Templanter: A Plant Nursery Simulator

COS 214 Final Project Report
Team Templation

- Jo Reardon – u24597652
- Paul Hofmeyr – u24618391
- Finnley Wyllie – u24754120
- Mutombo Kabau – u24957102
- Alessandro Paravano – u24713122

University of Pretoria

- November 2025
- Google Doc Link:

https://docs.google.com/document/d/1rdOeh1aZ57GRGRswDw6wNYUqwda3ksXJLmDQzdqiFbQ/edit?usp=sharing

1. Project Overview

Templanter is a software simulation of a plant nursery, built in C++. The system models the core operations of a greenhouse, including plant growth, staff management, and customer interactions. The primary objective was to design a maintainable and scalable system by applying object oriented principles and design patterns to solve specific architectural problems. The simulation is structured around three main subsystems: Greenhouse, Staff, and Customers.

2. Research Brief

Effective nursery management requires coordination between plant care, inventory tracking, and customer service. Our research into real world nurseries and games - such as Stardew Valley - highlighted the need for systems that can manage different plant lifecycles, employee schedules, and customer preferences. This informed our design to prioritize flexibility and scalability. This is achieved by providing the ability to easily specialise or generalise parts of the system. We simplified real world problems, such as pest control and weather, in order to focus on the core challenge of creating a cohesive system.

3. System Architecture

The system is divided into three subsystems that communicate through well defined interfaces.

Greenhouse Subsystem: Manages all plant related logic. This includes plant creation, growth cycles, health status, and inventory tracking. It is the core of the simulation.

Staff Subsystem: Handles worker entities. Staff are assigned tasks such as watering, fertilizing, and harvesting plants, and assisting customers.

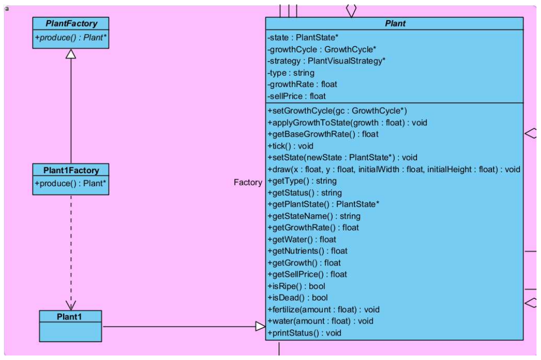
Customer Subsystem: Manages customer entities. Customers can browse available plants, interact with staff, and make purchases.

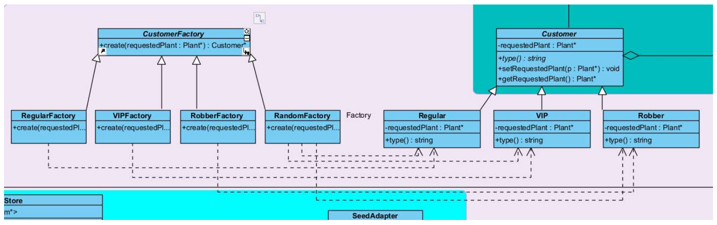
4. Design Patterns Application

We implemented 11 design patterns to address specific challenges in the system's design. The five required patterns from the project specification are indicated.

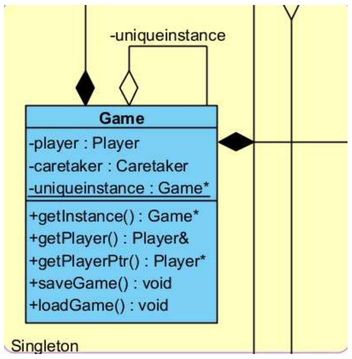
4.1 Creational Patterns

Factory Pattern: Used to create different types of plants and customers. The PlantFactory generates specific plant objects like Tomato or Sunflower, while the CustomerFactory creates different customer types such as Regular, VIP, and Robber.



