University of Pretoria

Department of Computer Science

Faculty of Engineering, Built Environment & IT

COS214
Plant Nursery Simulator

Final Project Report

PHOTOSYNTECH

Team Members:

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

Submission Date: November 2025

https://github.com/marcelstoltz00/Photosyntech

Contents

1 Research Brief

1.1 Overview

We looked into key areas of the plant care and nursery industry, like staff roles, business size, seasonal changes, plant varieties and their needs, and stock tracking.

1.2 Plant Types and Care Needs

We picked four main plant types(trees, succulents, shrubs, herbs) as basics that can be combined with add-ons to cover most plants. For instance, a rose starts as a shrub with extras like thorns and flowers.

- Care involves watering by groups, slow-release fertilizing, regular pruning, and proper spacing.
- Monitoring uses sensors for temperature, humidity, light, CO2, pH, and soil moisture.
- Plant stages include starting from cuttings or seeds, growing, toughening up before sale, quality checks, and selling.
- Adjust for seasons by prioritising hardy plants.
- Account for losses from diseases or damage at different stages.

1.3 Staff Roles and Customer Service

Staff handle various tasks:

- Managers oversee operations with walkthroughs and dashboards.
- Supervisors manage climate and daily tasks.
- Horticulturists diagnose issues and record data.
- Propagation teams handle cuttings and batch tracking.
- Technicians maintain equipment and update on the go.
- Inventory staff count and update stock.
- Sales team advises on prices and consults.
- Shipping picks and stages orders.

Customer interactions include assessing needs, giving advice, handling purchases with instructions, processing payments, offering guarantees, and managing loyalty programs.

1.4 Nursery Operations

- Manage many plant varieties at once.
- Prices change with seasons based on demand.
- Source stock as young plants for growing or ready ones for fast sales.
- Use first-in-first-out rotation and real-time tracking with barcodes or RFID.
- Track and adjust for losses at each stage.

1.5 Assumptions for simplicity

- Basic plant stages: seed, vegetative, mature, dead.
- Care focused on water and sunlight strategies.
- Medium-scale with thousands of plants max.
- Simulated environment without real sensors.
- Single-site inventory.

1.6 References

1. Types of Plants-Herbs, Shrubs, Trees, Climbers, and Creepers. BYJU'S Biology, 2025. Available at: https://byjus.com/biology/plants/.
2. Reducing Crop Shrinkage. Greenhouse Grower, 2009. Available at: https://www.greenhousegrower.com/production/plant-culture/reducing-crop-shrinkage/.
3. Top 10 Nursery Production Integrated Pest Management Practices in the Southeast. UGA Cooperative Extension Circular 1008, 2011. Available at: https://fieldreport.caes.uga.edu/publications/C1008/top-10-nursery-production-integrated-pest-management-practices-in-the-southeast/.
4. Shrink happens. Nursery Management, 2023. Available at: https://www.nurserymag.com/article/shrink-happens/.
5. Starting in the Nursery Business. Purdue Extension HO-212, 2001. Available at: https://www.extension.purdue.edu/extmedia/ho/ho-212.pdf.

2 Functional and Non-Functional Requirements

2.1 Functional Requirements

2.1.1 Overview

Functional requirements specify what the system must do. These define the features, functions, and behaviours the system must implement.

Subsystem: Greenhouse & Plant Management

- **FR-1: Plant Species Creation and Configuration**
  Description: The system shall provide the ability to create plant species (e.g., Rose, Sunflower) as combinations of base plant types with specific configurations including decorative attributes, watering needs, sunlight requirements, and initial maturity state.
  Acceptance Criteria:
  - Plant species can be created from base types (Succulent, Shrub, Tree, Herb) with decorative attributes
  - All required plant attributes are initialised through a construction process
  - Configuration options are validated
  - Complex plant compositions (base type + decorations) are supported
- **FR-2: Plant Type Cloning**
  Description: The system shall allow new plant instances to be created by copying the attributes of existing plant types (Succulent, Shrub, Tree, Herb) and their decorations.
  Acceptance Criteria:
  - Plant types can be cloned with identical attributes
  - Cloned plants maintain all properties of the original (decorations, strategies, states)
  - Multiple copies can be created from a single plant instance
  - Cloning preserves both base plant type and applied decorations
- **FR-3: Plant Lifecycle Management**
  Description: The system shall manage plant maturity states which specify minimum ages, growth rates, and price adjustments.
  Acceptance Criteria:
  - Plants transition automatically between maturity states
  - State transitions based on plant age
  - Each state has defined characteristics
- **FR-4: Watering Management**
  Description: The system shall implement different watering approaches that define the water amount and frequency for different types of plants.
  Acceptance Criteria:
  - Different plant types receive appropriate water amounts
  - Watering frequency varies by plant type
  - Watering approach can change based on plant lifecycle
  - Staff can execute watering actions

- **FR-5: Sunlight Management**
  Description: The system shall implement different sunlight exposure approaches that define the intensity and duration of sun exposure needed for different types of plants.
  Acceptance Criteria:
  - Different plant types receive appropriate sunlight
  - Sunlight intensity and duration are configurable
  - Sunlight requirements vary by plant type

Subsystem: Inventory Management

- **FR-6: Centralised Inventory**
  Description: The system shall maintain a single, shared inventory accessible to both customers and staff to ensure consistency.
  Acceptance Criteria:
  - Single inventory instance exists
  - Both staff and customers access the same data
  - No duplicate or conflicting inventory records
  - Inventory state is consistent across all users
- **FR-7: Hierarchical Plant organisation**
  Description: The system shall organise plants into hierarchical structures where plants can be individual items or groups of plants.
  Acceptance Criteria:
  - Plants can be grouped by categories
  - Individual plants and groups are handled uniformly
  - Operations can traverse the entire structure
  - Total plant count can be calculated across all levels
- **FR-8: Inventory Tracking and Notifications**
  Description: The system shall notify staff and customers of inventory changes including sales, new stock arrivals, and stock level changes.
  Acceptance Criteria:
  - Multiple staff can monitor the same inventory
  - All stakeholders maintain awareness of stock levels
- **FR-9: Seasonal Plant Filtering**
  Description: The system shall provide the ability to traverse and filter plants based on their assigned growing season.
  Acceptance Criteria:
  - Plants can be filtered by season
  - Traversal through plant collections is possible
  - Seasonal information is maintained for each plant
  - Users can view season-specific inventory

- **FR-10: Shared Data Memory Optimisation**
  Description: The system shall minimise memory usage by sharing immutable data objects (seasons, watering strategies, sunlight strategies) across multiple plant instances.
  Acceptance Criteria:
  - Immutable data objects are shared rather than duplicated
  - Memory footprint is reduced for large inventories
  - Shared objects include season names and care strategies
  - System maintains performance while reducing memory consumption

Subsystem: Sales Floor

- **FR-11: Plant Decoration**
  Description: The system shall allow plants to be customised with additions such as decorative pots, charms, or other embellishments.
  Acceptance Criteria:
  - Multiple customisations can be applied to a single plant
  - Customisations can be added incrementally
  - Customised plants maintain core plant functionality
  - Original plant properties remain accessible
- **FR-12: Staff-Customer Communication**
  Description: The system shall coordinate communication between staff and customers on the sales floor while keeping these entities independent.
  Acceptance Criteria:
  - Customers can request staff assistance
  - Staff can respond to customer inquiries
  - Communication is coordinated centrally
  - Staff and customers remain loosely coupled
- **FR-13: Plant Sales Transactions**
  Description: The system shall process plant sales transactions, updating inventory and recording purchase details.
  Acceptance Criteria:
  - Plants can be purchased by customers
  - Inventory automatically updated on sale
  - Transaction details are recorded
  - Sales affect inventory notifications

Subsystem: Staff Management

- **FR-14: Plant Monitoring by Staff**
  Description: The system shall allow staff members to monitor assigned plants and receive notifications about health changes and lifecycle transitions.
  Acceptance Criteria:
  - Staff can be assigned to plant group

  - Health status changes trigger alerts

- **FR-15: Staff Action Execution**
  Description: The system shall allow staff to execute actions such as watering plants, managing inventory, and performing care routines. Complex multi-step actions shall be encapsulated to support undo/redo functionality.
  Acceptance Criteria:
  - Staff can perform watering actions
  - Care routines can be executed
  - User interface actions are decoupled from business logic

- **FR-16: Staff Hiring and Termination**
  Description: The system shall support hiring new staff members and terminating existing staff through administrative actions.
  Acceptance Criteria:
  - New staff can be added to the system
  - Existing staff can be removed
  - Staff changes are recorded

Subsystem: System Interface

- **FR-17: Unified System Interface**
  Description: The system shall provide a simplified interface layer that handles all business logic and coordinates subsystem interactions, enabling both graphical and command-line user interfaces to interact with the system through simple function calls.
  Acceptance Criteria:
  - Complex subsystem interactions are handled internally
  - User interface can interact through simple function calls
  - Business logic is decoupled from presentation layer
  - Both TUI interface can be supported without code duplication
  - System operations are accessible through a single unified interface

Summary by Subsystem

- **Greenhouse & Plant Management**: FR-1 to FR-5
- **Inventory Management**: FR-6 to FR-10
- **Sales Floor**: FR-11 to FR-13
- **Staff Management**: FR-14 to FR-16
- **System Interface**: FR-17

2.2 Non-Functional Requirements

2.2.1 Overview

Non-functional requirements specify how well the system performs rather than what it does. They define quality attributes such as performance, maintainability, usability, scalability, and reliability.

- **NFR-1: Performance**
  Quality Attribute: Performance
  Description: The system must be able to execute all staff care routines for a simulation of up to 10,000 active plants within a single simulated day cycle to ensure smooth progression.
  Measurement Criteria:
  - All care routines complete within one day cycle
  - No noticeable lag or delays during simulation
  - Maximum plants supported: 100,000,000
- **NFR-2: Maintainability/Extensibility**
  Quality Attribute: Maintainability, Extensibility
  Description: 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.
  Measurement Criteria:
  - New plant species: ≤ 2 file modifications
  - New care routine: ≤ 2 file modifications
  - Changes are localised and do not ripple through the system
- **NFR-3: Usability**
  Quality Attribute: Usability
  Description: The system must utilize a clear, text-based or visual 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 interface prompts.
  Measurement Criteria:
  - All actions clearly labeled and accessible
  - Help/instructions available on request
  - No external documentation required for basic operation
  - First-time users can complete basic tasks without assistance
- **NFR-4: Scalability**
  Quality Attribute: Scalability, Memory Efficiency
  Description: The system must be able to handle a simulation with an inventory of up to 100,000,000 unique plant instances without memory exhaustion or a noticeable decrease in simulation speed. The system shall achieve this through efficient memory management including sharing of immutable data objects across plant instances.
  Measurement Criteria:
  - Maximum unique plant instances: 100,000,000
  - No memory exhaustion
  - No significant performance degradation
  - Simulation speed remains consistent

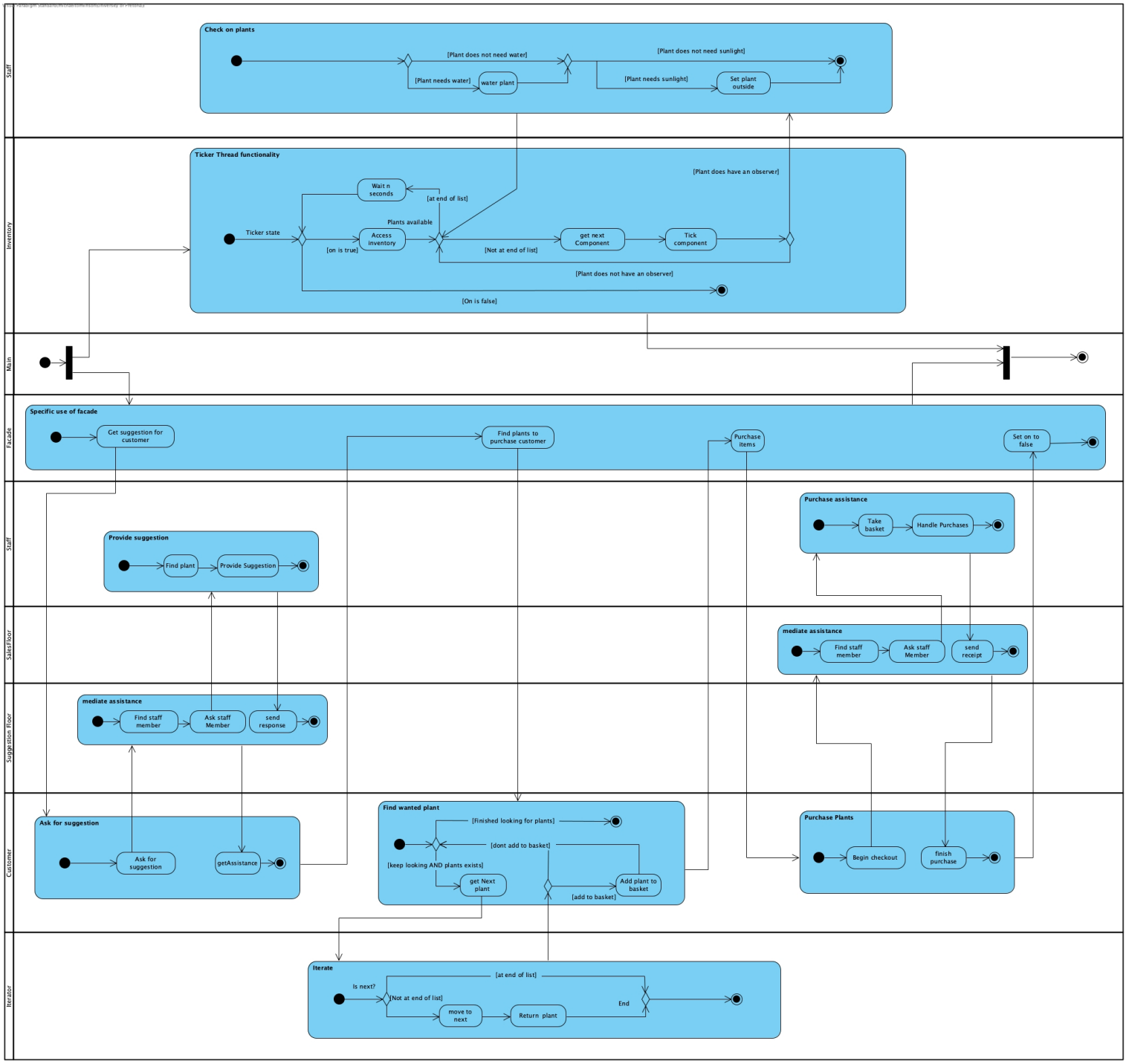
  - Shared immutable objects (strategies, seasons) are reused across instances
  - Memory footprint scales sub-linearly with number of plants

