COS214 FINAL PROJECT

Green Fingers

GERARD JORDAAN (U24568385)

MEGAN NORVAL (U24575764)

ALLYSON ANDRÉ (U23525984)

MORGAN WATTRUS (U23541068)

ANÉ BURGER (U24565068)

2 NOVEMBER 2025

1. Introduction
2. Functional Requirements
    2.1 Game Selection Menu
    2.2 Plant Creation
    2.3 Buying Seeds for the Greenhouse
    2.4 Plant-Staff Notification System with Individualized Life-Cycles
        2.4.1 Observer Registration and Management
        2.4.2 Notification Mechanism
        2.4.3 Observer Reaction and State Update
    2.5 Plant State Management
        2.5.1 State Representation
        2.5.2 State Transitions
    2.6 Player Shifts to Staff Perspective, NPCs Get Created
    2.7 Customer Interaction
    2.8 Handles Client Requests
        2.8.1 Example
    2.9 Plant Sales & Returns
    2.10 Customers can purchase plants
    2.11 Logging System
    2.12 C++ API Engine
    2.13 End of day Summary
3. Non-Functional Requirements
    3.1 Scalability
    3.2 Reliability
    3.3 Performance
    3.4 Usability
4. Nursery Management Research
    4.1 Research Brief
    4.2 Assumptions
        4.2.1 Controlled Environment
        4.2.2 Simplified Care Routines
        4.2.3 Predictable Customer Behavior
        4.2.4 Pattern based system design
        4.2.5 Stable system resources
    4.3 References
5. UML Diagrams
    5.1 State Diagram
    5.2 Activity Diagram
    5.3 Sequence Diagram
    5.4 Object Diagram
    5.5 Communication Diagram

GITHUB

NOTION

DIAGRAMS

DOXYGEN

1. Introduction

This report outlines the functional and non-functional requirements for the implementation of object-oriented design patterns in C++ as part of our final project, GreenFingers - a gamified plant nursery simulation.

In GreenFingers, the player manages a virtual nursery that mirrors the operations of a real one. The gameplay involves purchasing, nurturing, and selling various plants while overseeing the nursery's day-to-day activities. Each plant has distinct care requirements and life cycles that must be carefully managed. The player directs staff members who perform tasks in order to maintain plant health according to the player's instructions. At times, the player shifts perspective, from acting as a high-level manager to taking on the role of a staff member assisting customers and facilitating sales.

The system's architecture has been designed with extensibility, maintainability, and scalability in mind, achieved through the deliberate and meaningful application of design patterns. These patterns provide elegant solutions to recurring design problems and enhance the system's modularity.

By simulating a nursery through a structured, pattern-driven design, GreenFingers demonstrates how software engineering principles can be applied creatively to model a dynamic, evolving environment while maintaining robust and reusable code.

Link to the GitHub repository

Link to Google Docs version of report

2. Functional Requirements

Section 5.4.2 from the specification.

2.1 Game Selection Menu

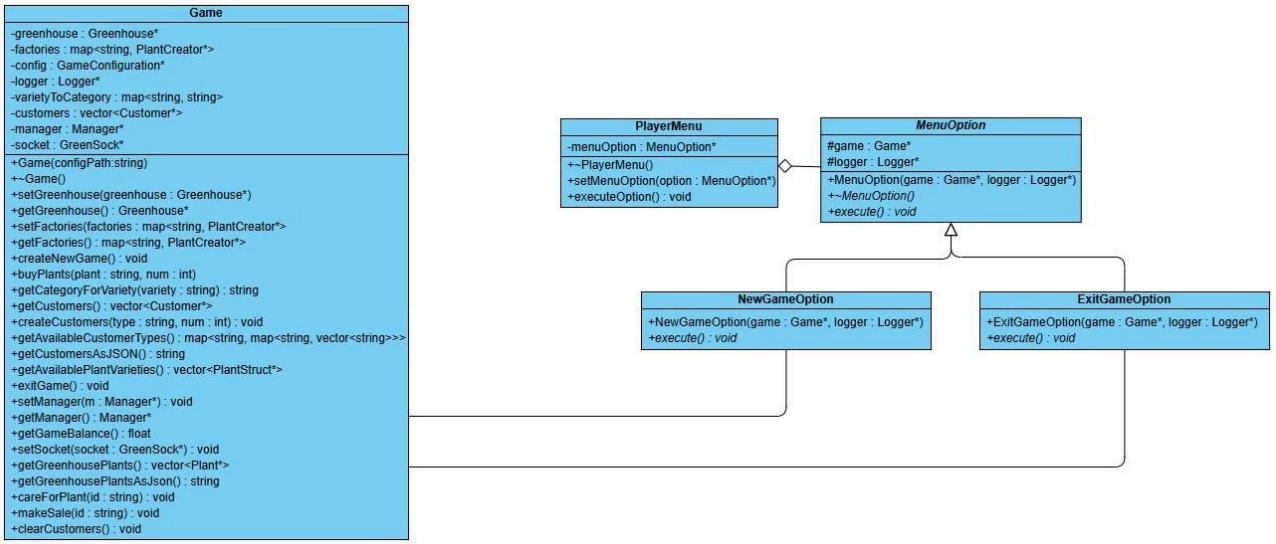
Upon launching GreenFingers, the player is greeted with a landing page offering three primary options: "New Game", "Tutorial", and "Exit."

- **New Game**: Starts a new gameplay session by sending a request to initialize and load a new game instance.
- **Tutorial**: Opens a static page that provides textual instructions on how to play the game. This page is separate from the backend and does not interact with the system's core logic.
- **Exit**: While inactive on the landing page, the Exit option functions as intended once a game has started. When selected during gameplay, it terminates the current session and resets the system, preparing it to accept a new game.

The Command design pattern is employed to encapsulate the logic behind creating and exiting a game. Each menu option corresponds to a specific command object responsible for executing its associated action. This design provides several advantages:

- **Robust error handling:** The *ConcreteCommand* classes can catch and manage exceptions that occur during execution.
- **Extensibility:** The methods invoked by the commands can be modified without changing how the client of the pattern interacts with them.
- **Separation of concerns:** The pattern decouples the user interface from backend control logic, improving maintainability and scalability.

Through this implementation, the system achieves a modular and flexible design that simplifies command management and future feature expansion.



2.2 Plant Creation

The system must be able to create different plant types (e.g. Flowers, Succulent and Tree plants) without the player or higher-level components directly instantiating them. This requires the use of a specialized factory method responsible for creating the appropriate plant subclasses.

Implementation using the **Factory Method**:

This requirement is implemented using the Factory Method design pattern. The system defines an abstract *PlantCreator* interface that declares a method for creating plant instances. Each *ConcreteCreator* (e.g., *FlowerCreator*, *SucculentCreator*, *TreeCreator*) implements this interface and encapsulates the logic for creating its corresponding plant type.

When a new game is started, the Game subsystem instantiates the appropriate factories and delegates creation to it. Each factory then produces the correct type of

Plant by invoking the clone() method on its maintained prototype. The clone operation is essential because each plant runs on its own dedicated thread to simulate its independent life cycle - such as growth, health, and environmental interactions - without affecting other plants in the system.

This design ensures that:

- Plant creation is handled through factory instantiation, maintaining loose coupling between game logic and object creation.
- New plant types can be added by simply defining a new ConcreteCreator and prototype, without altering existing code.
- The system provides a consistent and extensible interface for plant creation while enabling concurrent simulation of individual plant lifetimes.

In summary, the Factory Method pattern fulfills the functional requirement of dynamic plant creation through explicitly instantiated factories that manage object construction and thread initialization in a modular, maintainable manner.

The class diagram for the Factory Method will be shown together with the Prototype design pattern on the next page.

2.3 Buying Seeds for the Greenhouse

The system will allow new Plant objects (e.g., Cactus, Daisy, Oak) to be created by cloning predefined prototype instances rather than instantiating them directly. This approach enables efficient and consistent creation of plants while providing explicit lifetime management for each instance.

Implementation using the **Prototype Pattern**:

This requirement is implemented using the Prototype design pattern, which allows the system to duplicate existing, preconfigured plant prototypes instead of constructing new objects from scratch. Each prototype encapsulates the default state and configuration of its plant type.