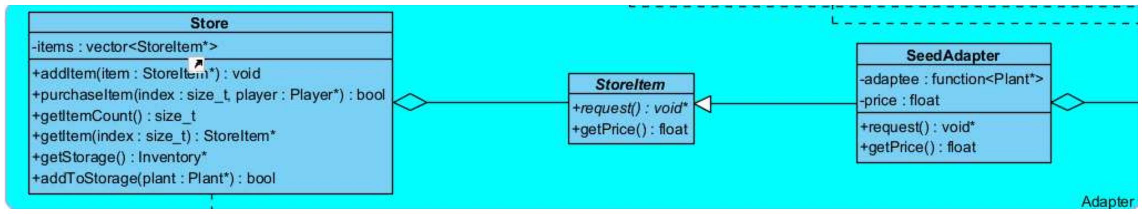


Singleton Pattern: Ensures a single instance of the core Game class exists. This provides a global point of access for the game state, inventory, and greenhouse, preventing inconsistencies.

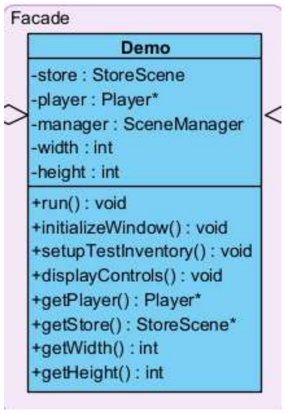


4.2 Structural Patterns

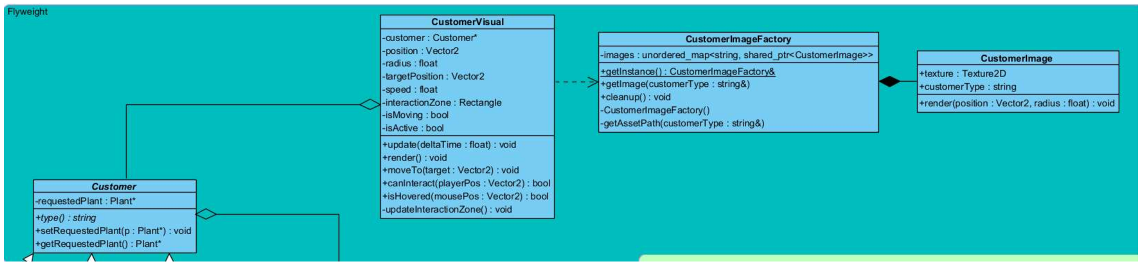
Adapter Pattern: Converts the Plant interface to the StoreItem interface. This allows plant objects to be displayed and sold on the sales floor without modifying the original Plant class.


Store
-items : vector<StoreItem*>
+addItem(item : StoreItem*) : void
+purchaseItem(index : size_t, player : Player*) : bool
+getItemCount() : size_t
+getItem(index : size_t) : StoreItem*
+getStorage() : Inventory*
+addToStorage(plant : Plant*) : bool
StoreItem
+request() : void*
+getPrice() : float
SeedAdapter
-adaptee : function<Plant*>
-price : float
+request() : void*
+getPrice() : float
Adapter

Facade Pattern: Provides a simplified interface, the Demo class, to the complex subsystems of the game loop. This makes the system easier to use and reduces dependencies.

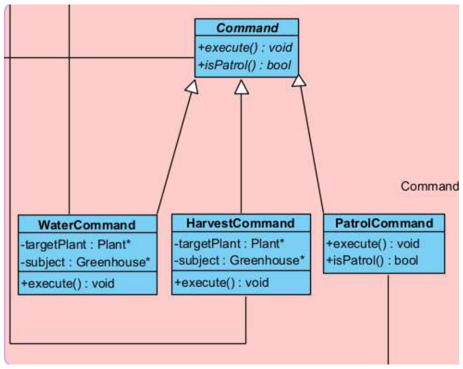


Flyweight Pattern: Shares common, immutable data like customer images across multiple objects. This significantly reduces the memory footprint when handling many customers.