- **NFR-5: Reliability**
  Quality Attribute: Reliability, Modularity
  Description: 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.
  Measurement Criteria:

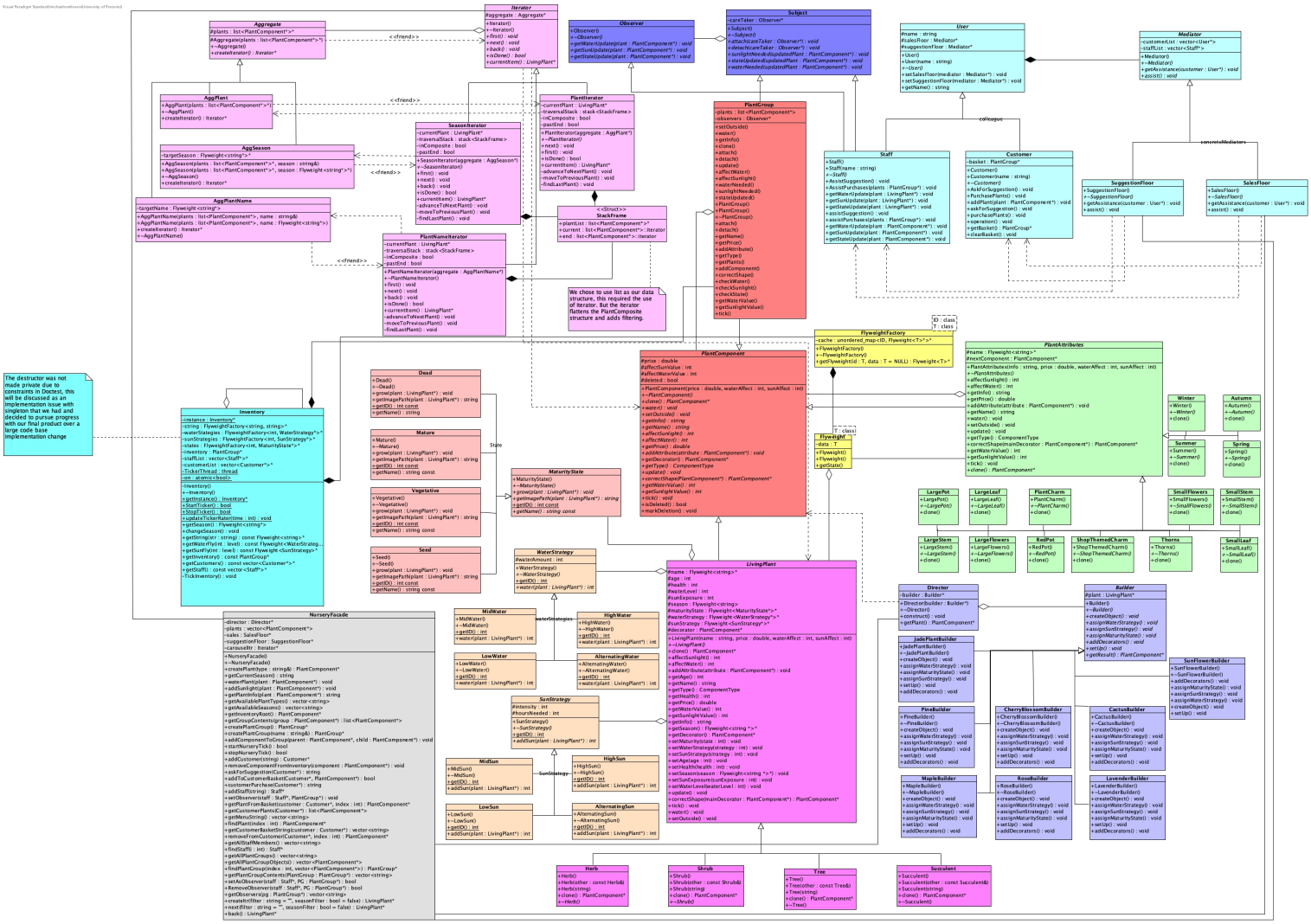
  - Clear separation between modules
  - Changes in one module do not affect others
  - Module interfaces are well-defined
  - System remains stable during module updates