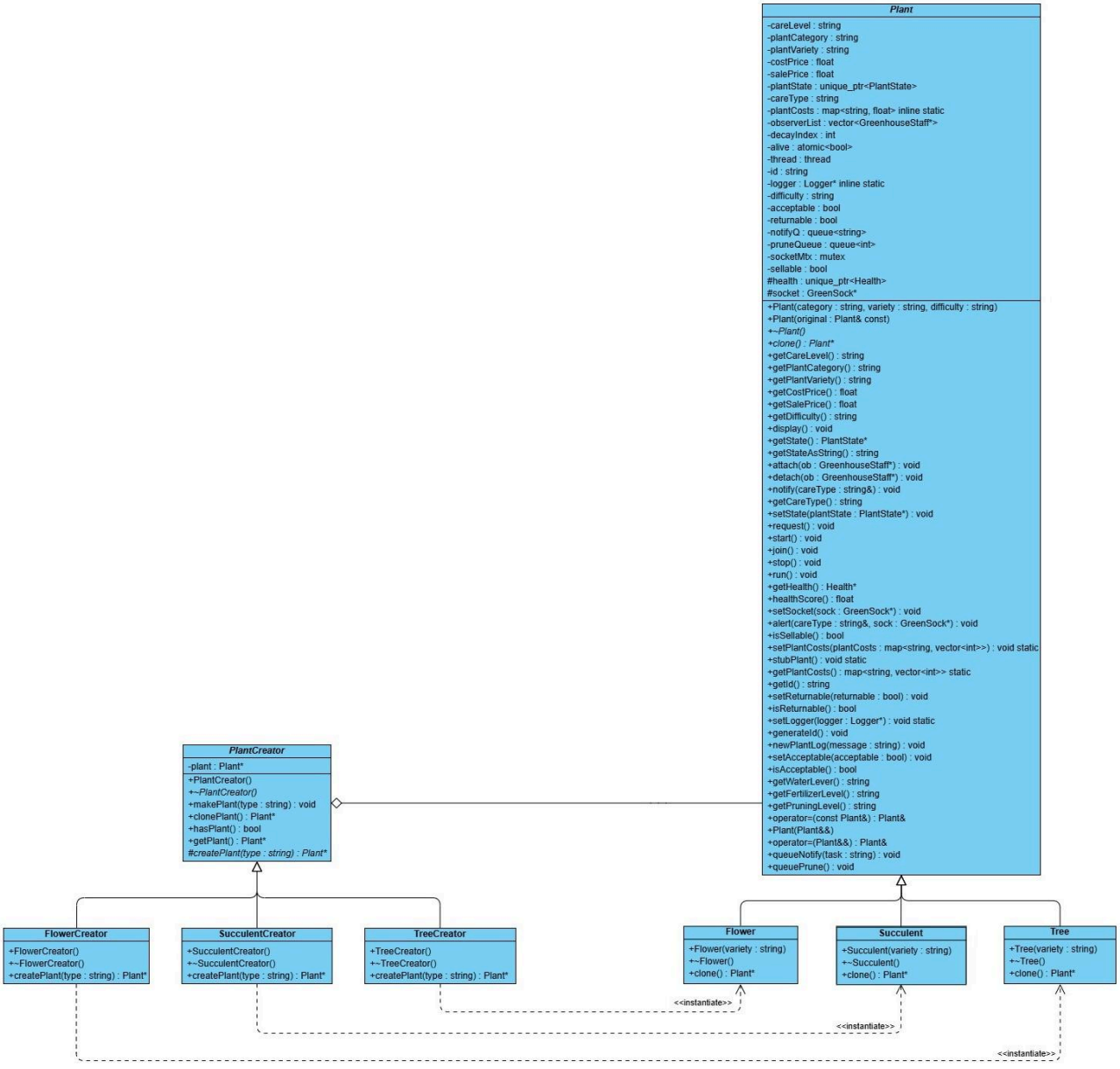
When the player chooses to buy seeds, the system references the corresponding prototype (e.g., FlowerPrototype, SucculentPrototype, or TreePrototype) and clones it to create a new plant. During cloning, the system also starts a dedicated thread for the cloned plant to track its independent life cycle. The lifecycle affects the *Health* object of the plant, which encapsulates all attributes related to the wellbeing of the plant, if a plant is left uncared for it will decay and eventually die. This ensures that each plant operates autonomously while sharing the same baseline characteristics as its prototype.

The Prototype pattern provides:

- **Efficiency:** New plants are created through cloning, avoiding repeated instantiation overhead.

- **Explicit lifetime management:** Each cloned plant initiates its own thread upon creation, ensuring isolated and concurrent life-cycle simulation.
- **Consistency:** All cloned plants maintain uniform base properties while developing independently.

In summary, the Prototype pattern fulfills the requirement of efficiently “buying seeds” by cloning plant prototypes, enabling scalable plant creation and individual life-cycle management across all plant varieties.



2.4 Plant-Staff Notification System with Individualized Life-Cycles

The system will use the **Observer** design pattern to enable an indirect, event-driven notification mechanism between *Plant* objects and the corresponding *GreenhouseStaff* responsible for their care. Each plant acts as a subject that can be observed by one or more staff observers.The player can submit a request to the queue, which is later processed to alert the staff. This maintains control over notification flow and concurrency.

2.4.1 Observer Registration and Management

Each Plant maintains a list of registered observers (staff members) responsible for its care.

The system provides methods for observers to attach or detach themselves from specific plant subjects.

Only observers of the corresponding type (e.g., FlowerStaff for Flower) are permitted to register with that plant type, ensuring proper delegation of responsibilities.

2.4.2 Notification Mechanism (Queued Updates)

When a player indicates that the staff should take care of the plant, the care instruction is stored in a queue.

The queue is periodically processed by the system to dispatch these events to the appropriate staff observers.

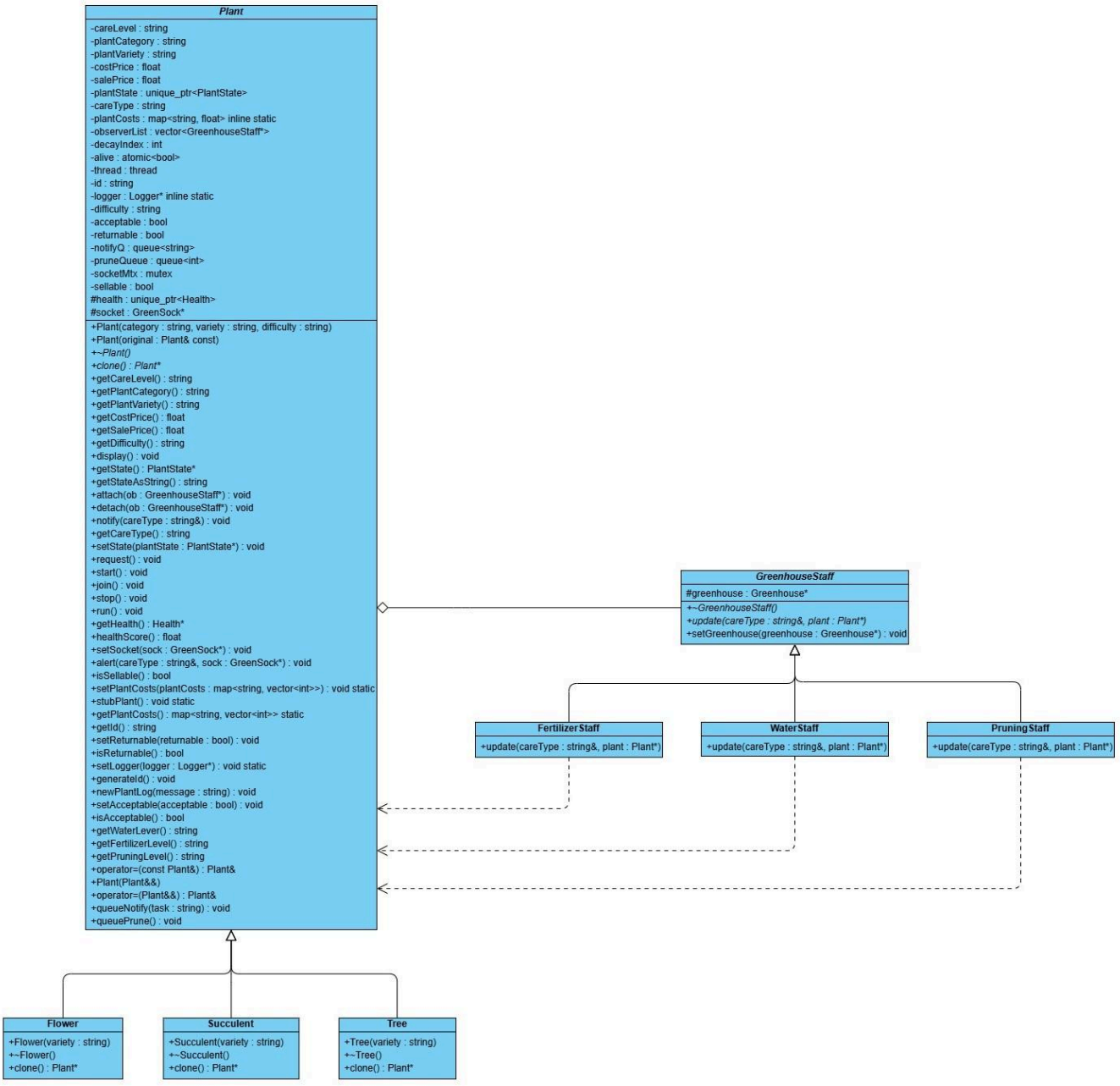
This approach ensures thread-safe communication between concurrently running plant lifecycles and the staff responsible for their care.

2.4.3 Observer Reaction and State Update

Each subclass of GreenhouseStaff implements an update() method defining the care procedure for its role.

When a notification event for a specific plant is dequeued and processed, the relevant observer executes its update routine - performing the required care actions and adjusting the plant's internal state (e.g., health, hydration, maturity).

In summary, the Observer pattern, combined with an interactive queued event-processing mechanism, fulfills the requirement for an efficient and thread-safe Plant–Staff notification system, enabling individualized lifecycle management and responsive care.



2.5 Plant State Management

The system will use the State design pattern to manage each plant's lifecycle as it transitions between stages of growth and sellability. A plant can exist in one of two primary states: NotSellable or Sellable. The plant's behavior is determined by its current state. This allows each plant to alter its functionality dynamically without conditional logic scattered throughout the codebase.

2.5.1 State Representation

Each Plant maintains a reference to a state object representing its current condition (e.g., NotSellable or Sellable).

By delegating behaviors to state objects, plants modify their responses automatically as their state changes, ensuring consistent and modular behavior management.