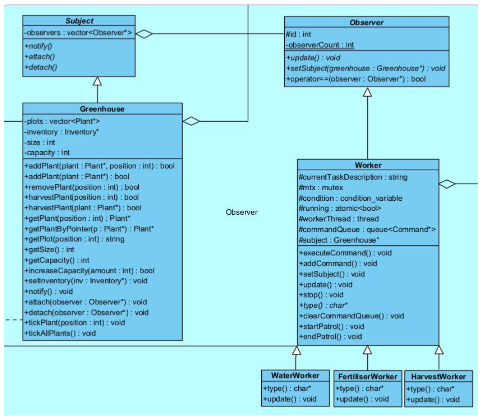

Flyweight
CustomerVisual
-customer : Customer*
-position : Vector2
-radius : float
-targetPosition : Vector2
-speed : float
-interactionZone : Rectangle
-isMoving : bool
-isActive : bool
+update(deltaTime : float) : void
+render() : void
+moveTo(target : Vector2) : void
+canInteract(playerPos : Vector2) : bool
+isHovered(mousePos : Vector2) : bool
-updateInteractionZone() : void
CustomerImageFactory
-images : unordered_map<string, shared_ptr<CustomerImage>>
+getInstance() : CustomerImageFactory&
+getImage(customerType : string&)
+cleanup() : void
-CustomerImageFactory()
-getAssetPath(customerType : string&)
CustomerImage
+texture : Texture2D
+customerType : string
+render(position : Vector2, radius : float) : void
Customer
-requestedPlant : Plant*
+type() : string
+setRequestedPlant(p : Plant*) : void
+getRequestedPlant() : Plant*

4.3 Behavioral Patterns

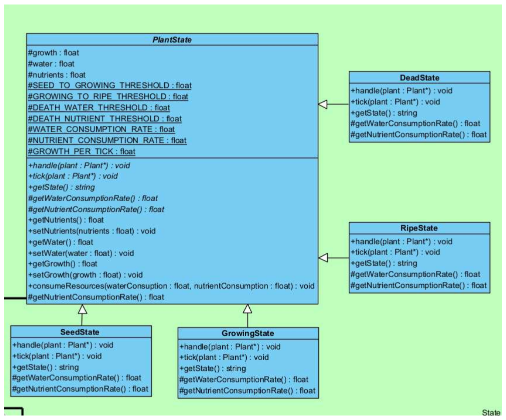
Command Pattern: Encapsulates actions like watering or harvesting as objects. These command objects are passed between workers and plants, allowing for flexible task execution and queuing.



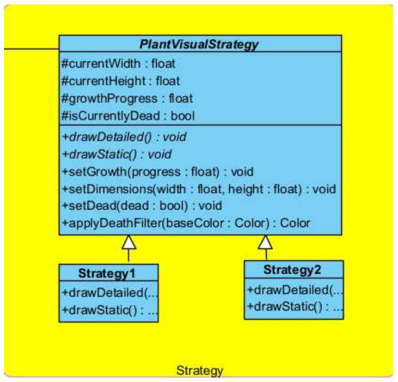
Observer Pattern: Allows the Greenhouse to notify Workers when plants need care. This creates a one to many dependency so that multiple workers can be alerted automatically without the plants having direct references to them.


Subject
-observers : vector<Observer*>
+notify()
+attach()
+detach()
Greenhouse
-plots : vector<Plant*>
-inventory : Inventory*
-size : int
-capacity : int
+addPlant(plant : Plant*, position : int) : bool
+addPlant(plant : Plant*) : bool
+removePlant(position : int) : bool
+harvestPlant(position : int) : bool
+harvestPlant(plant : Plant*) : bool
+getPlant(position : int) : Plant*
+getPlantByPointer(p : Plant*) : Plant*
+getPlot(position : int) : string
+getSize() : int
+getCapacity() : int
+increaseCapacity(amount : int) : bool
+setInventory(inv : Inventory*) : void
+notify() : void
+attach(observer : Observer*) : void
+detach(observer : Observer*) : void
+tickPlant(position : int) : void
+tickAllPlants() : void
Observer
Observer
#id : int
-observerCount : int
+update() : void
+setSubject(greenhouse : Greenhouse*) : void
+operator==(observer : Observer*) : bool
Worker
#currentTaskDescription : string
#mtx : mutex
#condition : condition_variable
#running : atomic<bool>
#workerThread : thread
#commandQueue : queue<Command*>
#subject : Greenhouse*
+executeCommand() : void
+addCommand() : void
+setSubject() : void
+update() : void
+stop() : void
+type() : char*
+clearCommandQueue() : void
+startPatrol() : void
+endPatrol() : void
WaterWorker
+type() : char*
+update() : void
FertiliserWorker
+type() : char*
+update() : void
HarvestWorker
+type() : char*
+update() : void