3 UML Diagrams

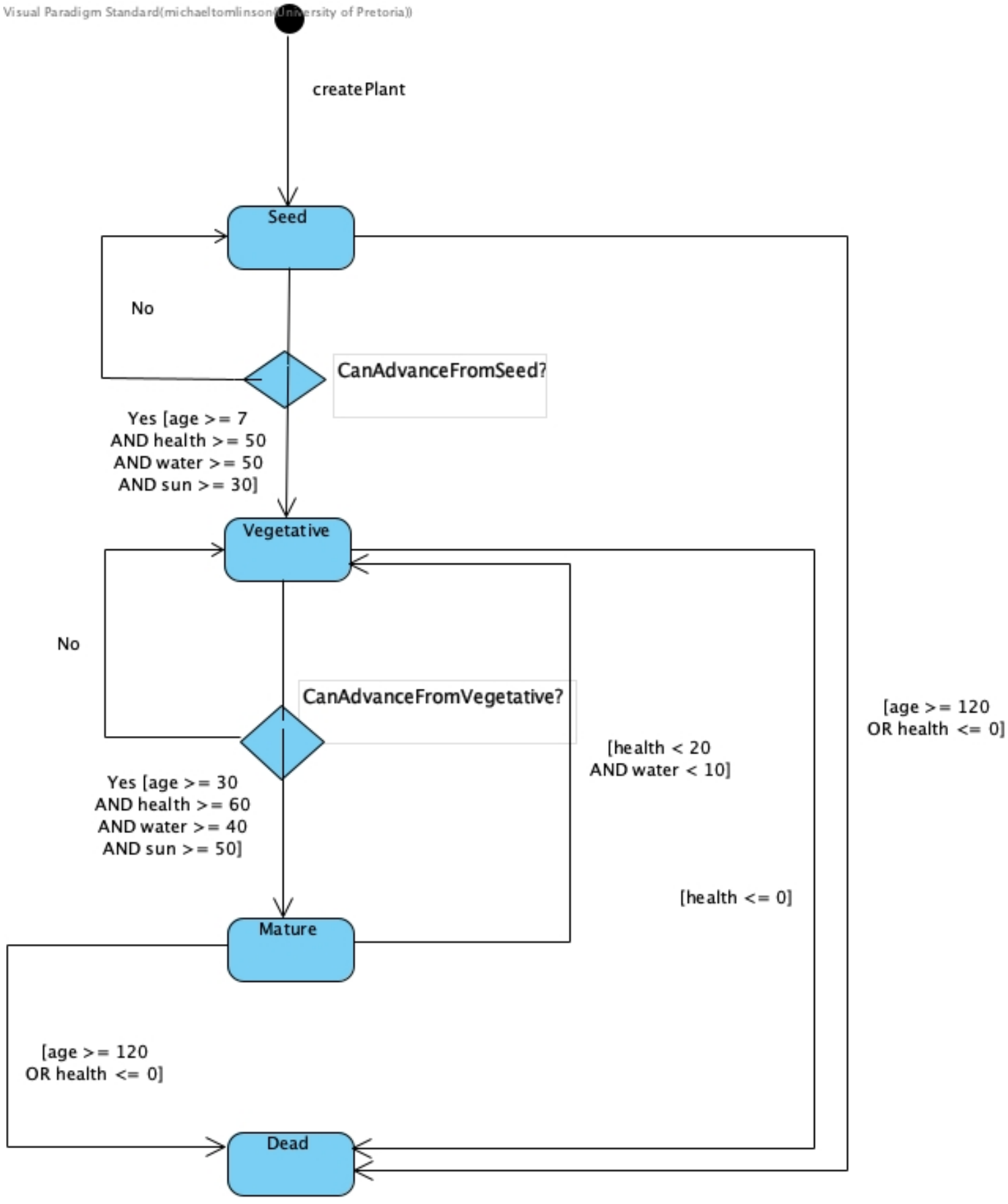
3.1 High-Level Activity Diagram



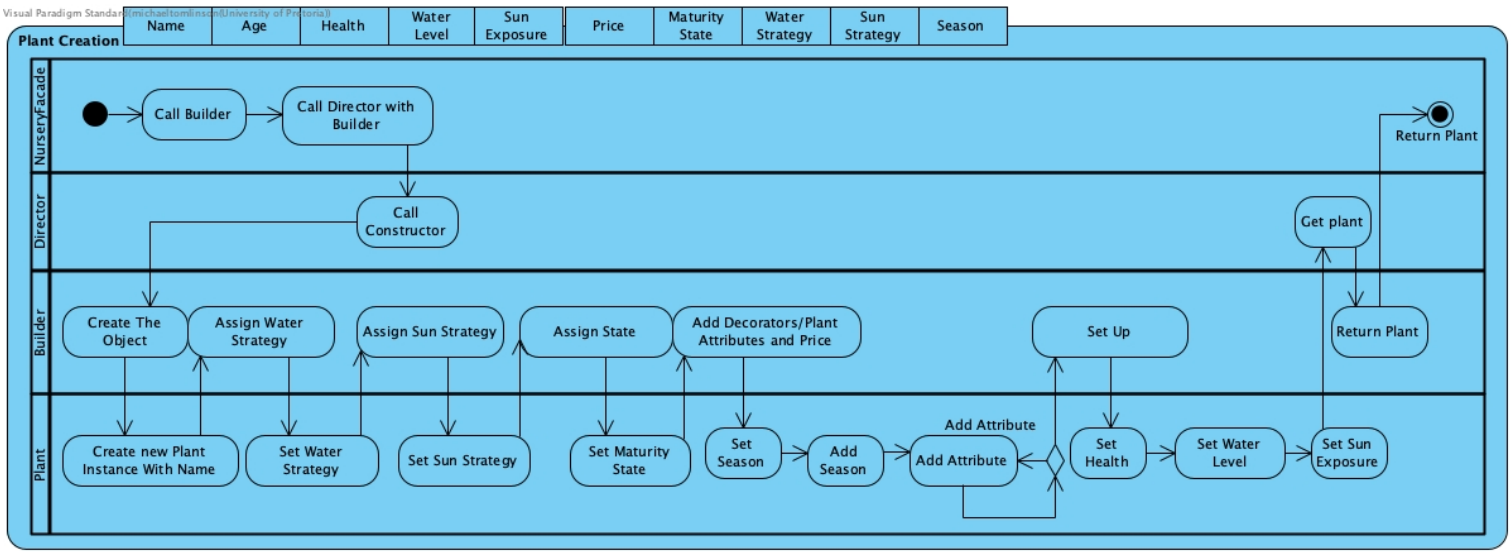
3.2 Class Diagram



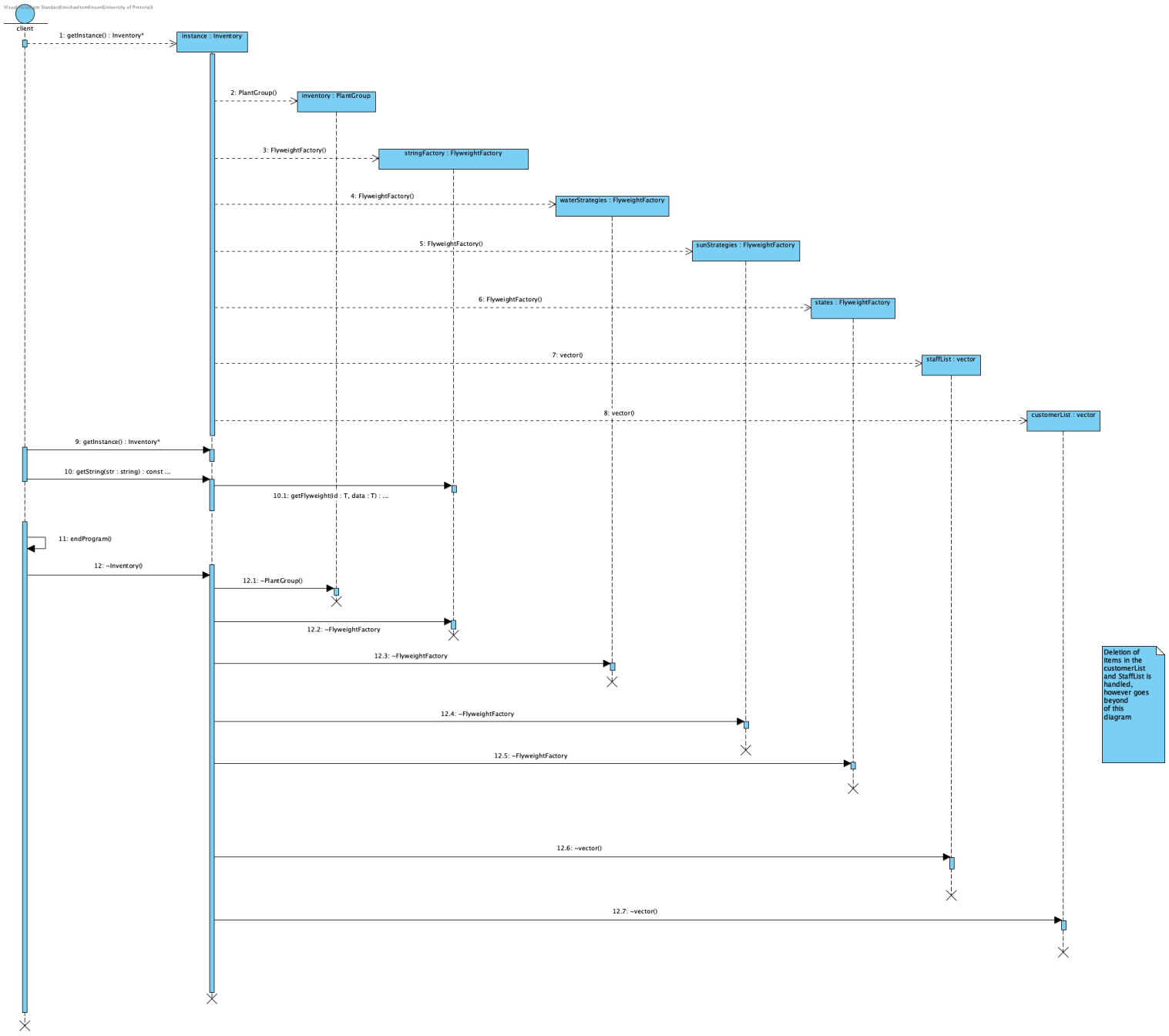
3.3 State Diagram



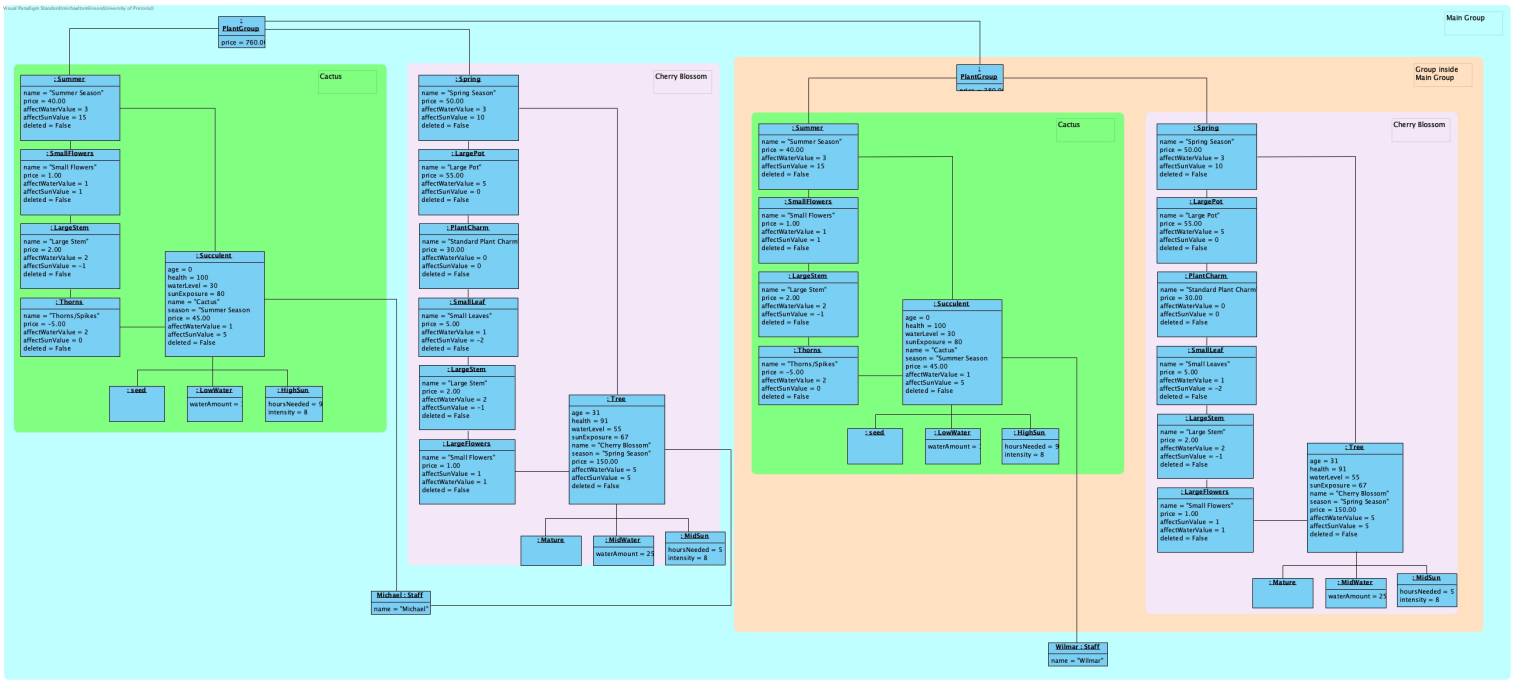
3.4 Activity Diagram



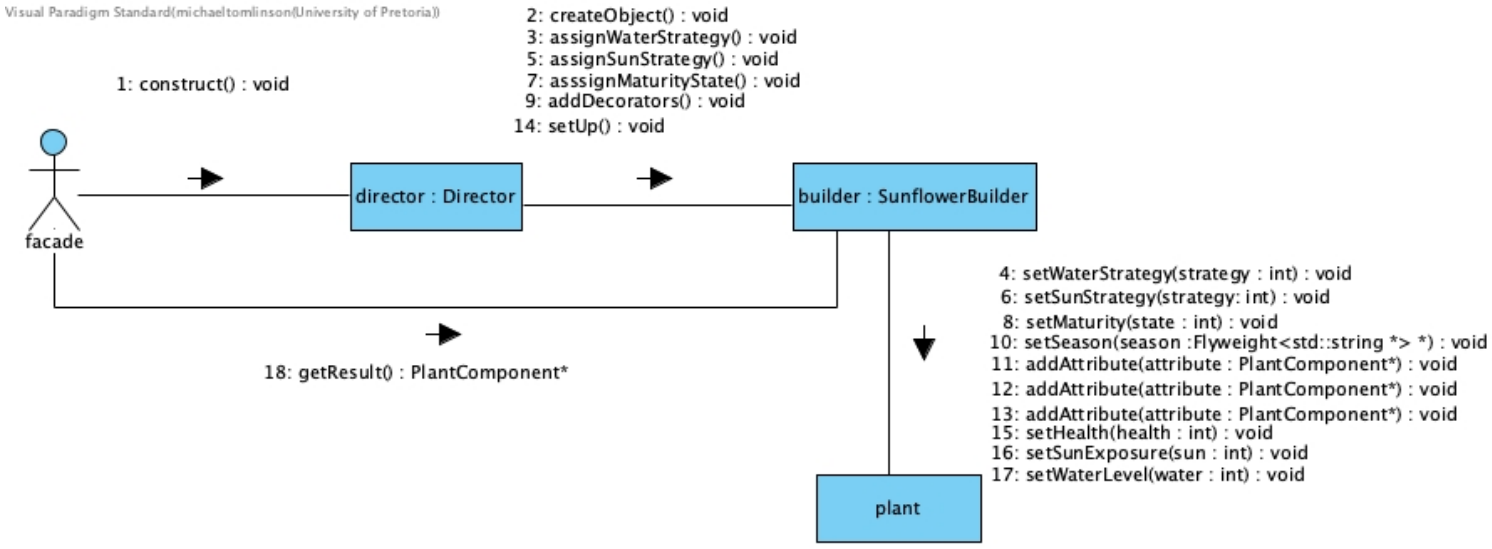
3.5 Sequence Diagram



3.6 Object Diagram



3.7 Communication Diagram



4 Design Patterns

4.1 Overview

The Photosyntech Plant Nursery Simulator employs 12 design patterns to create a robust, scalable system for managing plant lifecycle simulation, inventory organisation, and staff-customer interactions.

4.2 Pattern Distribution

Creational Patterns (3)	**Structural Patterns (4)**	**Behavioural Patterns (5)**
• Builder • Prototype • Singleton	• Composite • Decorator • Flyweight • Facade	• Iterator • Mediator • Observer • State • Strategy

4.3 Builder

Classification: Creational — **Strategy:** Delegation (Object)

Intent: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

4.3.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

- **FR-1: Plant Species Creation and Configuration** – Provides systematic plant creation from base types (Succulent, Shrub, Tree, Herb) with complete initialisation via `Builder` interface and `Director` orchestration.

Supporting Requirements

- **NFR-2: Maintainability/Extensibility** – New plant species require only a new builder class without modifying core plant logic.
- **NFR-5: Reliability** – Ensures plants are always in a valid state after construction via guaranteed step execution order.
- **FR-17: Unified System Interface** – Provides consistent construction interface (`createObject()`, `assignWaterStrategy()`, etc.) across all plant types.

4.3.2 Pattern Benefits

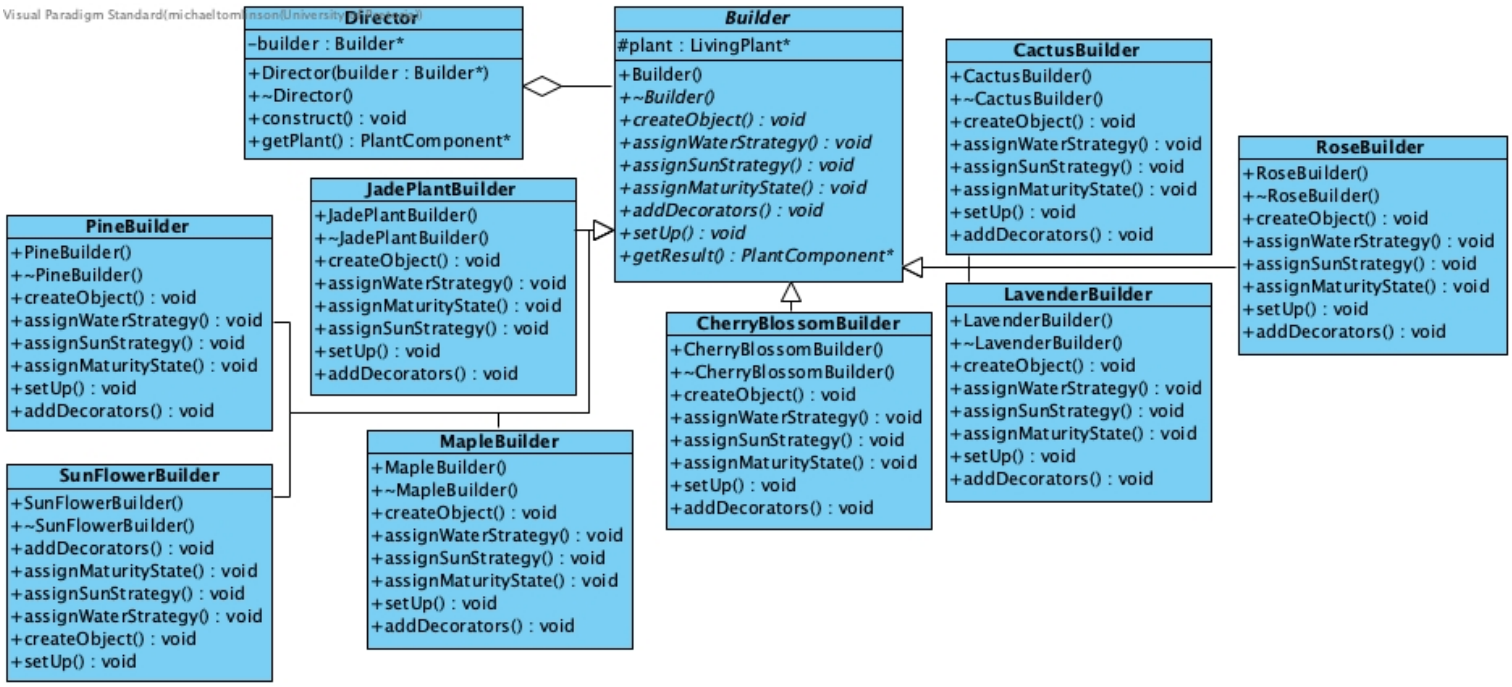
General Benefits:

- Separates construction logic from product representation; enables different representations via same process.
- Guarantees valid object state through controlled, sequential step execution.
- Simplifies client code by abstracting complex multi-step initialisation.