2.5.2 State Transitions

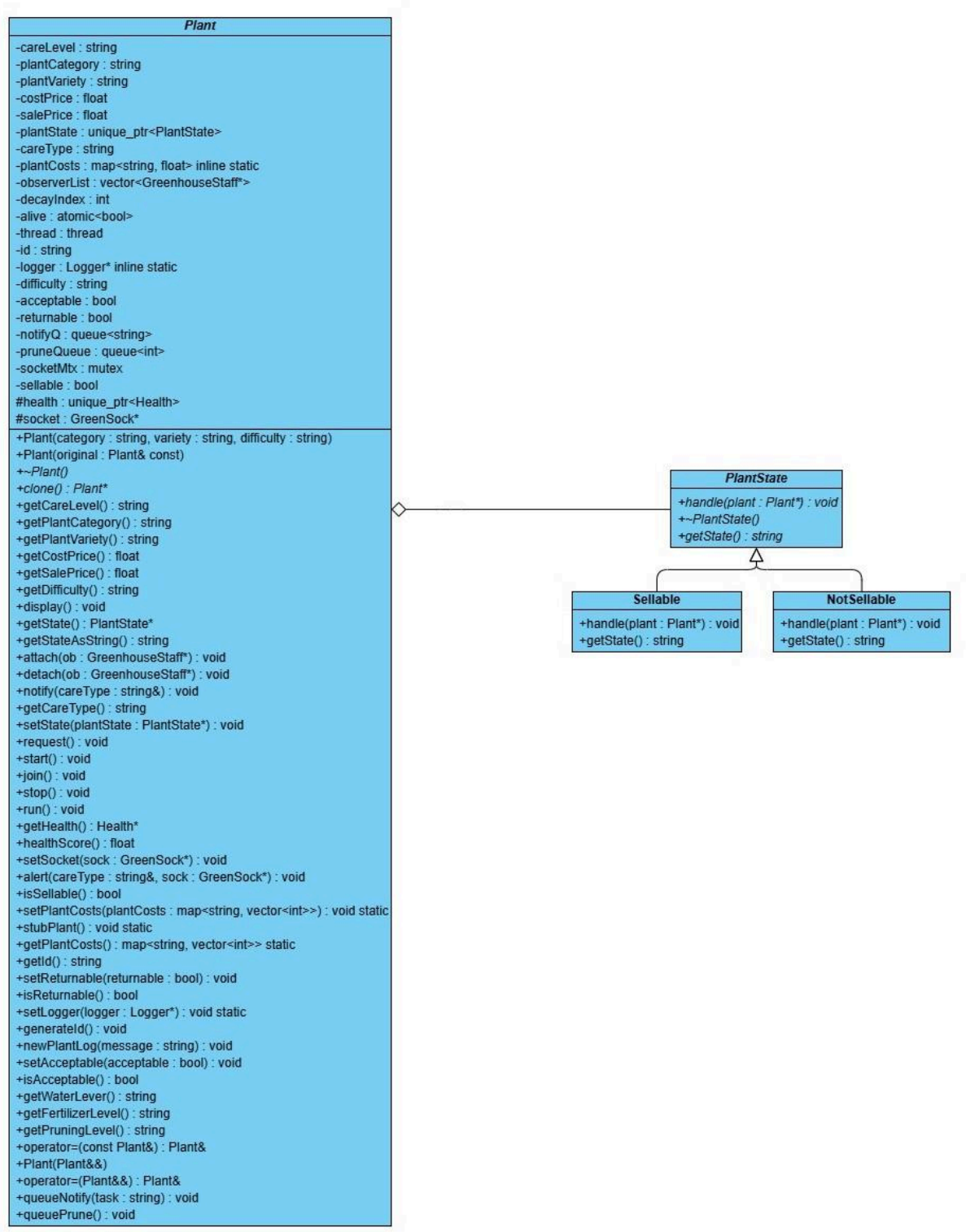
The system will allow a plant to transition from NotSellable to Sellable once it reaches a predefined maturity level.

Maturity increases as the player provides care.

When these conditions are satisfied, the plant's state reference is updated to the Sellable state, altering its behavior accordingly.

For example, when a player's care actions cause a Flower (of the variety Tulip) to reach the required maturity threshold, its internal state is switched from NotSellable to Sellable. From that point onward, it behaves as a sellable plant, eligible to be moved to the sales floor or sold to a customer.

In summary, the State pattern fulfills this requirement by defining clear boundaries between plant behaviors and enabling seamless transitions based on maturity. This approach simplifies lifecycle management, ensures encapsulation, and allows new states or behaviors to be introduced without modifying existing code.



2.6 Player Shifts to Staff Perspective, NPCs Get Created

When the player shifts from a managerial to a staff perspective, the system will generate interactive customer NPCs (Non-Playable Characters) that the player can engage with.

The **Builder** design pattern is used to manage the creation of these complex customer objects, separating the construction process from their specific representation.

The final product of this process is a stringified JSON object representing the fully built customer, which is then returned as a response to an API request.

This allows seamless communication between the backend game logic and the frontend interface, where the constructed customer data (including dialogue, name, and plant preferences) is rendered dynamically for player interaction.

This approach allows the system to reuse the same step-by-step build process, such as selecting dialogue options, assigning unique customer names, and preparing offered plants, while producing customers with different characteristics and levels of botanical knowledge.

2.6.1 Customer Construction Process

The system defines a *CustomerBuilder* interface outlining the process for assembling customer components (e.g., name, dialogue, and plant preferences based on customer knowledge levels).

Concrete builders such as *IgnorantCustomerBuilder*, *AverageCustomerBuilder*, and *GreenfingerCustomerBuilder* implement this interface, customizing how each customer type is assembled.

Despite variations in the resulting customers, the *Director* class follows the same build sequence to create them — ensuring a uniform construction workflow while supporting variety in the final output.

Once the build process is complete, the constructed Customer object is serialized into a JSON-formatted string, ready to be sent as the payload of an API response.

2.6.2 Customer Representation and Variation

Each Customer instance produced by the Builder pattern has unique combinations of dialogue, preferences, and behaviors.

The Visitor design pattern (discussed later) determines which plants are offered to the customer during interaction.

For example:

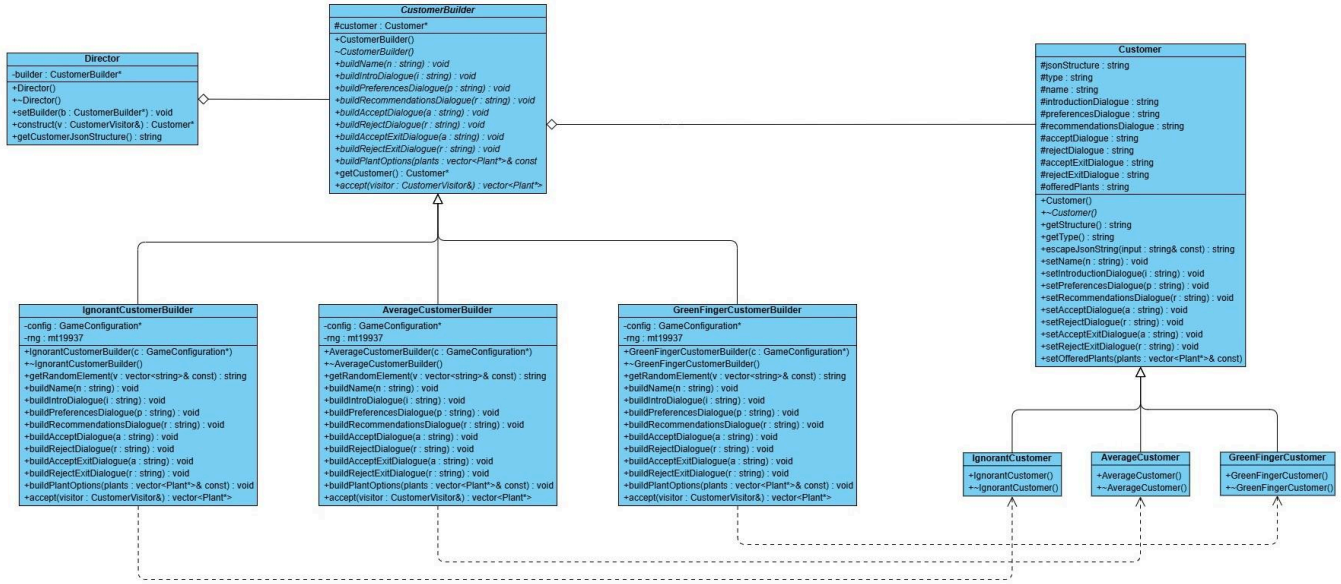
- The *IgnorantCustomer* may have limited plant knowledge and humorous or uncertain dialogue, preferring plants that require minimal care.
- The *AverageCustomer* displays balanced knowledge and moderate confidence, preferring plants with medium care requirements.
- The *GreenfingerCustomer* demonstrates expertise and precise preferences, requesting specialized plants with high care demands.

By isolating the construction steps from customer representation, the system maintains a clear separation of concerns.

This design enables consistent data serialization and network communication while allowing new customer archetypes to be added by defining new builders without modifying the existing build process.

In summary, the Builder pattern fulfills this requirement by constructing complex customer NPCs as stringified JSON responses.

This ensures a consistent, extensible, and API-compatible mechanism for generating dynamic and varied customer data, enhancing both gameplay variation and system modularity.



2.7 Handles Client Requests

When a new Customer is being built, the system will determine which plants to offer that customer using the **Visitor** design pattern. The Visitor is applied during the Builder process, not afterward, allowing the evaluation logic to remain separate from both the Customer and Plant classes.

The Visitor pattern enables new operations to be performed on elements (in this case, the *ConcreteBuilders*) without modifying their underlying structure. Each visitor represents a different customer difficulty category and encapsulates the logic for selecting appropriate plants based on their care requirements.

The Builder integrates the Visitor to compile a list of plants to be offered and determine whether each would be accepted by the customer. The resulting data is then included in the stringified JSON object that represents the fully built customer and is returned as an API response.