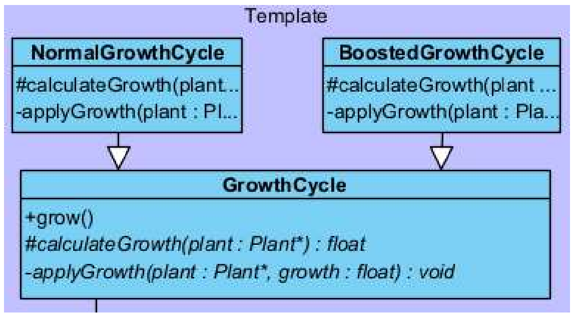
State Pattern: Manages a plant's behavior through its lifecycle. A plant can be in different states such as Seed, Growing, Ripe, or Decaying, with each state defining its specific behavior.


PlantState
#growth : float
#water : float
#nutrients : float
#SEED_TO_GROWING_THRESHOLD : float
#GROWING_TO_RIPE_THRESHOLD : float
#DEATH_WATER_THRESHOLD : float
#DEATH_NUTRIENT_THRESHOLD : float
#WATER_CONSUMPTION_RATE : float
#NUTRIENT_CONSUMPTION_RATE : float
#GROWTH_PER_TICK : float
+handle(plant : Plant*) : void
+tick(plant : Plant*) : void
+getState() : string
#getWaterConsumptionRate() : float
#getNutrientConsumptionRate() : float
+getNutrients() : float
+setNutrients(nutrients : float) : void
+getWater() : float
+setWater(water : float) : void
+getGrowth() : float
+setGrowth(growth : float) : void
+consumeResources(waterConsuption : float, nutrientConsumption : float) : void
#getNutrientConsumptionRate() : float
DeadState
+handle(plant : Plant*) : void
+tick(plant : Plant*) : void
+getState() : string
#getWaterConsumptionRate() : float
#getNutrientConsumptionRate() : float
RipeState
+handle(plant : Plant*) : void
+tick(plant : Plant*) : void
+getState() : string
#getWaterConsumptionRate() : float
#getNutrientConsumptionRate() : float
SeedState
+handle(plant : Plant*) : void
+tick(plant : Plant*) : void
+getState() : string
#getWaterConsumptionRate() : float
#getNutrientConsumptionRate() : float
GrowingState
+handle(plant : Plant*) : void
+tick(plant : Plant*) : void
+getState() : string
#getWaterConsumptionRate() : float
#getNutrientConsumptionRate() : float
State

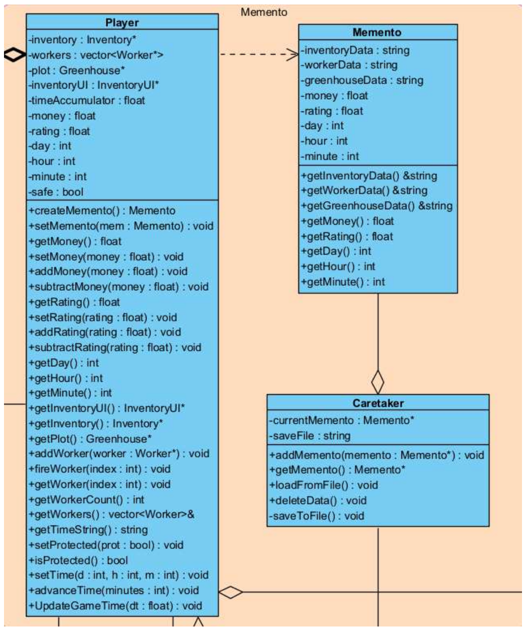
Strategy Pattern: Defines a family of interchangeable algorithms for rendering plants. Different visual strategies can be applied to a plant without changing its core class.



Template Method Pattern: Defines the skeleton of the plant growth algorithm in a base class. Subclasses can override specific steps, allowing different plants to have unique growth cycles while following a common structure.