Implementation Benefits:

- Supports 8+ plant species with minimal code duplication; decouples species from core `LivingPlant`.
- Guarantees consistent initialisation order across all builders via `Director`.
- Enables extensible decorator and strategy application without modifying builders.

4.3.3 Class Diagram



4.3.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Director	`Director`	Orchestrates construction steps in correct order; calls builder methods sequentially to ensure valid plant state.
Builder	`Builder (abstract)`	Defines construction interface (`createObject()`, `assignWaterStrategy()`, `assignSunStrategy()`, `assignMaturityState()`, `addDecorators()`, `setUp()`, `getResult()`).
ConcreteBuilder	• `RoseBuilder` • `SunflowerBuilder` • `CactusBuilder` • `CherryBlossomBuilder` • `JadePlantBuilder` • `LavenderBuilder` • `MapleBuilder` • `PineBuilder`	Implements species-specific construction; assigns appropriate strategies and decorators; creates base plant type.
Product	`LivingPlant` and subclasses (Succulent, Shrub, Tree, Herb)	Complex object being constructed with all necessary strategies, decorators, and states.

4.3.5 Implementation Notes

Important Functions:

- **construct():** Director method orchestrating all builder steps in correct sequence; ensures valid plant state through guaranteed step execution order.
- **addDecorators():** Applies seasonal and attribute decorators (flowers, leaves, stems, thorns, pots) in specific combinations per species; integrates with Decorator pattern.

Implementation Challenges:

- **Multi-Builder Sequence Consistency:** Eight concrete builders required enforced step order. Solution: Director pattern ensures identical execution sequence across all builders, preventing state inconsistencies.
- **Flyweight Integration:** Initial approach created new Strategy/State instances per plant (memory waste). Solution: Switched to ID-based flyweight lookups; builders retrieve shared instances from singleton factories.
- **Decorator Chain Management:** Decorators attached to root caused infinite Iterator loops. Solution: Use `getDecorator()->addAttribute()` to attach decorators to first node, preserving composite hierarchy.

4.4 Composite

Classification: Structural — **Strategy:** Delegation (Object)

Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

4.4.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

- **FR-7: Hierarchical Plant organisation** – Provides tree-based organisation where individual plants and groups can be treated uniformly through `PlantComponent` interface; enables nesting of groups at arbitrary levels while supporting common operations on both leaves and composites.

Supporting Requirements

- **FR-9: Seasonal Plant Filtering** – Composite hierarchy enables iterators to traverse plant collections for filtering operations without exposing internal structure.
- **NFR-4: Scalability** – Hierarchical organisation efficiently manages large plant collections via composite containment rather than flat arrays.
- **FR-17: Unified System Interface** – Provides consistent interface for operations on both individual plants and groups; clients operate uniformly regardless of component type.

4.4.2 Pattern Benefits

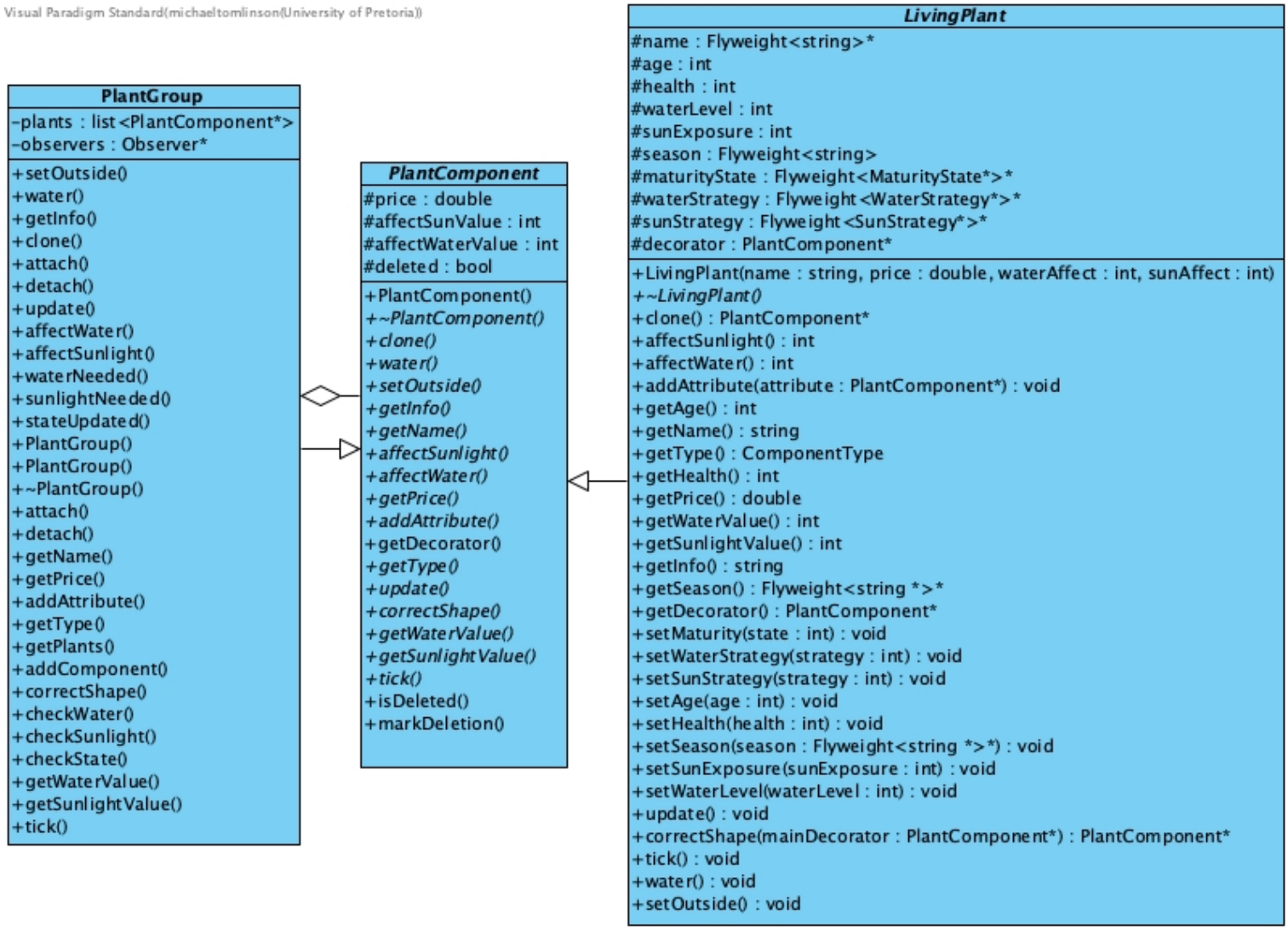
General Benefits:

- Treats individual objects and compositions uniformly through common interface; eliminates type checking.
- Enables flexible hierarchies with arbitrary nesting; transparent tree operations on entire structures.
- Facilitates adding new component types without modifying existing client code.

Implementation Benefits:

- Supports hierarchical inventory organisation; operations (water) apply uniformly via recursive delegation.
- Group-level observer notifications reduce overhead; iterators traverse transparently for filtering.
- Compatible with Decorator for applying attributes to entire groups.

4.4.3 Class Diagram



4.4.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Component	`PlantComponent` (`abstract`)	Declares common interface for both individual plants and plant groups; defines methods like `water()`, `getInfo()`, `getPrice()`, `clone()` that work uniformly on leaves and composites.
Leaf	• `LivingPlant` • `Succulent` • `Shrub` • `Tree` • `Herb`	Represents individual plants with no children; implements all component operations directly.
Composite	`PlantGroup`	Represents groups of plants; stores child `PlantComponent` instances in `plants` list; implements group-level operations that delegate to children; acts as `Subject` for observer notifications.
Client	• `Singleton` (inventory) • `Iterator` • `Staff` • `Facade`	Works with components through `PlantComponent` interface without distinguishing between individual plants and groups.

4.4.5 Implementation Notes

Important Functions:

- **getPrice():** Calculates total price of all plants in group through recursive aggregation. The delegation is polymorphic, allowing the call to cascade correctly through the decorator chain (`PlantAttributes::getPrice()`) to ensure accurate pricing.
- **getInfo():** Returns formatted information about all contained plants via recursive traversal. **The Composite (`PlantGroup`) uses the non-standard call `component->getDecorator()->getInfo()` to explicitly force the start of the Decorator chain, compromising the uniform interface intended by the Composite pattern.**
- **clone():** Creates a deep copy of the `PlantGroup` structure and delegates the clone operation recursively. **However, the `LivingPlant` (Leaf) copy constructor only performs a shallow copy of the base plant and explicitly sets its decorator pointer to `nullptr`, thus breaking the deep copy of the Decorator chain at the Leaf level.**
- **addComponent():** Adds child component to group with validation for decorated plants; ensures decorator chain integrity during hierarchy construction.

Implementation Challenges:

- **Handling Decorated Plants in Hierarchy:** `addComponent()` must preserve decorator chains embedded in plants. Solution: Accept any `PlantComponent` polymorphically; decorator chain remains intact, accessed via `getDecorator()`.
- **Non-Uniform Delegation for Getters:** Operations like `getPrice()` and `water()` use standard polymorphism, but `getInfo()` **requires explicit access to the decorator via** `getDecorator()` in the `PlantGroup` to ensure all attribute details are included. This non-uniform delegation compromises the Composite's intent to treat all components uniformly.
- **Polymorphic Ambiguity:** `getDecorator()` returns `this` for non-plants (groups), creating confusion. Requires careful caller understanding.
- **Memory Management (Destructors):** Due to the tight coupling with Decorator, **complex deletion logic** is required in both `PlantGroup:: PlantGroup()` and `LivingPlant:: LivingPlant()` to prevent **double-deletion** of the Decorator chain while ensuring all dynamically allocated objects are properly cleaned up.

4.5 Decorator

Classification: Structural — **Strategy:** Delegation (Object)

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

4.5.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

- **FR-11: Plant Decoration** – Allows plants to be customised with pots, charms, and embellishments; supports multiple customisations applied incrementally via `PlantAttributes` decorators while maintaining core plant functionality.

Supporting Requirements

- **NFR-2: Maintainability/Extensibility** – New decorations can be added without modifying existing plant classes or the decorator base.
- **FR-1: Plant Species Creation and Configuration** – Decorators contribute to the final plant configuration applied after species construction.
- **FR-9: Seasonal Plant Filtering** – Seasonal decorators (Spring, Summer, Autumn, Winter) enable season-based plant categorisation and filtering.