2.7.1 Visitor Evaluation Process

The system defines multiple concrete visitors used during customer construction:

- *VisitEasyCustomer* - selects plants suited for customers with limited botanical knowledge.
- *VisitMediumCustomer* - selects plants suited for customers with moderate knowledge.
- *VisitHardCustomer* - selects plants suited for expert customers.

Each visitor evaluates the plants available on the SalesFloor and compiles a list of up to five plants (or fewer, depending on availability).

This selection is based on plant difficulty, ensuring that:

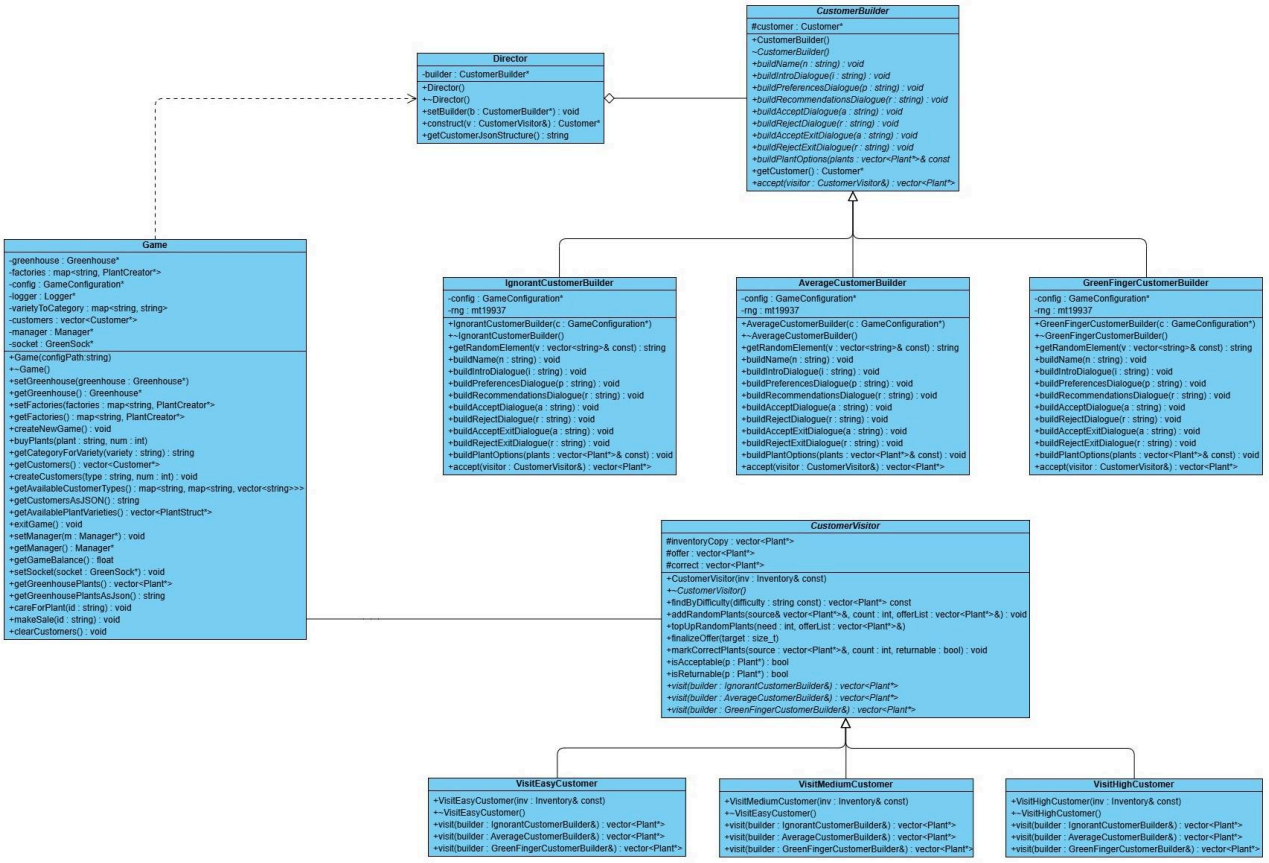
- Easy customers (e.g., IgnorantCustomer) accept low-maintenance plants.
- Medium customers (e.g., AverageCustomer) accept moderate-care plants.
- Hard customers (e.g., GreenfingerCustomer) accept specialized, high-maintenance plants

Where *accepting* is defined as the customer purchasing the plant, since it satisfies their needs

Each customer will accept a plant only if its difficulty aligns with their corresponding care level.

By integrating the Visitor directly into the Builder, the system maintains a clean separation between customer construction and evaluation logic, enabling new visitor types or selection strategies to be added without modifying existing customer or plant classes.

In summary, the Visitor design pattern fulfills this requirement by allowing the system to perform plant selection dynamically during the customer-building process, ensuring extensibility, maintainability, and realistic customer-plant interactions in the serialized API response.



2.8 Plant Sales

The system will handle all financial and inventory-related events through the **Strategy** design pattern. Each transaction, such as a plant sale or restock, is processed using a corresponding *TransactionStrategy*, allowing the system to update the nursery's balance and inventory dynamically without modifying the core *Transaction* class.

2.8.1 Strategy Implementation

The system defines several concrete strategies to manage different transaction types:

- **SaleStrategy** - Applied when a customer purchases a plant. This strategy increases the balance and decreases the available stock.
- **RestockStrategy** - Applied when the player buys new stock for the nursery. This strategy decreases the balance and increases inventory.
- **RefundStrategy** - Mentioned for extensibility. Although not implemented in the current version, it would enable future support for customer returns by reversing the sale transaction.

Each transaction event delegates execution to the selected strategy, ensuring consistent balance updates and financial tracking while maintaining a clean separation between business logic and transaction handling.

2.8.2 Transaction Flow

A sales-related event (e.g., Sale or Restock) is initiated by player interaction.

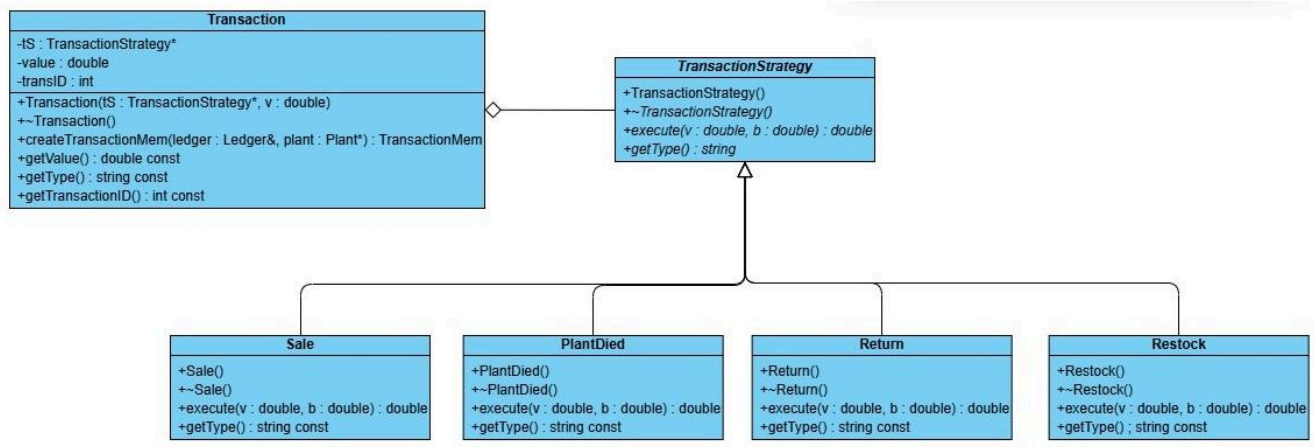
The system creates a *Transaction* object and associates it with the appropriate *TransactionStrategy*.

The strategy executes its apply() method, updating both the balance and inventory accordingly.

Each processed transaction is recorded in the *TransactionHistory*, maintaining a log of all financial events for auditing and potential future extensions.

The system currently logs transactions individually. An end-of-day financial summary was not implemented but could be added by aggregating the entries in TransactionHistory.

In summary, the Strategy pattern fulfills this requirement by providing a flexible and extensible mechanism for processing nursery transactions. It cleanly separates the logic for handling sales, restocking, and potential refunds, ensuring that new transaction types can be integrated seamlessly without altering the existing Transaction class or disrupting the system's financial integrity.



2.9 Logging System

The system will use the **Decorator** design pattern to extend the functionality of the logging subsystem without altering the structure of the original Logger. This design allows new features (such as console output) to be dynamically added to the base logging mechanism.

The Decorator pattern enables recursive, layered composition of functionality, where each decorator wraps another logger instance ("onion-layered") to progressively enhance behavior. This approach ensures extensibility, allowing future enhancements to be integrated without modifying existing code or breaking the base logger interface.

All logs are automatically timestamped when written, providing clear temporal tracking for debugging and audit purposes.

2.9.1 Logging Components

The logging subsystem consists of the following key participants:

Logger (Component):

Defines the base newLog(message: string) operation for creating log entries.

BasicLogger (ConcreteComponent):

Implements the core file-based logging functionality. It writes timestamped log messages to a designated file, providing the foundation for all logging operations.

LogDecorator (Decorator):

Serves as a wrapper class that extends the behavior of any Logger implementation dynamically. It maintains a reference to another Logger object and delegates basic logging while allowing additional functionality to be layered on top.

CoutAndLog (ConcreteDecorator):

The currently implemented decorator that adds console output functionality on top of file-based logging. When a log entry is created, it is both written to the file and displayed on the console simultaneously.