Template
NormalGrowthCycle
#calculateGrowth(plant...
-applyGrowth(plant : Pl...
BoostedGrowthCycle
#calculateGrowth(plant ...
-applyGrowth(plant : Pla...
GrowthCycle
+grow()
#calculateGrowth(plant : Plant*) : float
-applyGrowth(plant : Plant*, growth : float) : void

Memento Pattern: Captures and externalizes the game's internal state. This enables save and load functionality, allowing the simulation state to be restored for day night cycles or different sessions.


Memento
Player
-inventory : Inventory*
-workers : vector<Worker*>
-plot : Greenhouse*
-inventoryUI : InventoryUI*
-timeAccumulator : float
-money : float
-rating : float
-day : int
-hour : int
-minute : int
-safe : bool
+createMemento() : Memento
+setMemento(mem : Memento) : void
+getMoney() : float
+setMoney(money : float) : void
+addMoney(money : float) : void
+subtractMoney(money : float) : void
+getRating() : float
+setRating(rating : float) : void
+addRating(rating : float) : void
+subtractRating(rating : float) : void
+getDay() : int
+getHour() : int
+getMinute() : int
+getInventoryUI() : InventoryUI*
+getInventory() : Inventory*
+getPlot() : Greenhouse*
+addWorker(worker : Worker*) : void
+fireWorker(index : int) : void
+getWorker(index : int) : void
+getWorkerCount() : int
+getWorkers() : vector<Worker>&
+getTimeString() : string
+setProtected(prot : bool) : void
+isProtected() : bool
+setTime(d : int, h : int, m : int) : void
+advanceTime(minutes : int) : void
+UpdateGameTime(dt : float) : void
Memento
-inventoryData : string
-workerData : string
-greenhouseData : string
-money : float
-rating : float
-day : int
-hour : int
-minute : int
+getInventoryData() &string
+getWorkerData() &string
+getGreenhouseData() &string
+getMoney() : float
+getRating() : float
+getDay() : int
+getHour() : int
+getMinute() : int
Caretaker
-currentMemento : Memento*
-saveFile : string
+addMemento(memento : Memento*) : void
+getMemento() : Memento*
+loadFromFile() : void
+deleteData() : void
-saveToFile() : void

5. Task 2:

5.1 Functional and Non-Functional

Functional
Design Patterns:

1. Singleton:
FR1: The system uses singleton design pattern to ensure the Game class ensures single instance of game state.

2. Factory:
FR1: The system uses factory design pattern. PlantFactory and CustomerFactory create specific object types.

3. Adapter:
FR1: The system uses adapter design pattern to convert Plant interface to StoreItem interface for sales.

4. Facade:
FR1: The system uses facade design pattern. Demo class simplifies access to complex subsystems.

5. Flyweight:
FR1: The system uses flyweight design pattern to share immutable customer data across multiple objects.

6. Command:
FR1: The system uses command design pattern to encapsulate worker tasks like watering and harvesting.

7. Observer:
FR1: The system uses observer design pattern. Greenhouse notifies workers when plants need care.

8. State:
FR1: The system uses state design pattern to manage plant lifecycle (Seed, Growing, Ripe, Decaying).

9. Strategy:
FR1: The system uses strategy design pattern to define interchangeable rendering algorithms for plants.

10. Template Method:
FR1: The system uses template method design pattern to define plant growth algorithm skeleton for subclass specialization.