4.5.2 Pattern Benefits

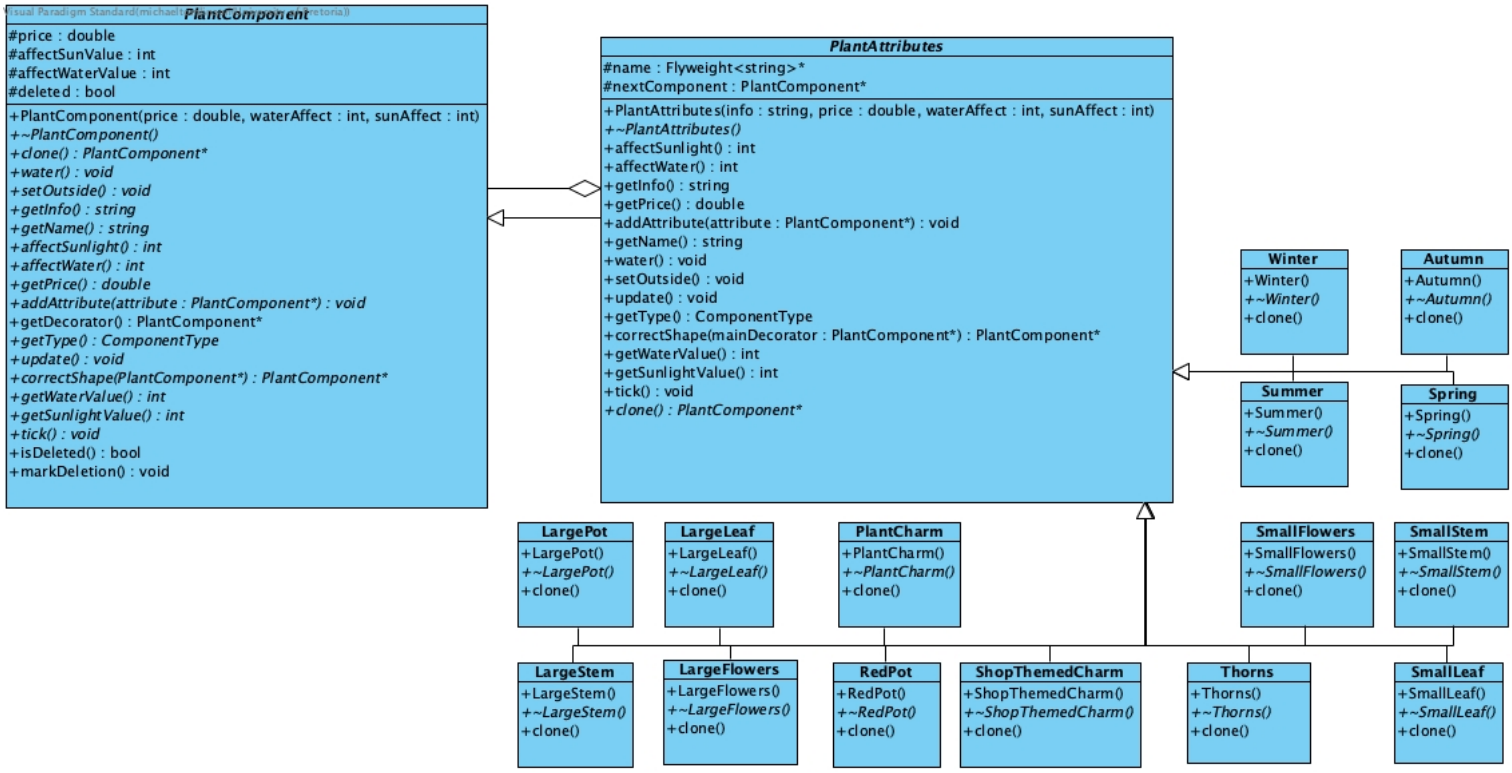
General Benefits:

- Avoids subclass explosion; enables runtime addition/removal of responsibilities via composition.
- Supports multiple concurrent decorations on single objects with single responsibility per decorator.
- Flexible attribute combinations without polluting class hierarchy.

Implementation Benefits:

- Enables flexible plant customisation (pots, charms, features, seasons); price and care cascade through chain.
- Seasonal decorators enable Iterator-based filtering; decorator stacking preserves Composite interface.
- Clone operations recursively preserve decorator layers via Prototype integration.

4.5.3 Class Diagram



4.5.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Component	`PlantComponent (abstract)`	Defines interface for objects that can have decorations added; methods for info, price, care requirements, and lifecycle operations.
ConcreteComponent	`LivingPlant` and subclasses (Succulent, Shrub, Tree, Herb)	Base plant objects being decorated; provide core plant attributes and lifecycle.
Decorator	`PlantAttributes (abstract)`	Maintains reference to wrapped PlantComponent; implements PlantComponent interface conforming to component; cascades operations through `nextComponent`.
ConcreteDecorator	• `RedPot`, `LargePot` • `PlantCharm`, `ShopThemedCharm` • `SmallFlowers`, `LargeFlowers` • `SmallStem`, `LargeStem` • `SmallLeaf`, `LargeLeaf` • `Summer`, `Winter`, `Spring`, `Autumn` • `Thorns`	Add specific visual attributes and modify plant characteristics via `getPrice()`, `affectWater()`, `affectSunlight()`, `getInfo()`.

4.5.5 Implementation Notes

Important Functions:

- **affectWater()/affectSunlight()**: Accumulate modifications from all decorator layers through recursive chain traversal (delegating to `nextComponent`) to determine cumulative care requirements.

- **addAttribute()**: Implements complex logic to chain a new decorator, pushing the existing chain down one layer while maintaining pointer relationships. The `LivingPlant` acts as the required chain terminator.
- **clone()**: Recursively deep-clones the entire decorator chain (`PlantAttributes`) by calling `nextComponent-` **However, the** `LivingPlant` **copy constructor explicitly sets the decorator pointer to** `nullptr`, breaking the chain preservation at the base component level.

Implementation Challenges:

- **Complex Pointer Chaining:** The `LivingPlant` (ConcreteComponent) is responsible for initializing and terminating the decorator chain by calling `attribute->addAttribute(this)` in its own `addAttribute()`. This complex initialization logic replaces simpler client construction.
- **Memory Management and Double Deletion:** Since the Decorator chain is also a part of the Composite hierarchy, the destructors (`PlantAttributes:: PlantAttributes()` and `LivingPlant:: Livi` must include explicit checks and a deletion flag to safely delete the chain only once and prevent circular deletion or double-free errors.

4.6 Facade

Classification: Structural — **Strategy:** Simplification

Intent: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use by coordinating multiple patterns and hiding implementation complexity.

4.6.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

- **FR-17: Unified System Interface** – Provides simplified interface layer handling all business logic and coordinating subsystem interactions; enables TUI and CLI to interact through simple function calls in `NurseryFacade`; decouples business logic from presentation layer.

Supporting Requirements

- **NFR-3: Usability** – Simplifies complex operations into easy-to-understand methods like `createPlant()`, `waterPlant()`, and `customerPurchase()`.
- **NFR-5: Reliability** – Centralises coordination logic, reducing coupling between subsystems and ensuring consistent operation sequencing.
- **NFR-2: Maintainability/Extensibility** – Changes to subsystem implementations remain hidden behind stable facade interface; new operations added through single class.
- **All FRs** – Facade provides access to all functional requirements through unified interface, delegating to appropriate pattern implementations.

4.6.2 Pattern Benefits

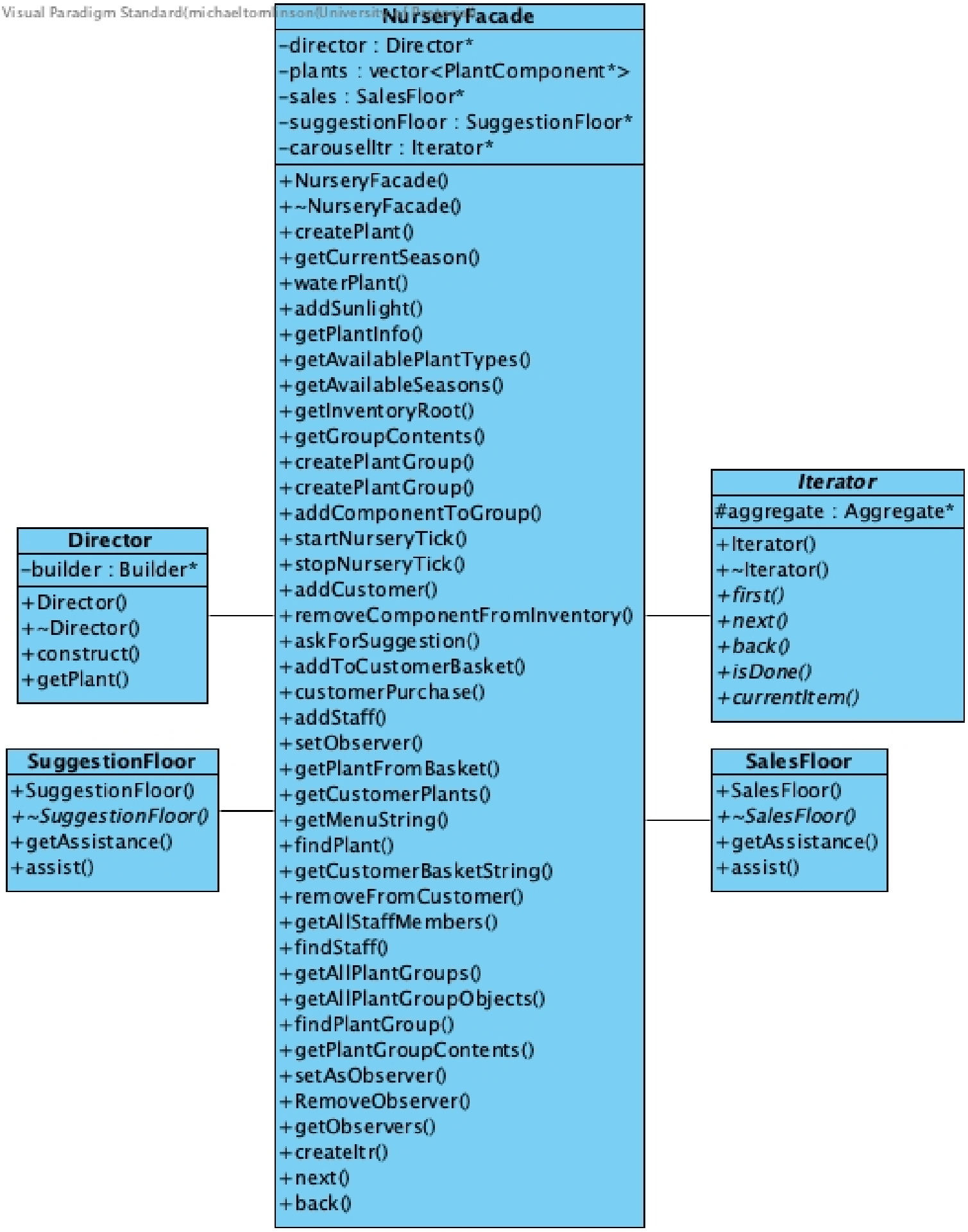
General Benefits:

- Shields clients from subsystem complexity; provides single entry point for system operations.
- Reduces dependencies between clients and subsystems; promotes loose coupling.
- Simplifies client code by eliminating need to understand multiple subsystem interfaces.
- Enables layered architecture with clear separation between presentation and business logic.

Implementation Benefits:

- Coordinates 12 design patterns through single integration point; TUI/CLI need only know facade interface.
- Enables complex multi-step operations through simple method calls (e.g., plant creation + decoration + inventory addition).
- Provides stable API for external interfaces despite internal pattern changes or additions.
- Centralises business logic coordination, reducing duplicate coordination code in UI layers.

4.6.3 Class Diagram



4.6.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Facade	`NurseryFacade`	Provides high-level methods for common operations; delegates to appropriate subsystems; coordinates complex multi-step processes; serves as single integration point for entire system.
Subsystems	• `Director` (Builder) • `Singleton` (Resources) • `SalesFloor` (Mediator) • `SuggestionFloor` (Mediator) • `AggPlant, AggSeason, AggPlantName` (Iterator factories) • `PlantGroup` (Composite) • `PlantDecorator` (Decorator) • `Observer/Subject` (Monitoring) • `State, Strategy, Flyweight` (Care systems)	Implement detailed subsystem functionality; work together through facade coordination; handle pattern-specific operations.
Client	`TUI`, External interfaces	Uses facade to interact with system through simple method calls; unaware of subsystem complexity.

4.6.5 Implementation Notes

Important Functions:

- **createPlant(type):** Delegates to Builder/Director to construct plant from species string; adds to inventory via Singleton; returns configured plant instance.
- **waterPlant(plant):** Executes plant's water strategy; delegates to Strategy pattern implementation; triggers Observer notifications if thresholds crossed.
- **addSunlight(plant):** Executes plant's sun strategy; coordinates State transitions if maturity changes; notifies observers.
- **addCustomer(name), addStaff(name):** Creates and registers Customer/Staff instances with Singleton for memory management; initialises Mediator connections.
- **addToCustomerBasket(customer, plant):** Routes purchase request through SalesFloor Mediator; validates inventory availability; updates customer basket.
- **customerPurchase(customer):** Coordinates multi-step transaction: validates basket, processes payment, removes from inventory, transfers ownership.
- **askForSuggestion(customer):** Routes customer query through SuggestionFloor Mediator to available staff; returns recommendation.
- **setObserver(staff, plantGroup):** Attaches Staff observer to PlantGroup subject; enables monitoring notifications.
- **createPlantGroup(name):** Creates Composite node in inventory hierarchy; returns group for further organisation.
- **addComponentToGroup(parent, child):** Manages Composite structure; validates hierarchy rules; updates inventory tree.
- **startNurseryTick(), stopNurseryTick():** Coordinates global time progression; triggers State transitions and care cycles.
- **createItr(filter), next(), back():** Delegates to Iterator factories (AggPlant, AggSeason, Agg-

PlantName) for filtered traversal; encapsulates iterator management.

4.7 Flyweight

Classification: Structural — **Strategy:** Memory Optimisation

Intent: Use sharing to support large numbers of fine-grained objects efficiently by separating intrinsic (shared) state from extrinsic (unique) state, minimising memory consumption through factory-managed caches.

4.7.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

- **FR-10: Shared Data Memory Optimisation** – Minimises memory usage by sharing immutable data objects across multiple plant instances; reduces memory footprint for large inventories from potentially thousands of duplicate objects to a single cached instance per unique data value.