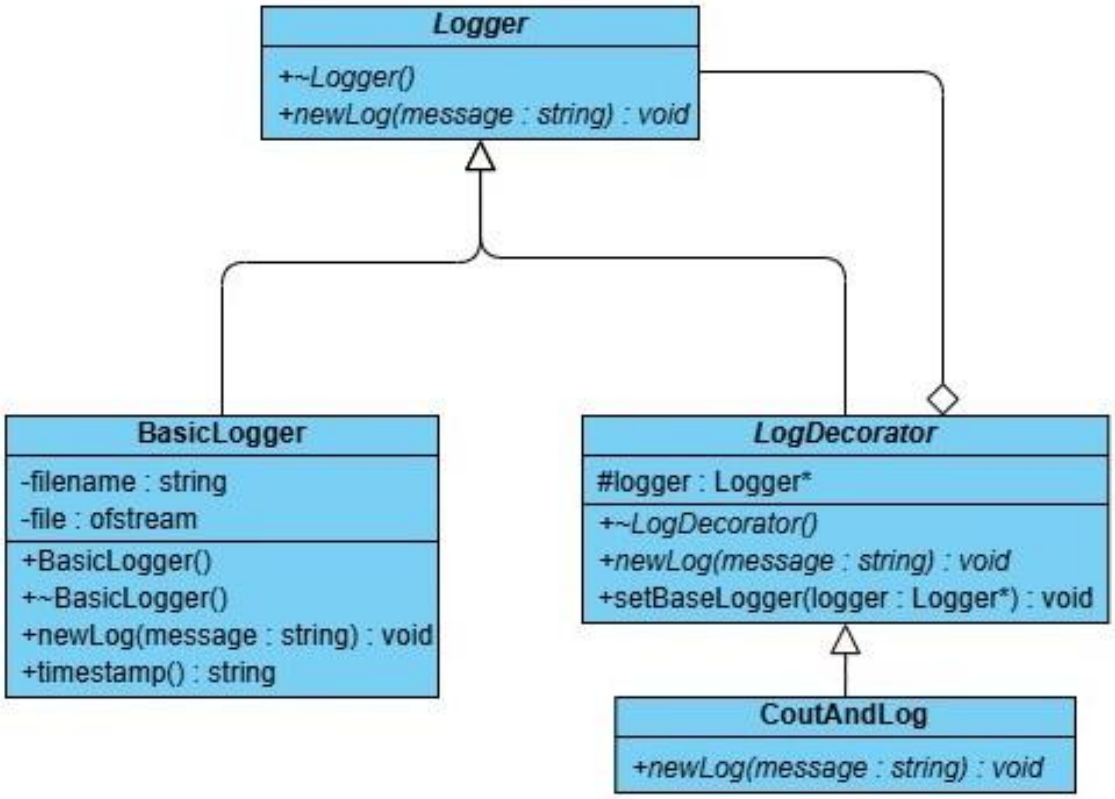
2.9.2 Extensibility and Future Enhancements

While the current system includes only one concrete decorator (CoutAndLog), the Decorator pattern allows additional decorators to be added easily. Examples of potential extensions include:

- **ErrorHighlightDecorator**: Adds color-coded console output to visually distinguish warnings or errors.
- **RemoteLogDecorator**: Extends the logger to send log entries to a remote server or monitoring dashboard in addition to local output.

Because decorators can be layered recursively, these extensions can be combined. For example, the *BasicLogger* could be wrapped by both the *CoutAndLog* and *RemoteLogDecorator*, providing file logging, console output, and remote transmission simultaneously.

In summary, the Decorator design pattern fulfills this requirement by providing a flexible, extensible, and composable logging framework. It enhances functionality dynamically without modifying existing components, ensuring maintainable and scalable logging behavior that can adapt to future debugging and monitoring needs.



2.10 Internal API

The system will use the **Façade** design pattern to unify and simplify interactions between internal subsystems, enabling the definition of clear and concise REST API endpoints. The *Game* class serves as the central Façade, coordinating operations between the system's core components such as the *Manager*, *Greenhouse*, *SalesFloor*, and *Transaction* subsystems.

Instead of requiring each endpoint to directly access and manipulate these individual modules, the Façade provides a single unified interface that handles the necessary coordination and delegation behind the scenes. This design abstracts the complexity of subsystem interactions, ensuring that the REST API layer remains simple, readable, and easy to maintain.

2.10.1 Facade Structure and Role

Game (Façade): Acts as the central coordination point for all major subsystems. It defines high-level methods that encapsulate multiple internal operations, such as creating new plants, managing staff care routines, or processing sales transactions.

Subsystems: Include the Manager, Greenhouse, SalesFloor, Transaction, etc. components. Each subsystem handles its own domain logic, but their interactions are mediated exclusively through the Game facade.

REST API (Client): The REST API acts as the client of the Facade. Each endpoint calls the appropriate Game methods rather than dealing with the internal structure directly, ensuring consistent and predictable system behavior.

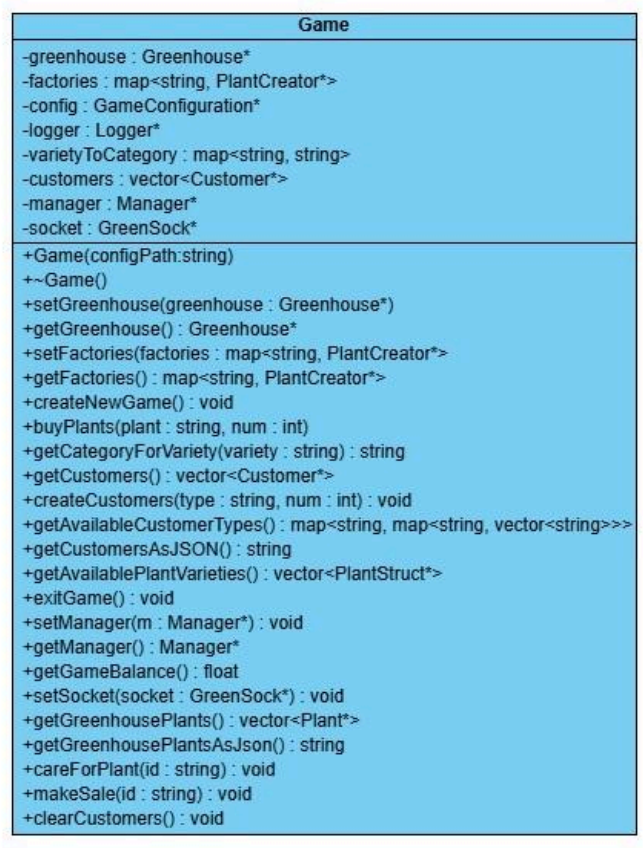
2.10.2 API Simplification and Maintainability

By introducing the Game Façade, the system allows developers to define concise, high-level REST API endpoints that remain decoupled from the complexity of the internal architecture. This ensures:

- Ease of maintenance: Changes within subsystems do not affect the API structure.
- Clarity: Each endpoint corresponds to a single, clearly defined operation in the Game interface.
- Encapsulation: Internal logic and dependencies are hidden from the API layer, reducing coupling and potential errors.

For example, a single API request such as */api/greenhouse/buy-plants* can internally trigger coordinated operations across multiple modules (inventory update, balance adjustment, and plant creation) via the Game facade without exposing that complexity externally.

In summary, the Façade pattern fulfills this requirement by acting as an intermediary between the REST API and the system's internal modules. It unifies subsystem interactions into a cohesive interface, enabling clean, maintainable, and easily extendable API endpoints that simplify the overall architecture while preserving system flexibility.



2.11 Loading a Game Configuration

The system will use the **Strategy** design pattern to manage the loading of the game configuration. The configuration defines all predefined categories, plant varieties, sale and cost prices, and the atomic components used by the Builder pattern to construct customers.

The Strategy pattern allows the algorithm responsible for loading this configuration to be interchanged dynamically, enabling the same end result (loading a valid configuration into the game) through multiple possible implementations.

The configuration data (currently stored in JSON format) specifies entities such as customers, their dialogue, preferences, and interactions, as well as plant categories (flowers, succulents, and trees) with corresponding cost and sale prices.

2.11.1 Strategy Structure and Participants

Game (Context):

Acts as the client of the configuration strategy. The Game facade interacts only with the unified GameConfiguration interface, ensuring that the loading process remains independent of the specific configuration format.

GameConfiguration (Strategy):

Defines the unified interface for configuration-loading strategies. From the perspective of the Game, any configuration source (JSON, YAML, or otherwise) can be used transparently through this interface.

JSONGameConfiguration (ConcreteStrategy):

Implements configuration loading from a JSON file. It parses and loads predefined plant categories, varieties, and customer data, making them accessible to the Game and other subsystems.

2.11.2 Extensibility and Alternative Strategies

The Strategy pattern enables extensibility by allowing new configuration formats or data sources to be introduced without modifying existing system components.