11. Memento:
FR1: The system uses memento design pattern to enable save/load functionality for game state restoration.

Non-Functional

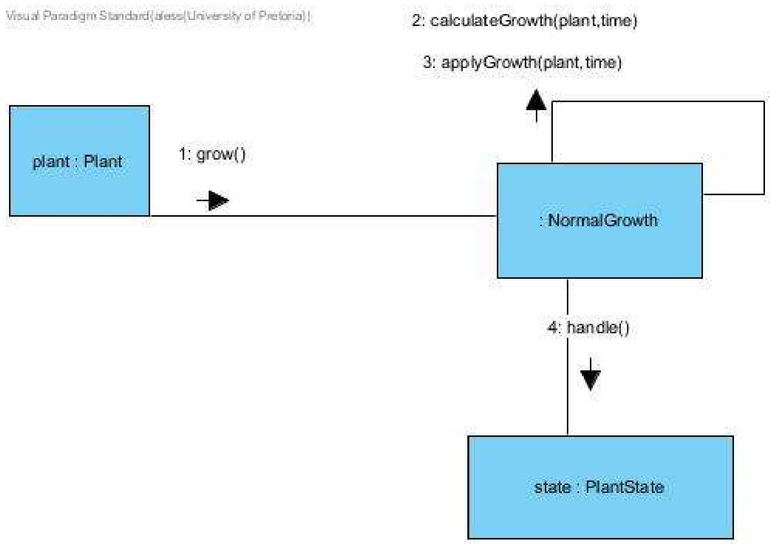
1. Usability:
The system's interface (raylib-based GUI) will allow for the Player to easily manage their greenhouse, workers, inventory, customers and take care of their plants.

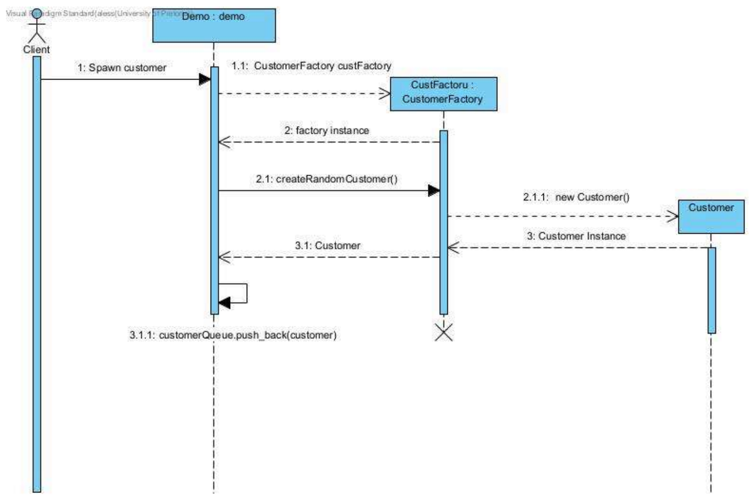
2. Scalability:
The system shall support the up to 1000 plant instances across all farm plots and inventory without degradation in response time for core operations.

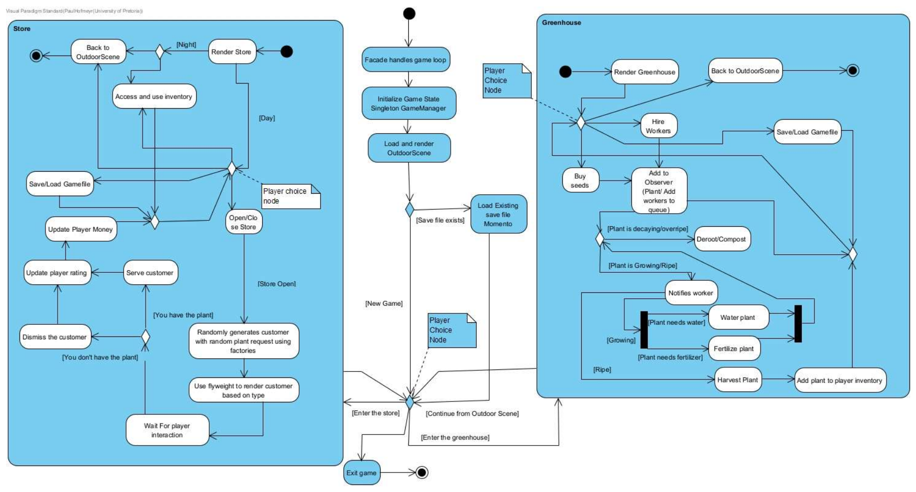
3. Maintainability:
The system should support the easy maintainability of several aspects of the design. It allows for the easy creation of more types of plants or customers. System provides a maintainable and easy way to create additions to the types of growth, commands and the plant states – in case of additional implementation.