Supporting Requirements

- **NFR-4: Scalability** – Enables system to handle 10,000,000+ plant instances without memory exhaustion; memory footprint scales sub-linearly with number of plants through shared flyweight references.
- **NFR-1: Performance** – Reduces memory allocation overhead, improving overall system performance; cache lookup via `getFlyweight()` is O(1) amortized cost due to the use of a map.
- **FR-6: Centralised Inventory** – Works with Singleton to manage shared resources efficiently; factories managed by singleton for global cache control.

4.7.2 Pattern Benefits

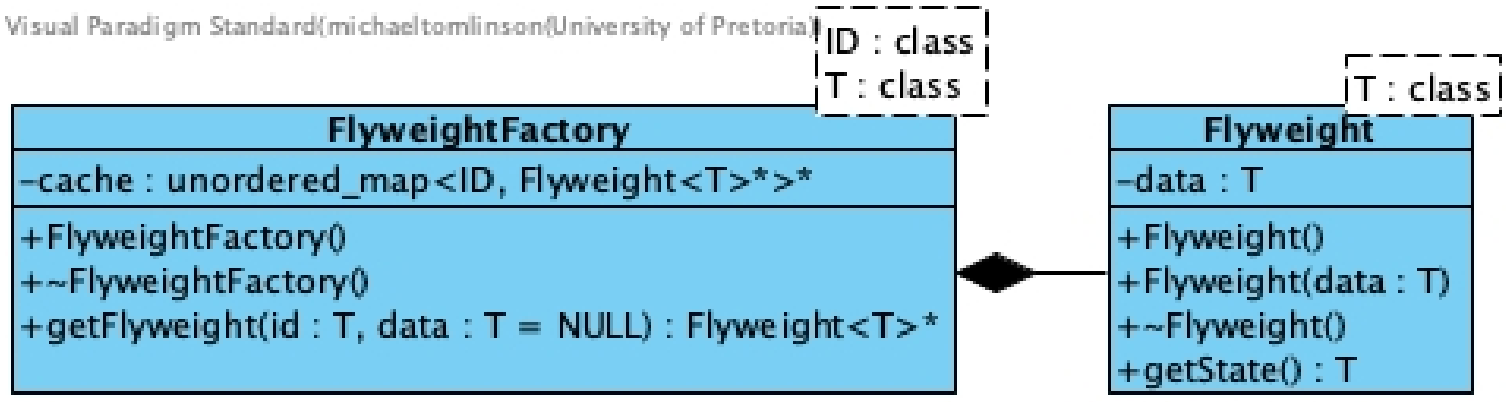
General Benefits:

- Dramatically reduces memory consumption by sharing immutable data across large object collections.
- Transparent sharing via factory cache; clients use objects normally without awareness.
- Supports unbounded object proliferation with bounded memory usage; improves system performance.

Implementation Benefits:

- Reduces memory by approximately 75% enables 10,000,000+ plant instances with minimal overhead.
- Three factories manage shared strategies (water, sun, season); template design supports any immutable type.
- Singleton integration ensures global cache control; cloned plants automatically share flyweight instances.

4.7.3 Class Diagram



4.7.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Flyweight	`Flyweight<T>`	Stores shared immutable data of type `T` and provides access through `getState()`; template enables type-safe sharing of strategies and season names.
FlyweightFactory	`FlyweightFactory<ID, T>`	Creates and manages flyweight objects using a cache keyed by identifier; returns existing instances or creates new ones as needed via `getFlyweight(ID id, T data)`.
Client	• `LivingPlant` • `Singleton`	Uses flyweights for shared data (seasons, water strategies, sun strategies) instead of storing copies; accesses factories through singleton for global cache consistency.
Shared Data	• `WaterStrategy` • `SunStrategy` • `string` • `maturityState`	Immutable data objects that are shared across multiple plants; never modified after creation, enabling safe concurrent access.

4.7.5 Implementation Notes

Important Functions:

- **getFlyweight(ID id, T data):** Implements cache lookup with lazy creation; checks cache for existing instance via key lookup, creates and caches new flyweight if not found, achieving O(log n) amortized cost.
- **Destructor:** Cleans up all cached flyweight instances to prevent memory leaks; called by Singleton during system shutdown.

Implementation Challenges:

Challenge 1: Balancing Memory Benefits Against Compile-Time Cost

- Repeated data causes unnecessary overhead that can be avoided with the use of flyweights, however we wanted to ensure that the positives outweighed the negatives.

Solution: reuse of data

- Evaluated memory benefits using `sizeof()` measurements:
  - Assume a Living plant object has:
    - 5 attributes each with a string
    - 1 name attribute with a string
- Sizes:
  - strings are 32 bytes
  - Flyweight pointers are 8 bytes
- **Proof:**
  - Internally stored strings :
    - Assume a living plant is made to the above assumptions, where there is a plant that stores all its strings internally without the use of flyweight, as strings are 32 bytes this would be 6 x 32 pointers accumulating in 192 bytes per plant. Let this plant be X
  - Externally stored strings :
    - Assume a living plant is made to the above assumptions, where there is a plant that stores all its strings externally with the use of flyweights, as strings are 32 bytes this would be 6 x 32 pointers accumulating in 192 bytes stored in 6 flyweights as well as 6 x 8 pointers to those flyweights in each attribute accumulating to 48 bytes and in total 240. let the plant itself with 48 bytes of pointers be called Y and let the group of flyweights be called F
- **Memory Usage to scale**

  - X has 48 less bytes than Y+F
- **2 of X and Y**
  - 2X = 2(192) = 384 Bytes
  - 2Y + F = 2 (48) +192 = 288 Bytes
  - Saving = 96 Bytes
- **1000 of X and Y**
  - 1000X = 1000(192) = 192 000 Bytes
  - 1000Y + F = 1000(48) +192 = 48 192 Bytes
  - Saving = 143 808 Bytes = 143 KB
- **1 000 000 of X and Y**
  - 1 000 000X = 1 000 000(192) = 192 000 000 Bytes
  - 1 000 000Y + F = 1 000 000(48) +192 = 48 000 192 Bytes
  - Saving = 143 999 808 = 143 .9 MB
- **100 000 000 of X and Y**
  - 100 000 000X = 100 000 000(192) = 19 200 000 000 Bytes
  - 100 000 000(Y) + F = 100 000 000(48) + 192 = 4 800 000 192 Bytes
  - Saving = 14 399 999 808 = 14.4 GB
- **Efficiency Calculation**
  - (4 800 000 192 Bytes / 19 200 000 000 Bytes) x 100 = 25
  - Thus 75% less memory is used.
- Solution for long compile time.
  - Reduced template instantiation by centralising factory access through Singleton; minimises re-compilation when adding new flyweight types.

4.8 Iterator

Classification: Behavioural — **Strategy:** Delegation (Object)

Intent: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

4.8.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

- **FR-9: Seasonal Plant Filtering** – Enables traversal and filtering of plants by season through `SeasonIterator`.
- **FR-7: Hierarchical Plant organisation** – Supports stack-based traversal through nested `PlantGroup` structures; enables iteration through arbitrary nesting levels.

Supporting Requirements

- **NFR-3: Usability** – Simplifies browsing via uniform iterator interface with bidirectional navigation.
- **FR-17: Unified System Interface** – Provides consistent access methods (`first()`, `next()`, `back()`, `isDone()`, `currentItem()`).

4.8.2 Pattern Benefits

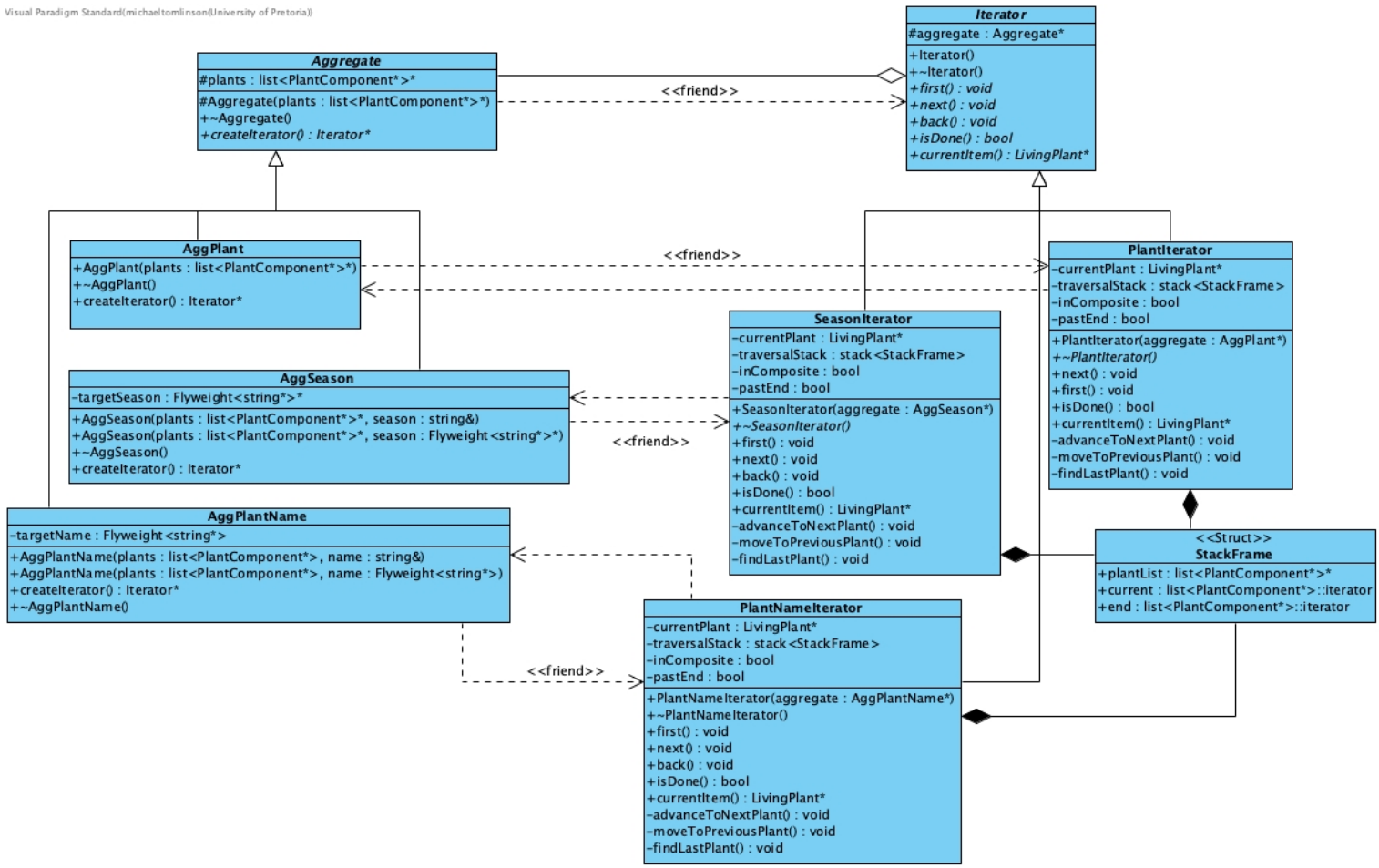
General Benefits:

- Hides collection structure; encapsulates traversal logic and supports multiple concurrent traversals.
- Enables filtering and transformation during iteration; simplifies client code via unified interface.
- Decouples traversal logic from business rules; extensible filtering criteria.

Implementation Benefits:

- Supports seasonal filtering without exposing internal lists; handles arbitrarily nested groups.
- Achieves amortized O(1) for `next()` and `back()` via stack-based iteration.
- Enables bidirectional navigation through composite hierarchies with extensible filtering criteria.

4.8.3 Class Diagram



4.8.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Iterator	`Iterator (abstract)`	Defines bidirectional traversal interface (`first()`, `next()`, `back()`, `isDone()`, `currentItem()`).
ConcreteIterator	• `PlantIterator` • `SeasonIterator` • `PlantNameIterator`	Implements bidirectional stack-based traversal; applies filtering logic via `advanceToNextPlant()` and `moveToPreviousPlant()`.
Aggregate	`Aggregate (abstract)`	Declares factory method `createIterator()`.
ConcreteAggregate	• `AggPlant` • `AggSeason` • `AggPlantName`	Create type-specific iterators; `AggSeason` and `AggPlantName` use flyweight pointers for O(1) filtering.

4.8.5 Implementation Notes

Important Functions:

- **advanceToNextPlant():** Stack-based forward traversal through nested composite hierarchies; pushes new frames when descending into groups, pops when backtracking; achieves amortized O(1) next() complexity. Applies filter criteria (seasonal, name-based, or unfiltered) during traversal.
- **moveToPreviousPlant():** Stack-based backward traversal; when hitting a group during reverse navigation, descends to its deepest last element via `findLastPlant()`. Maintains filter consistency in reverse direction.
- **findLastPlant():** Recursively positions iterator at last leaf within a group's hierarchy for backward navigation start point.

- **createIterator():** Factory method in aggregates; `AggSeason` and `AggPlantName` create filtered iterators via flyweight pointers; `AggPlant` creates unfiltered traversal.