For example:

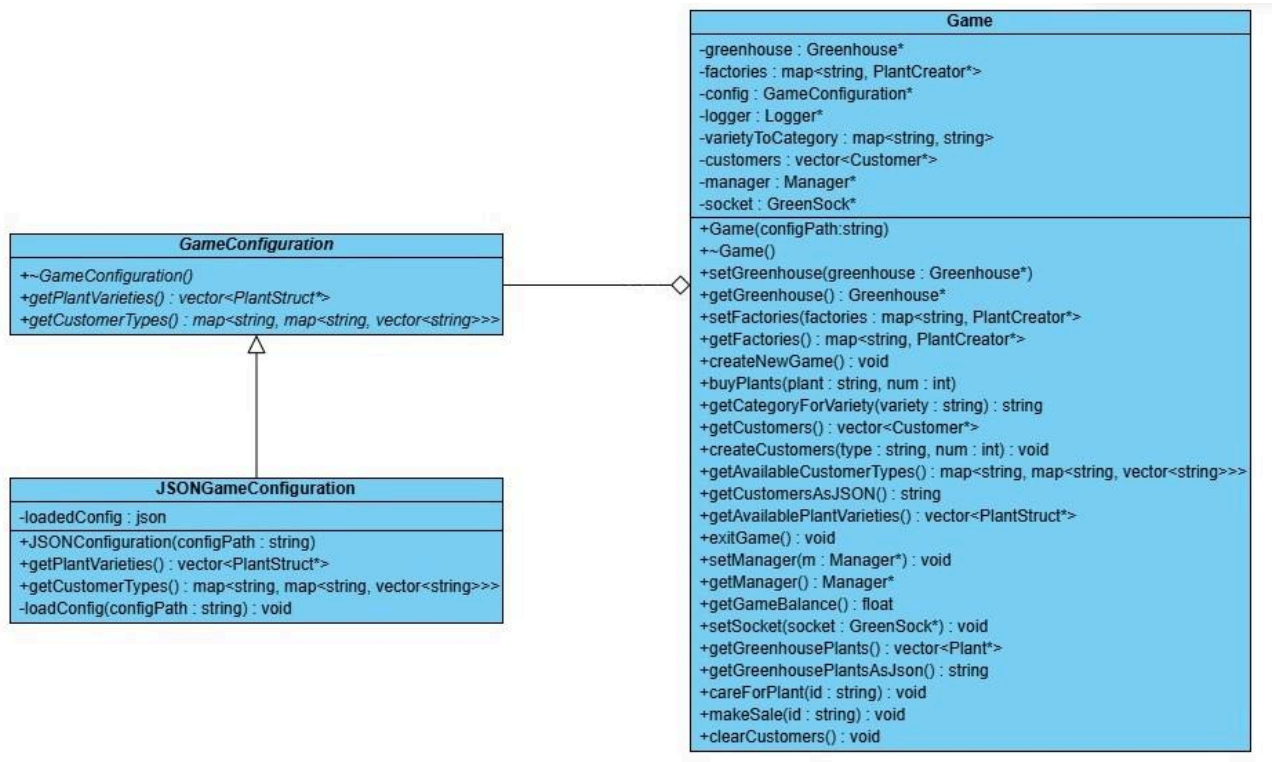
- **YAMLGameConfiguration**: Could load the same configuration from a YAML file for greater human readability.
- **DatabaseGameConfiguration:** Could retrieve the configuration dynamically from a database to allow real-time updates or multiplayer synchronization.
- **RemoteGameConfiguration:** Could load configuration data from a web service or cloud storage for centralized management.

Each of these concrete strategies would implement the same GameConfiguration interface, allowing the Game facade to swap between them seamlessly at runtime.

2.12.3 Benefits

- **Flexibility:** The system can change how configuration data is sourced or parsed without altering the Game facade or other components.
- **Separation of Concerns:** Configuration loading is isolated from game logic, improving maintainability.
- **Extensibility:** Future formats (e.g., YAML, database, remote API) can be supported by defining new strategies.

In summary, the Strategy pattern fulfills this requirement by defining a flexible and extensible mechanism for loading game configuration data. The current implementation uses a JSON-based strategy, but the system can easily adopt other configuration formats or sources in the future without changing the internal API or game logic.



3. Non-Functional Requirements

Section 5.4.2 from the specification.

3.1 Scalability

The following descriptions assume that scalability refers to the system's ability to be easily expanded or adapted to accommodate increased complexity or functionality as the need arises.

The API is designed for easy extensibility because the class defining it serves as the Fącade participant in the **Fącade** design pattern. This design centralises access to the system's functionality while maintaining low coupling between components. New

API endpoints can be constructed by chaining subroutines, allowing endpoint implementations to remain modular and separated from their definitions.

The *Logger* system shall support the addition of new decorators without altering the existing classes, ensuring the architecture remains open for extension but closed for modification.

The *Staff* and *Player* hierarchies are designed to be easily extensible, allowing new staff or player types to be added without altering existing structures. Through polymorphism, the system can accommodate these new types seamlessly, ensuring that existing code remains unchanged while still supporting extended behaviour.

3.2 Reliability

The following descriptions assume that reliability refers to the system's ability to operate consistently and maintain functionality even under abnormal or failing conditions.

The Observer design pattern ensures that there is reliable communication between plants and their assigned staff. When a plant's condition changes, only the relevant observer is notified, preventing unnecessary updates and reducing risk of inconsistent states.

The **Fạcade** design pattern enhances reliability by centralising error handling within the Game class, allowing for controlled exception management and consistent response handling across the API points.

The *Logger* subsystem contributes to the system reliability by maintaining persistent records of all key actions. Log data is written to a file, ensuring traceability and assistance in debugging throughout the system.

3.3 Performance

The following descriptions assume that performance refers to the efficiency with how the system handles operations such as plant updates, transactions and logging, while maintaining responsiveness during gameplay.

Each *Plant* maintains and updates its own lifetime concurrently. In other words, every *Plant* instance is responsible for tracking its own internal state over time. This concurrent implementation eliminates the need to halt the system and iterate through the entire inventory to update plant lifetimes, an operation with O(n) time complexity by allowing each plant to manage its own updates Independently.

The **Observer** design pattern optimises performance by using an event driven approach, updates are triggered only when required, avoiding continuous pulling of plant conditions.

The **Decorator** design pattern ensures that the logging of operations remains lightweight. File I/O is buffered to prevent delays, and console output operations are performed in constant time. This guarantees that performance remains stable even as the number of plants, transactions and customers increases.

3.4 Usability

It is assumed that usability refers to the quality of the user experience from the perspective of a user who interacts with the system at a developmental or code-based level. In this context, the “user” is not an end-user of a graphical interface, but rather a developer or technical entity engaging directly with the system’s underlying implementation.

All API endpoints are centrally defined within the *Game* class, which serves as the Fącade participant in the **Fącade** design pattern. This design provides a single, unified interface through which all API routines are exposed, allowing users to easily identify and understand the purpose of each endpoint.

The use of consistent design patterns (e.g. **Factory Method**, **Strategy**, and **Builder**) ensures that there is consistency within the system's behavior.

4. Nursery Management Research

Section 5.4.1 from the specification.

Research into real-world nursery practices, plant development, and customer interaction shaped the design of the nursery management simulation. This foundation guided the creation of components that replicate the structure and challenges of running a greenhouse. Each system module, plants, staff, and customers, was modeled on authentic nursery operations, with design patterns chosen to maintain flexibility, clarity, and realism.

4.1 Research Brief

4.1.1 Plant Categorization and Life Cycles

Nurseries typically manage various plant types such as succulents, flowers, and trees, each with distinct care needs. Succulents are low-maintenance and drought-tolerant, flowers demand consistent watering and sunlight, and trees require long-term care. These findings influenced the use of the **Factory Method** and **Prototype** patterns to streamline the *creation* of plant objects (FlowerCreator, SucculentCreator, TreeCreator) and enable efficient cloning of plants, mirroring bulk seed purchasing.

4.1.2 Growth and Maintenance Simulation

Research highlighted that nursery plants transition through multiple growth phases before reaching a sellable maturity. This inspired the **State Pattern**, where each plant transitions between *NotSellable* and *Sellable* states, each encapsulating specific behaviors related to care and market readiness.

4.1.3 Staff and Plant Interaction

In a real nursery, horticultural staff monitor plant health and apply the appropriate care routines. This dynamic was modeled using the **Observer Pattern**, allowing plants (*subjects*) to notify corresponding staff (*observers*) when care is required. As part of our gamification feature, we leave it up to the player to queue care commands for plants (using interactive GUI elements), the staff will see to the plants in the order that the commands were queued.

4.1.4 Customer Behavior and Sales Dynamics

Customer research showed varying levels of horticultural knowledge, influencing buying habits. Three main profiles emerged: *Ignorant*, *Average*, and *Greenfinger* (expert) customers. The **Builder Pattern** was used for flexible customer creation, encapsulating configuration logic for different personalities and preferences.

4.2 Assumptions

The following assumptions were made during the design and development of the nursery management system:

4.2.1 Controlled Environment

It is assumed that all plants are managed within a controlled greenhouse environment where external environment factors such as temperature, humidity and light are stable and can be controlled consistently.

4.2.2 Simplified Care Routines

The care routines for watering, fertilizing, and pruning are simplified representations of real-life plant care. While base biological principles are abstracted to maintain a playable and efficient system.

4.2.3 Predictable Customer Behavior

The customer behavior is assumed to be predictable based on typical nursery interactions rather than being in individual personality types. This assumption allows for predictable, and reliable customer behaviour that is easy for us to simulate the outcomes.

4.2.4 Pattern based system design

All the system components are assumed to be modular and capable of extension through the use of design patterns. This ensures that new features can be incorporated without structural redesign.

4.2.5 Stable system resources

It is assumed that there are sufficient computational resources available to support concurrent plant lifecycles, transaction management, and logging without performance degradation.