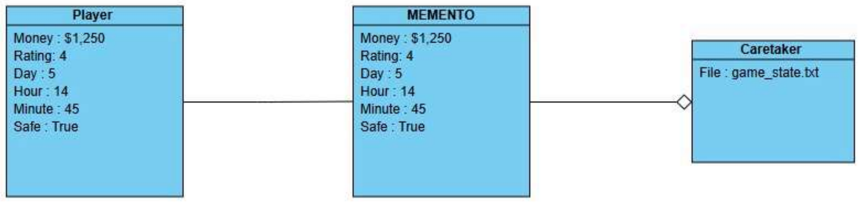
4. Thread Safety:
The system shall ensure thread-safe operations for all worker command execution and state transitions. Worker threads will safely queue and execute commands without race conditions or deadlocks. All shared data structures will be protected by mutexes to prevent concurrent access issues and ensure game stability even with multiple workers operating simultaneously.

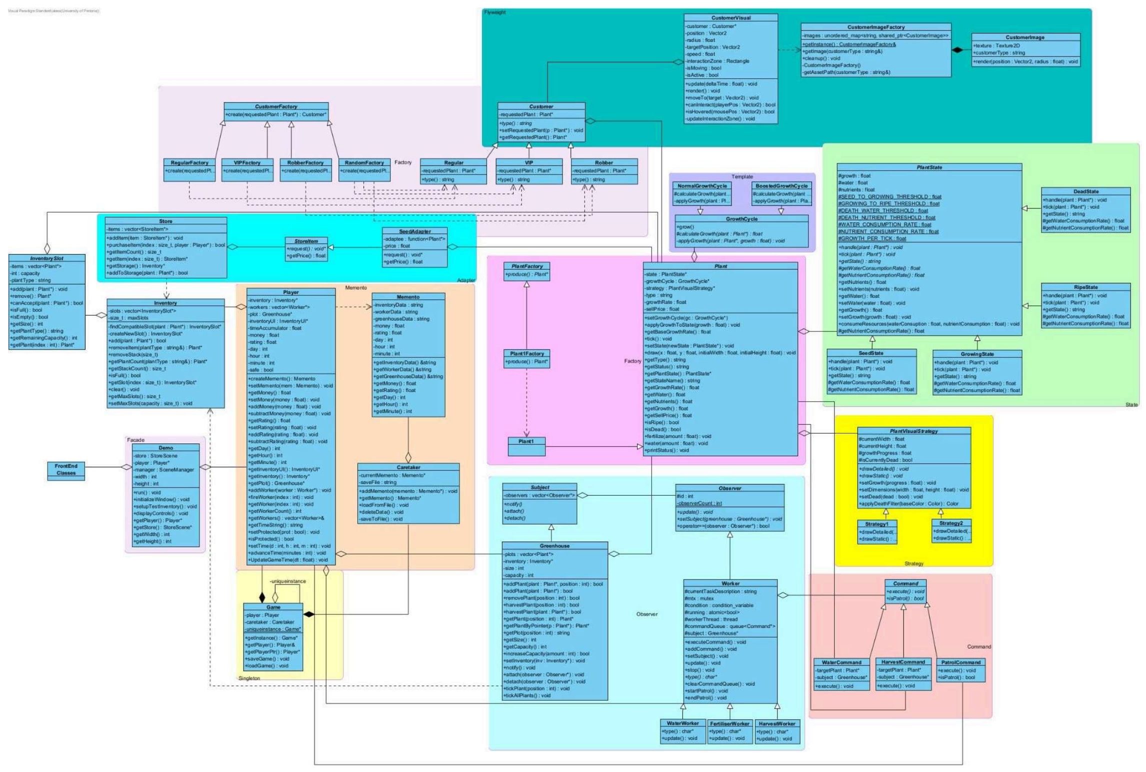
5.2 UMLs


Visual Paradigm Standard(aless(University of Pretoria))
2: calculateGrowth(plant,time)
3: applyGrowth(plant,time)
plant : Plant
1: grow()
: NormalGrowth
4: handle()
state : PlantState


Visual Paradigm Standard(aless(University of Pretoria))
Client
Demo : demo
1: Spawn customer
1.1: CustomerFactory custFactory
CustFactoru : CustomerFactory
2: factory instance
2.1: createRandomCustomer()
2.1.1: new Customer()
Customer
3: Customer Instance
3.1: Customer
3.1.1: customerQueue.push_back(customer)