Implementation Challenges:

- **Stack-Based Traversal Efficiency:** Initial recursive approach was inefficient. Solution: Replaced with stack-based iteration achieving O(1) amortized `next()` and `back()`; also replaced `dynamic_cast` with enum-based `getType()`.
- **Bidirectional Navigation in Composite:** Moving backward into `PlantGroup` requires descending to its deepest leaf. Solution: Implemented `findLastPlant()` for backward traversal start point.
- **Code Duplication in Filtering:** Multiple filtering iterators (seasonal, name-based) caused duplication. Solution: Created generic filtering architecture with `SeasonIterator` (season flyweight), `PlantNameIterator` (name flyweight), and `PlantIterator` (unfiltered) all using common stack-based traversal pattern.

4.9 Mediator

Classification: Behavioural — **Strategy:** Delegation (Object)

Intent: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

4.9.1 Functional and Non-Functional Requirement Mapping

- **FR-12: Staff-Customer Communication** – Provides centralised coordination between staff and customers on multiple floors; maintains separate mediators (SalesFloor, SuggestionFloor) for different interaction types; routes requests through `getAssistance()` and `assist()` methods.
- **FR-13: Plant Sales Transactions** – SalesFloor mediator coordinates customer purchases and staff processing; mediates interactions involving plant groups through `assistPurchases()` and `purchasePlants()` methods.
- **NFR-5: Reliability** – Decouples staff and customers, allowing changes to one without affecting the other; implements loose coupling through mediator routing.
- **NFR-2: Maintainability/Extensibility** – New interaction types can be added to mediator without modifying staff or customer classes; extensible through new Mediator subclasses.

4.9.2 Pattern Benefits

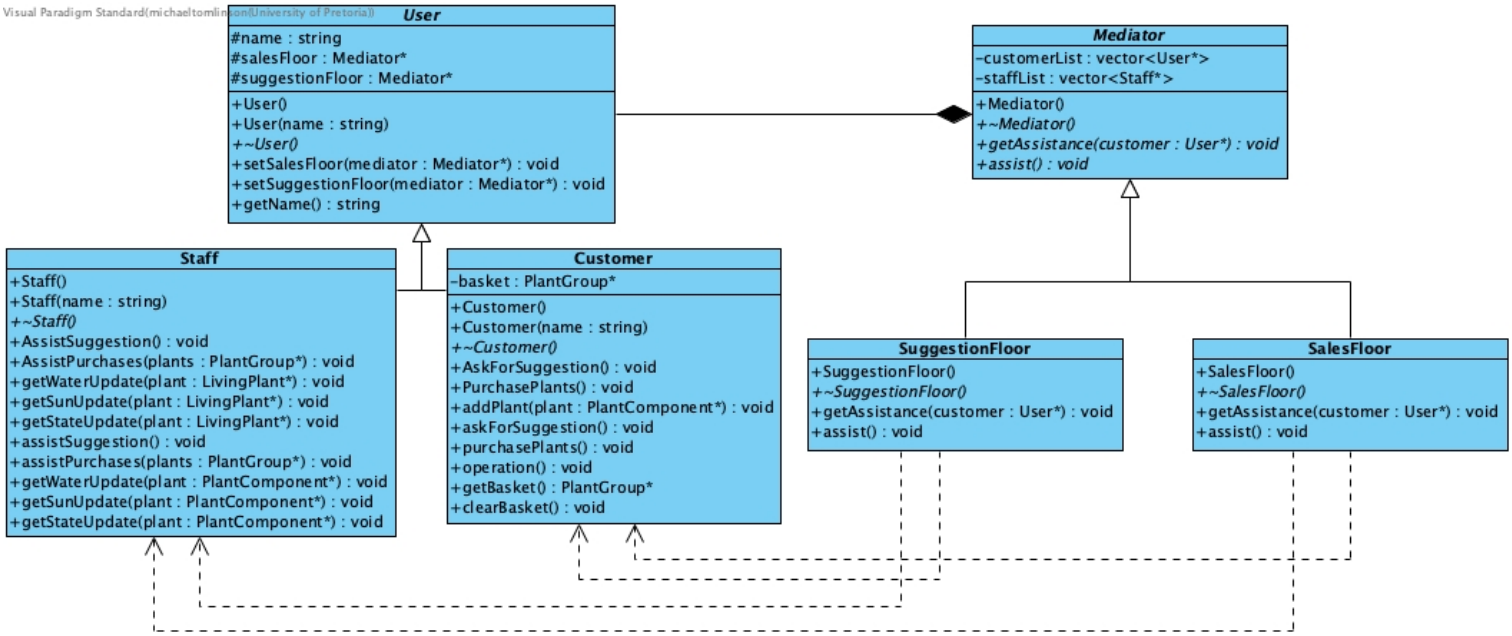
General Benefits:

- Decouples colleagues from direct dependencies; centralises interaction logic, preventing many-to-many coupling.
- Simplifies colleague classes by removing direct communication code; enables routing to appropriate mediators.
- Facilitates adding new interaction types and multiple concurrent mediators without modifying participants.

Implementation Benefits:

- SalesFloor and SuggestionFloor separate purchase and advisory coordination; extensible without coupling.
- Routes plant care notifications through observer-mediator integration.
- Supports scalability managing multiple staff and customer participants.

4.9.3 Class Diagram



4.9.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Mediator	`Mediator` (abstract)	Defines interface for colleague coordination via `getAssistance()` and `assist()`; maintains lists of participants (customers and staff).
ConcreteMediator	• `SalesFloor` • `SuggestionFloor`	• `SalesFloor`: Implements purchase coordination; routes customer purchase requests to sales staff; manages sales floor transactions. • `SuggestionFloor`: Implements advisory coordination; routes plant care inquiries to expert staff; manages suggestion floor interactions.
Colleague	`User` (abstract)	Base class for mediated participants; maintains references to both mediators (`salesFloor`, `suggestionFloor`); provides setter methods for mediator access.
ConcreteColleague	• `Customer` • `Staff`	• `Customer`: Concrete customer participant; interacts via `askForSuggestion()` and `purchasePlants()` through mediators; maintains shopping basket. • `Staff`: Concrete staff participant implementing both User (mediated communication) and Observer (plant monitoring); handles customer assistance and plant care notifications.

4.9.5 Implementation Notes

Important Functions:

- **getAssistance(customer):** Routes customer assistance request through mediator to identify available staff and coordinate response; prevents direct staff-customer coupling.
- **assistSuggestion()/assistPurchases():** Concrete mediator methods implementing coordination logic for specific interaction types (SuggestionFloor vs SalesFloor).

Implementation Challenges:

- **Evaluate use of different users:** The use case of different users caused a bit of implementation challenges, especially because we could not work with dynamic casting.
- **Type Boundary Enforcement:** Mediator must validate Customer vs Staff roles before routing requests. Solution: Type-checking logic before critical method invocations prevents category confusion.

4.10 Observer

Classification: Behavioural — **Strategy:** Delegation (Object)

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

4.10.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

- **FR-14: Plant Monitoring by Staff** – Allows staff members to monitor assigned plants and receive notifications about health changes and lifecycle transitions via `getWaterUpdate()`, `getSunUpdate()`, and `getStateUpdate()` methods.
- **FR-8: Inventory Tracking and Notifications** – Notifies staff of inventory changes through the `attach()` mechanism; supports multiple staff members monitoring the same inventory independently.

Supporting Requirements

- **NFR-5: Reliability** – Decouples plant groups from staff implementation, allowing changes to one without affecting the other through abstract `Observer` and `Subject` interfaces.
- **FR-15: Staff Action Execution** – Notifications trigger staff to execute appropriate care actions in response to plant state changes.

4.10.2 Pattern Benefits

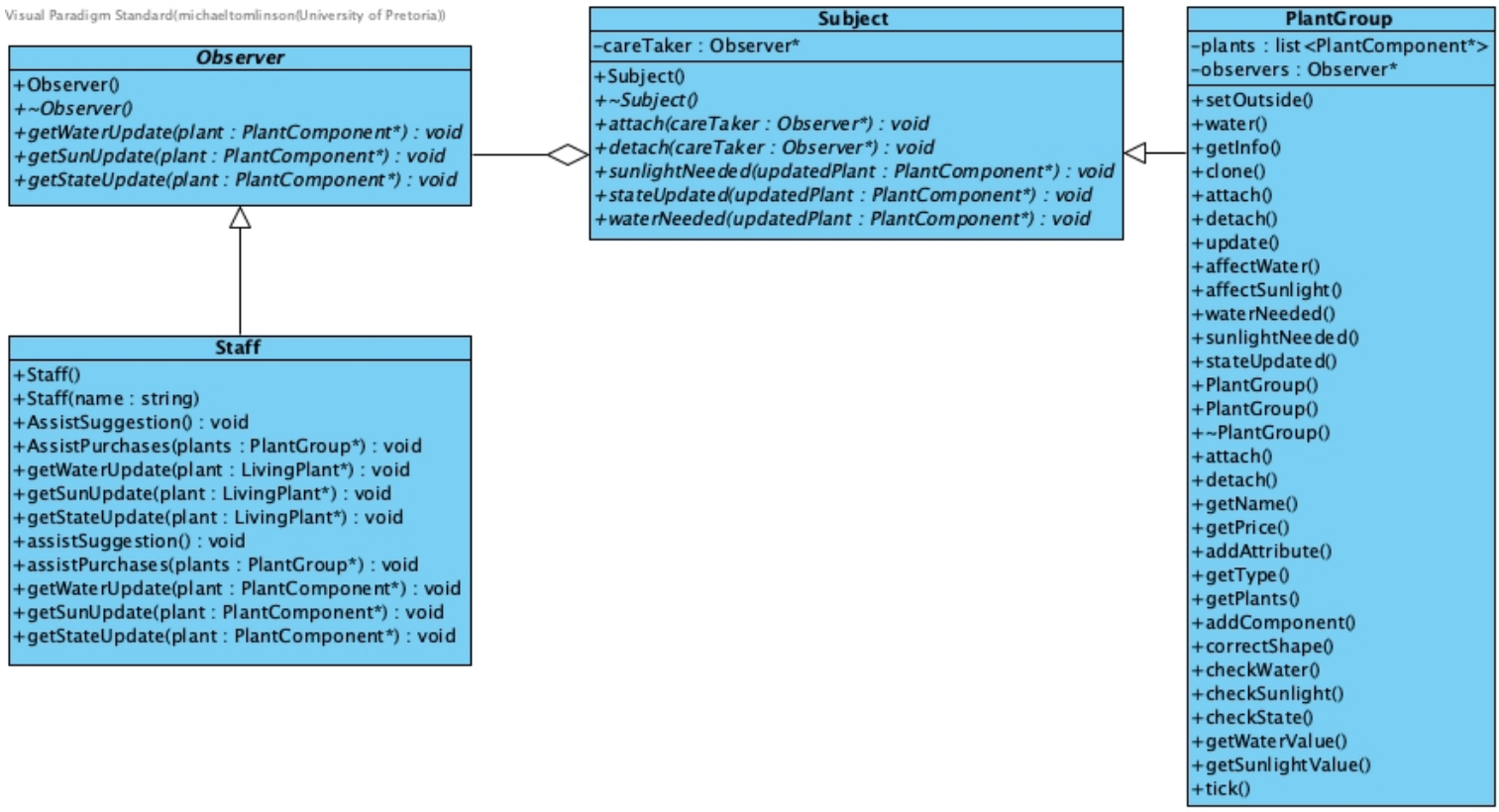
General Benefits:

- Establishes loose coupling through abstract interfaces; supports one-to-many relationships without subjects knowing observer implementations.
- Enables dynamic subscription/unsubscription at runtime; eliminates polling via push-based notifications.
- Facilitates adding new observer types without modifying subject code; simplifies event handling.

Implementation Benefits:

- Multiple staff monitor plant groups independently; plant groups unaware of staff implementation details.
- Staff dynamically added/removed without plant code modifications; automatic notifications eliminate polling.
- Decouples lifecycle transitions from care logic; scales efficiently with extensible observer types.

4.10.3 Class Diagram



4.10.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Subject	`Subject (abstract)`	Maintains list of observers and provides methods to attach/detach them; defines virtual methods to notify observers of plant care needs (`waterNeeded()`, `sunlightNeeded()`, `stateUpdated()`).
ConcreteSubject	`PlantGroup`	Implements `Subject` interface; maintains observer list; calls notification methods when plants require care or state changes occur.
Observer	`Observer (interface)`	Defines update interface for receiving notifications about plant status changes with three methods: `getWaterUpdate()`, `getSunUpdate()`, `getStateUpdate()`.
ConcreteObserver	`Staff`	Implements `Observer` interface; receives and responds to plant care notifications; executes appropriate care actions based on notification type.

4.10.5 Implementation Notes

Important Functions:

- **attach()/detach():** Manages observer registration; enables dynamic subscription/unsubscription of staff members to plant groups.
- **waterNeeded()/sunlightNeeded()/stateUpdated():** Broadcast notifications to all attached observers; iterates through observer list and calls corresponding update methods.