4.3 References

Royal Horticultural Society (RHS) - Plant care & general advice:
https://www.rhs.org.uk/advice

U.S. Department of Agriculture (USDA). Nursery Management: Best Practices:
https://www.fs.usda.gov/nsl/Wpsm%202008/Chapter%207.pdf

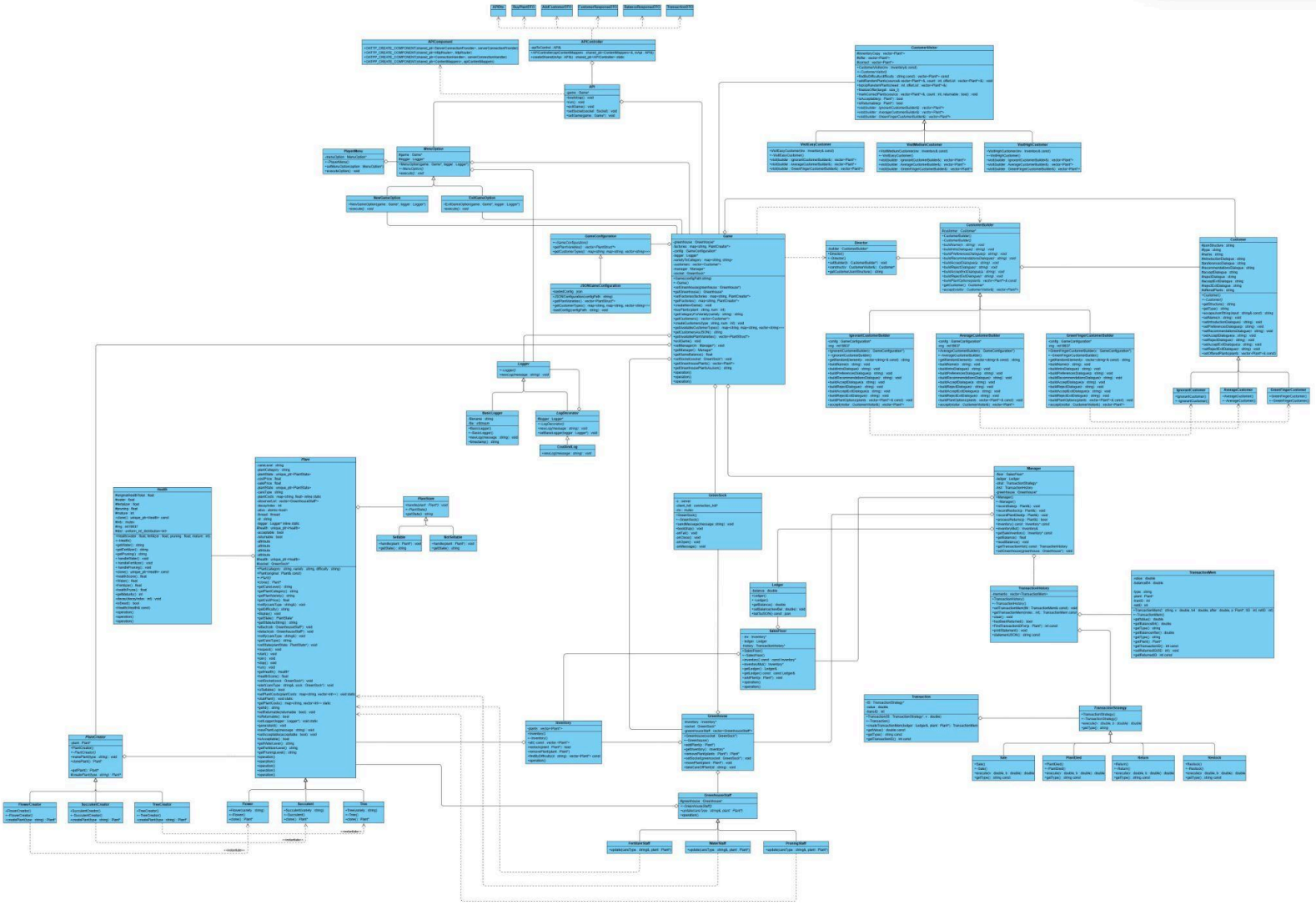
Better Homes & Gardens (BHG) – Plant encyclopedia / plant-care database:
https://www.bhg.com/gardening/plant-dictionary/

An academic review on greenhouse-management technology:
https://www.sciencedirect.com/science/article/pii/S1364032121005384