Visual Paradigm Standard(PaulHofmeyr(University of Pretoria))
Store
Back to OutdoorScene
[Night]
Render Store
Access and use inventory
[Day]
Save/Load Gamefile
Player choice node
Update Player Money
Open/Close Store
Update player rating
Serve customer
[Store Open]
[You have the plant]
Dismiss the customer
[You don't have the plant]
Randomly generates customer with random plant request using factories
Use flyweight to render customer based on type
Wait For player interaction
Facade handles game loop
Initialize Game State Singleton GameManager
Load and render OutdoorScene
[Save file exists]
Load Existing save file Momento
[New Game]
Player Choice Node
Player Choice Node
[Enter the store]
[Continue from Outdoor Scene]
[Enter the greenhouse]
Exit game
Greenhouse
Render Greenhouse
Back to OutdoorScene
Hire Workers
Save/Load Gamefile
Buy seeds
Add to Observer (Plant/ Add workers to queue)
[Plant is decaying/overripe]
Deroot/Compost
[Plant is Growing/Ripe]
Notifies worker
[Plant needs water]
Water plant
[Growing]
Fertilize plant
[Plant needs fertilizer]
[Ripe]
Harvest Plant
Add plant to player inventory


Player
Money : $1,250
Rating: 4
Day : 5
Hour : 14
Minute : 45
Safe : True
MEMENTO
Money : $1,250
Rating: 4
Day : 5
Hour : 14
Minute : 45
Safe : True
Caretaker
File : game_state.txt


Flyweight
CustomerVisual
CustomerImageFactory
CustomerImage
Customer
CustomerFactory
RegularFactory
VIPFactory
RobberFactory
RandomFactory
Factory
Regular
VIP
Robber
Template
NormalGrowthCycle
BoostedGrowthCycle
GrowthCycle
PlantState
DeadState
RipeState
SeedState
GrowingState
State
Store
StoreItem
SeedAdapter
Adapter
InventorySlot
Inventory
Player
Memento
Caretaker
PlantFactory
Plant1Factory
Plant1
Plant
Facade
Demo
FrontEnd Classes
PlantVisualStrategy
Strategy1
Strategy2
Strategy
Subject
Observer
Greenhouse
Worker
WaterWorker
FertiliserWorker
HarvestWorker
Command
WaterCommand
HarvestCommand
PatrolCommand
-uniqueInstance
Game
Singleton

6. Conclusion

The Templanter project successfully meets the requirements for the COS 214 final project. We developed a functional plant nursery simulator that effectively models greenhouse operations, staff activities, and customer interactions. The application of 11 design patterns, including all required ones, resulted in a codebase that is flexible, maintainable, and demonstrates a strong understanding of software design principles. The project provided valuable experience in object oriented design, team collaboration, and professional development practices.

References

1. Acovino, V. and Summers, J. (2025) *The legacy - and future - of the farming game Stardew Valley*, *NPR*. Available at: https://www.npr.org/2025/01/24/g-s1-44510/the-legacy-and-future-of-the-farming-game-stardew-valley (Accessed: 20 October 2025).

2. FarmERP (2025) *Advanced Nursery Management software to boost production and profitability*, *FarmERP*. Available at: https://www.farmerp.com/blog/advanced-nursery-management-software-to-boost-production-and-profitability/ (Accessed: 21 October 2025).

3. (No date) *Nursery management - an overview | sciencedirect topics*. Available at: https://www.sciencedirect.com/topics/agricultural-and-biological-sciences/nursery-management (Accessed: 21 October 2025).

4. (2017) © *July 2020| IJIRT | Volume 7 issue 2 | ISSN: 2349-6002 IJIRT 150053*. Available at: https://ijirt.org/publishedpaper/IJIRT150053_PAPER.pdf (Accessed: 20 October 2025).

5.*Development of Web Based Plant Nursery Management* (no date) *ISJEM Journal*. Available at: https://isjem.com/download/development-of-web-based-plant-nursery-management/ (Accessed: 04 November 2025).