Implementation Challenges:

- **Generic Notifications Lack Actionability:** Original system sent bare "update" signals without context. Staff treated all notifications equally, preventing prioritisation. Solution: Extended notifications with context (what changed, severity level) enabling intelligent prioritisation.
- **Severity-Based Response Coordination:** Different changes require different urgency responses. Solution: Added severity levels (`CRITICAL`, `MODERATE`, `ROUTINE`) allowing staff to respond appropriately to each alert.

4.11 Prototype

Classification: Creational — **Strategy:** Delegation (Object)

Intent: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

4.11.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

- **FR-2: Plant Type Cloning** – Allows new plant instances to be created by copying existing plant types and decorations via the `clone()` method; maintains all properties including strategies, states, and decorator chains.

Supporting Requirements

- **NFR-1: Performance** – Cloning is more efficient than rebuilding plants from scratch for mass production scenarios.
- **NFR-4: Scalability** – Enables rapid creation of multiple plant instances without repeated initialisation overhead, supporting inventory growth.
- **FR-1: Plant Species Creation and Configuration** – Complements Builder by providing efficient template instantiation once prototypes are established.

4.11.2 Pattern Benefits

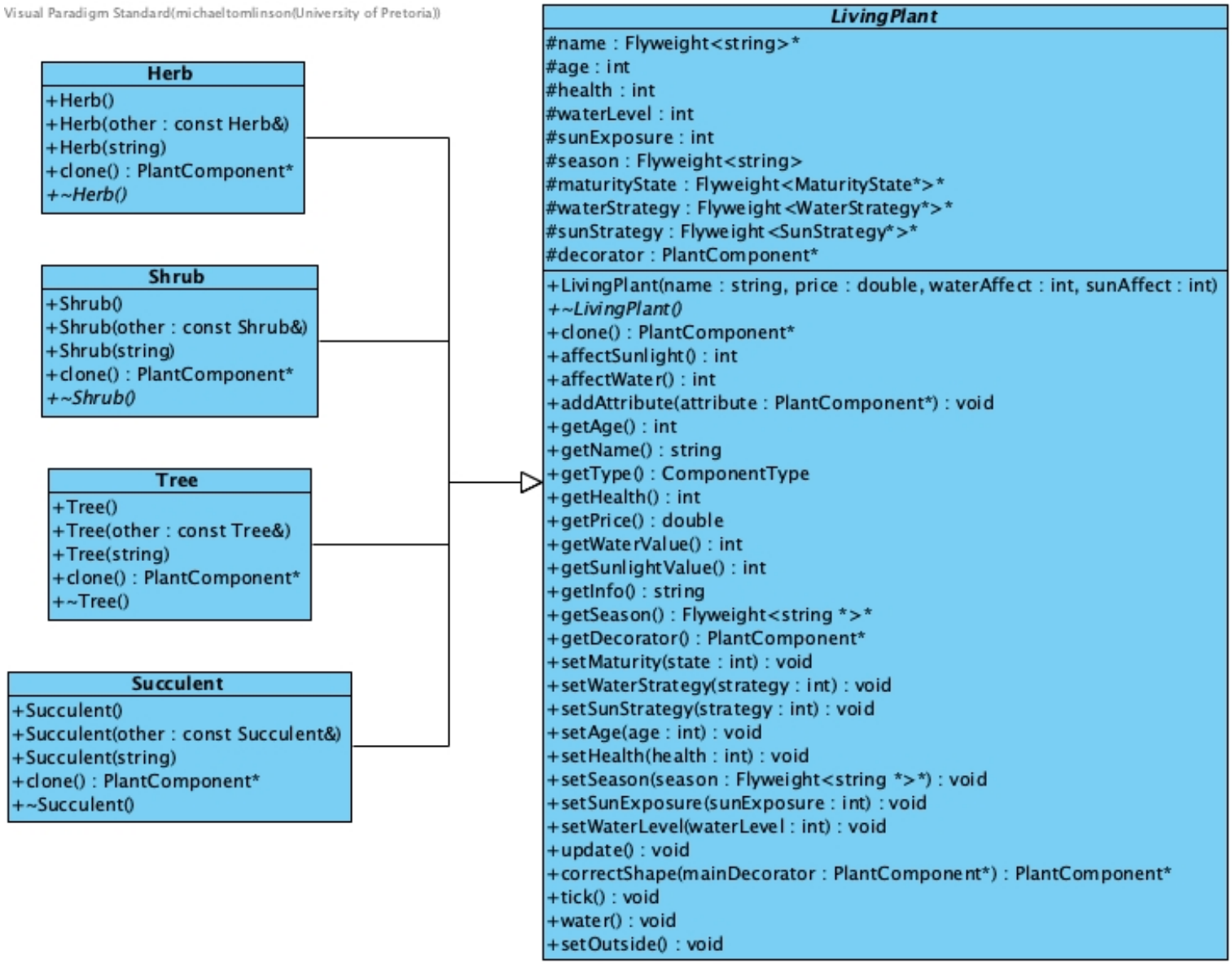
General Benefits:

- Avoids expensive object creation by copying existing objects; reduces initialisation overhead.
- Enables runtime object creation without hard-coded class dependencies; flexible strategies independent of hierarchies.
- Simplifies client code via direct instantiation alternatives.

Implementation Benefits:

- Supports efficient mass-production from templates; preserves decorator chains and strategies during cloning.
- Enables rapid plant variant creation; reduces memory and computation overhead for bulk operations.
- Maintains data consistency across cloned attributes and relationships.

4.11.3 Class Diagram



4.11.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Prototype	`PlantComponent (abstract)`	Declares the `clone()` interface for all plant objects; defines copy constructor for member-wise copying of flyweights and decorator chains.
ConcretePrototype	`Succulent`	Implements `clone()` to create copies of Succulent instances; preserves drought-tolerance characteristics and all decorations.
ConcretePrototype	`Shrub`	Implements `clone()` to create copies of Shrub instances; maintains moderate care requirements and all applied decorations.
ConcretePrototype	`Tree`	Implements `clone()` to create copies of Tree instances; preserves high-care requirements and premium decorations.
ConcretePrototype	`Herb`	Implements `clone()` to create copies of Herb instances; maintains moderate care needs and kitchen-themed decorations.
Decorator Classes	`PlantDecorator subclasses (e.g., SmallFlowers, LargeLeaf)`	Implement `clone()` to wrap and copy the decorated component; preserve layered decorations during cloning.
Client	`Builder, Inventory system`	Uses `clone()` to create new plant instances from established prototypes without rebuilding from scratch.

4.11.5 Implementation Notes

Important Functions:

- **clone():** Deep copies entire plant object preserving all attributes, strategies, states, and decorator chains; enables rapid inventory expansion from prototypes.
- **Copy Constructor:** Performs shallow copy of flyweight pointers (safe due to immutability) while deep-cloning decorator chain via recursive `nextComponent->clone()`.

Implementation Challenges:

- **LivingPlant Structure Refactoring:** Deep cloning required adding getters, setters, and copy constructor. Solution: Restructured `LivingPlant` with proper decorator management methods; changes cascaded to Composite and Decorator. `nullptr`. This design choice prevents a complete deep copy of the attached decorator chain when cloning the base plant object.
- **Cascading Design Changes:** Cloning updates propagated through architecture (PlantComponent, PlantGroup, PlantAttributes). Solution: Careful coordination across patterns; deep copy constructors for chain cloning.

4.12 Singleton

Classification: Creational — **Strategy:** Delegation (Object)

Intent: Ensure a class has only one instance, and provide a global point of access to it.

4.12.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

- **FR-6: Centralised Inventory** – Maintains a single, shared inventory accessible to both customers and staff through `getInstance()->getInventory()`; ensures no duplicate or conflicting inventory records exist across the system.
- **FR-10: Shared Data Memory Optimisation** – Manages four flyweight factories that minimise memory usage by sharing immutable data objects (strategies and states) across multiple plant instances; reduces memory footprint significantly for large inventories.

Supporting Requirements

- **NFR-4: Scalability** – Central management of shared objects and pre-cached strategies enables support for 5000+ plant instances without memory growth.
- **NFR-5: Reliability** – Single source of truth for all system resources prevents data inconsistencies and race conditions through centralised access.
- **FR-17: Unified System Interface** – Provides centralised access point for all system-wide resources; eliminates need for parameter passing across subsystems.

4.12.2 Pattern Benefits

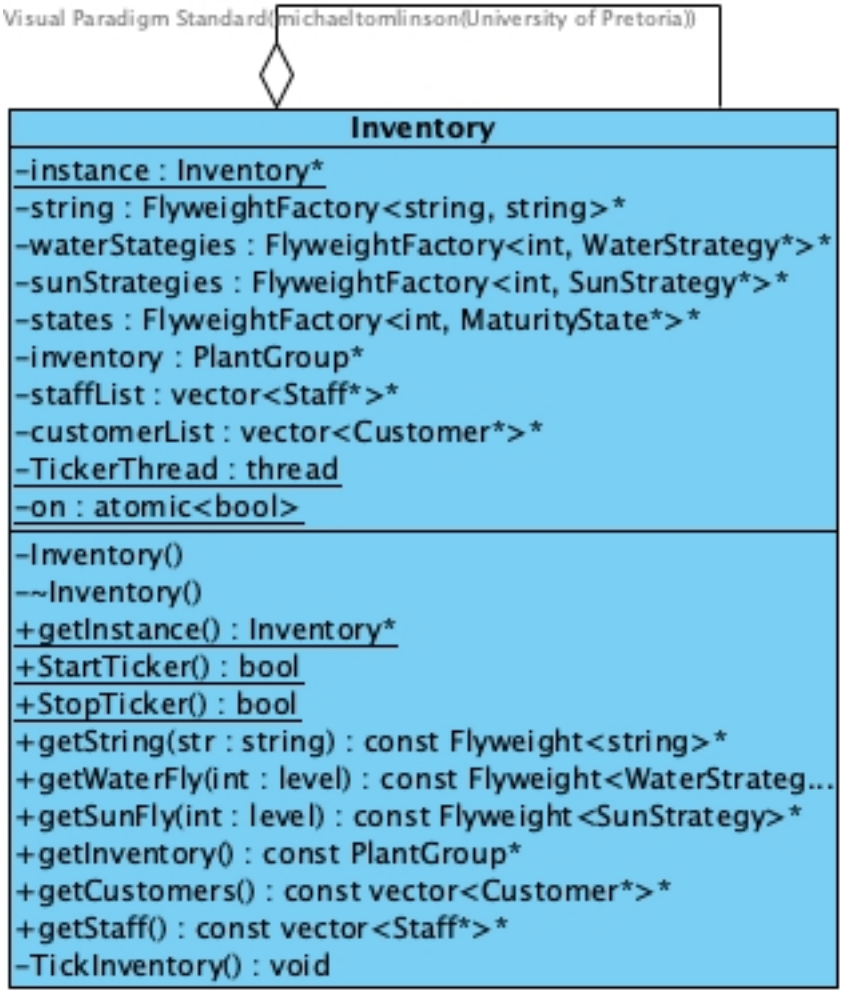
General Benefits:

- Ensures only one instance throughout application lifetime; provides global access without parameter passing.
- Centralises initialisation and resource creation; enables lazy initialisation of expensive resources.
- Simplifies resource management and lifecycle control; prevents inconsistent state.

Implementation Benefits:

- Manages four factories with significant memory reduction; owns inventory composite for hierarchical organisation.
- Maintains staff/customer lists for Observer/Mediator integration; hosts background ticker for autonomous simulation.
- Eliminates duplicate initialisation of expensive resources; prevents conflicting inventory instances.

4.12.3 Class Diagram



4.12.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Singleton	`Inventory`	Ensures single global instance via `getInstance()`; manages lifecycle of all system resources including inventory composite, four flyweight factories, staff/-customer lists, and background ticker thread.
Managed Inventory	`PlantGroup* inventory`	Root composite node managing entire hierarchical plant organisation; accessed via `getInventory()`.
Flyweight Factories	• `FlyweightFactory<int, WaterStrategy*>` • `FlyweightFactory<int, SunStrategy*>` • `FlyweightFactory<int, MaturityState*>` • `FlyweightFactory<string, string*>`	Manage and cache 8 water/sun strategies, 4 lifecycle states, and season name strings; reduce memory footprint by sharing immutable data across plant instances via `getWaterFly()`, `getSunFly()`, `getStates()`, `getString()`.
System Actors	• `vector<Staff*>* staffList` • `vector<Customer*>* customerList`	Lists maintained by singleton for Observer and Mediator pattern integration; accessed via `getStaff()`, `getCustomers()`, `addStaff()`, `addCustomer()`.
Background Thread	• `thread* TickerThread` • `atomic<bool> on`	Manages automated simulation via `startTicker()`, `stopTicker()`, `TickInventory()`; updates plant states every 5 seconds independently of client code.

4.12.5 Implementation Notes

Important Functions:

- **getInstance():** Lazy initialisation of singleton; thread-safe creation on first call via static instance pattern.
- **startTicker()/stopTicker():** Manages background simulation thread lifecycle; `startTicker()` creates thread with proper startup checks, `stopTicker()` gracefully shuts down via atomic flag and join.
- **TickInventory():** