant Base Classes (Includes Prototype Subject)

1. **Plant Abstract Base Class:** Create the Plant abstract class, which holds references to the strategy and state components and defines the abstract method for the **Prototype pattern**.
2. **Concrete Plant Classes:** Implement concrete classes like Plant1, Plant2 and Plant3, as well as clone.

Step 5: Composite and Iterator Patterns - Inventory

1. **Component Interface:** Create the plantComponent abstract base class.
2. **Composite Implementation:** Implement the compositePlant class, which manages children components. Update concrete Plant classes to inherit from plantComponent.
3. **Iterator Implementation:** Create the Iterator abstract base class and implement the concrete plantIterator for traversal.

Step 6: Builder Pattern - Plant Construction

1. **Builder Interface:** Create the Builder abstract base class.
2. **Concrete Builders:** Implement concrete builder classes.
3. **Director Implementation:** Create the Director class.

Step 7: Observer Pattern - Staff & Monitoring

1. **Observer Interface:** Create the Observer abstract base class.
2. **Staff Implementation:** Create the staff class, which implements the Observer interface.
3. **Subject Integration:** Ensure the Plant and maturityState classes contain observer management functionality.

Step 8: Singleton Pattern - Stock/Inventory

1. **Singleton Implementation:** Create the stock singleton class with a private constructor and a static instance method.

Step 9: Mediator Pattern - Sales Floor

1. **Mediator & Colleague Setup:** Create the mediator abstract base class and the colleague abstract base class.
2. **Concrete Mediator:** Implement the salesFloor class.
3. **Colleagues Implementation:** Update the staff class and create the customer class to inherit from colleague.

Step 10: Decorator Pattern - Plant Customisation

1. **Decorator Base Class:** Create the plantDecorator abstract class, which is a wrapper around a plantComponent.
2. **Concrete Decorators:** Implement concrete decorator classes such as potDecoration and charmDecoration.

Step 11: Aggregate Pattern - Seasonal Collections

1. **Aggregate & Iterator Classes:** Create the seasonal aggregate classes (e.g., AggSummerIterator, AggWinterIterator, etc.) that manage and provide iterators for plant collections.

Step 12: Facade & Command Patterns - GUI

1. **Facade Implementation:** Create the facade class to simplify the GUI interface.
2. **Command Pattern:** Create the Command abstract base class and implement concrete command classes (e.g., WaterPlantCommand, SellPlantCommand).

Step 13: System Integration & Testing

1. **Full System Integration:** Connect all subsystems (inventory, staff, customers, GUI) and implement the simulation loop.

Step 14: Documentation & Finalisation

1. **Code Documentation:** Add header comments and document all design patterns used.
2. **UML Diagrams:** Finalise class, object, and sequence diagrams.
3. **Written Documentation:** Write the system architecture overview and final submission documents.