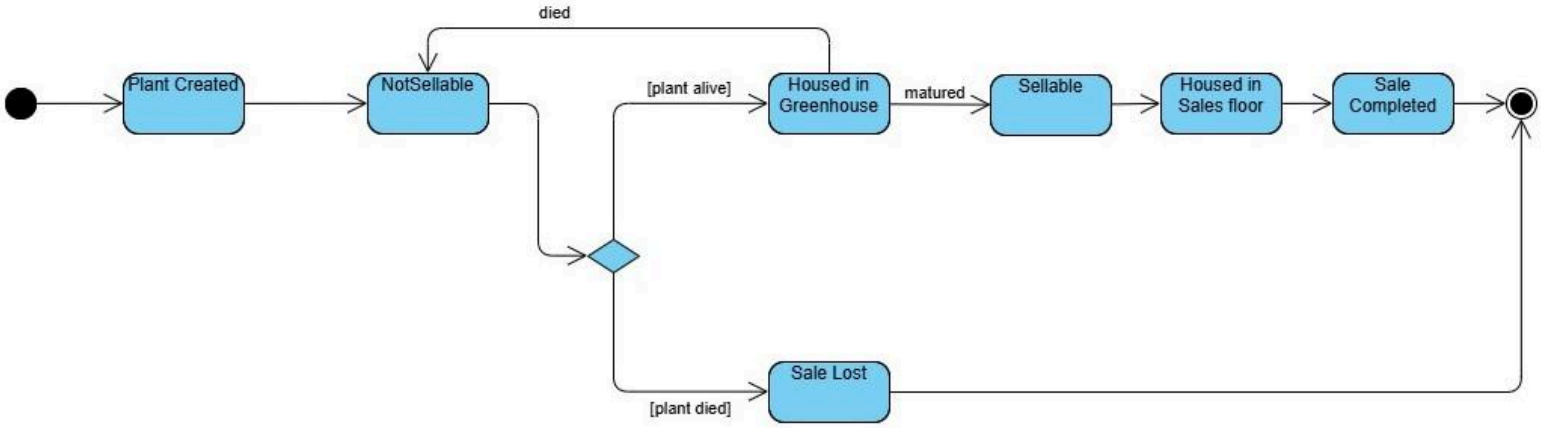
5. UML Diagrams

Section 5.4.3 from the specification

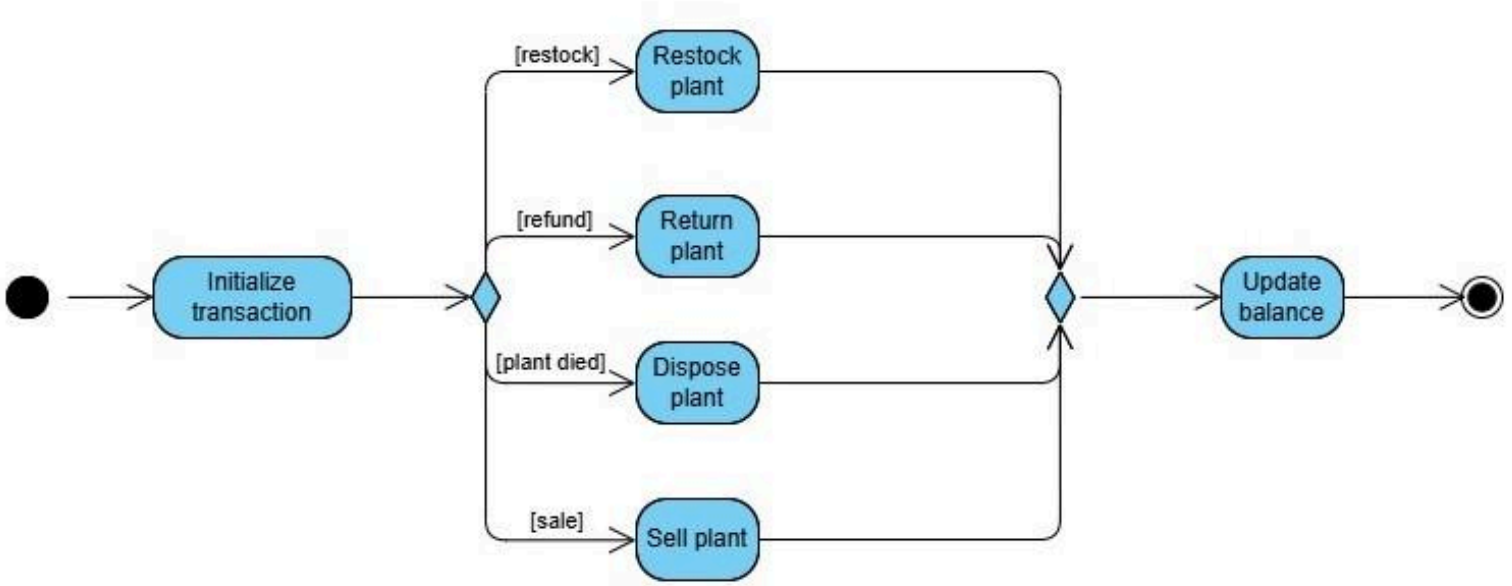
5.1 Class Diagram



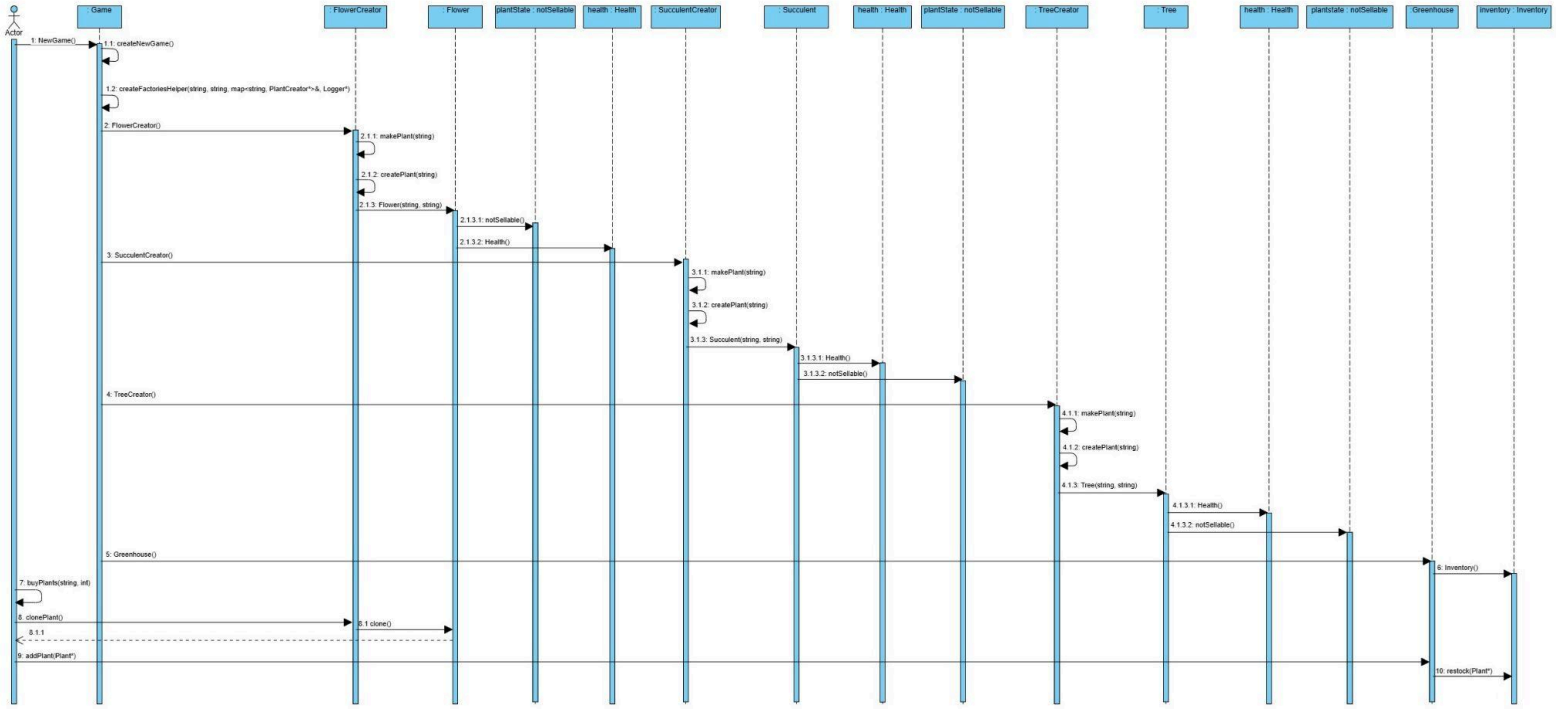
5.2 State Diagram



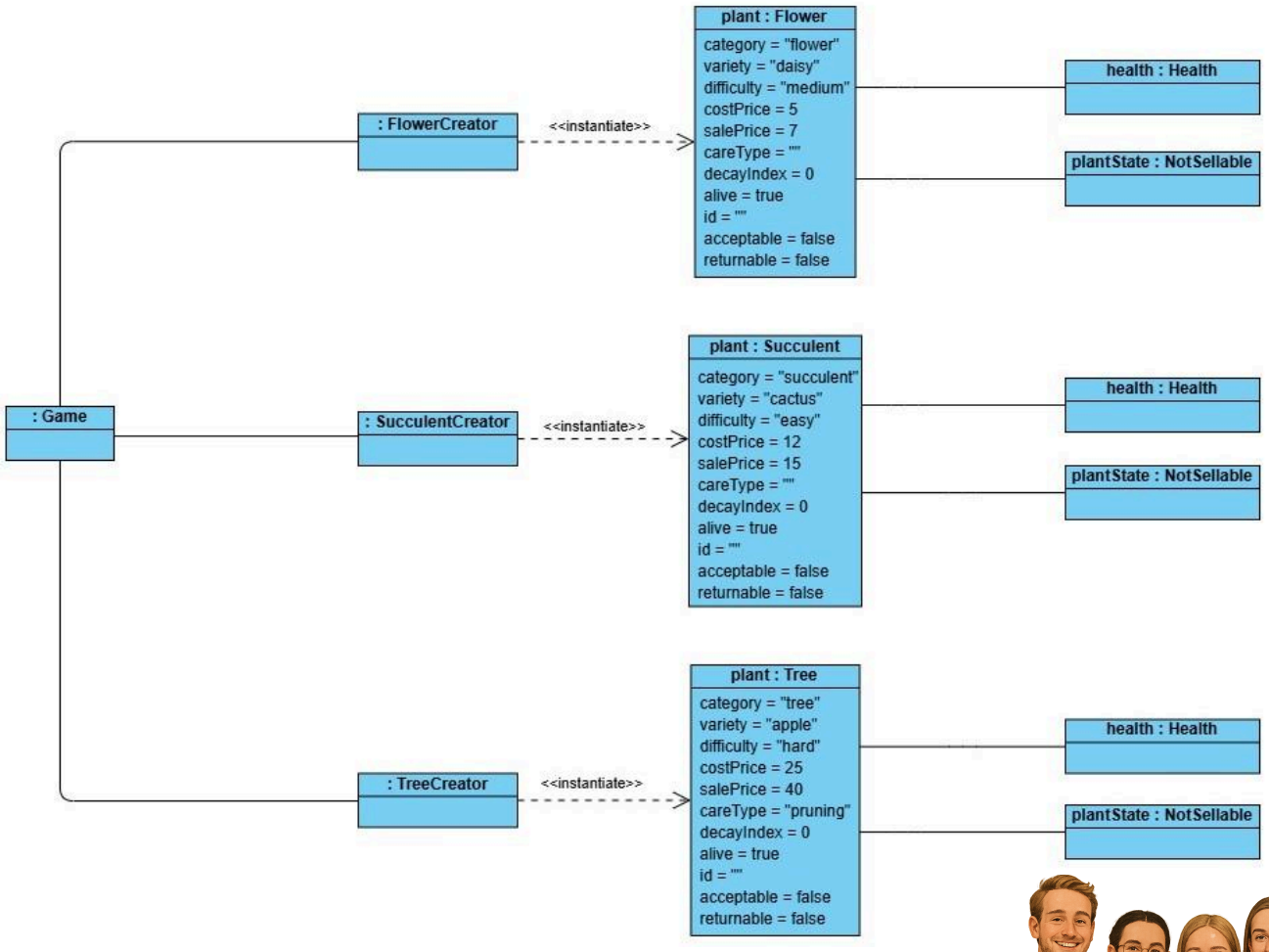
5.3 Activity Diagram



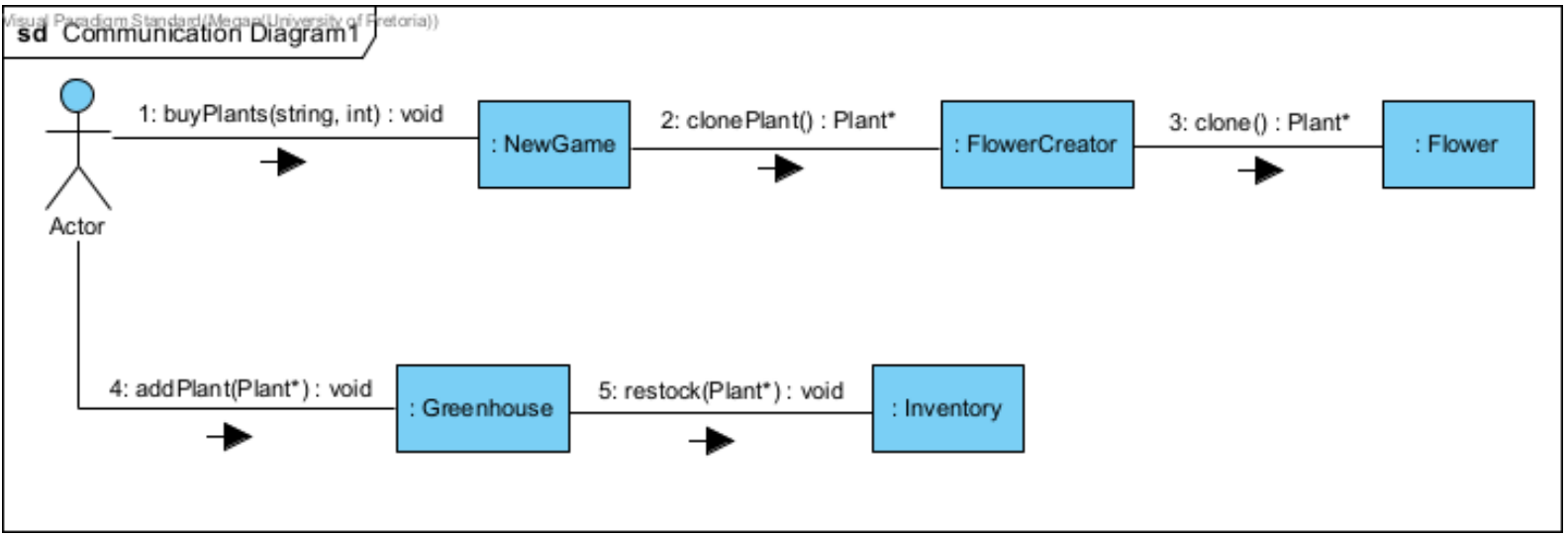
5.4 Sequence Diagram



5.5 Object Diagram



5.6 Communication Diagram



5.7 High-Level Activity Diagram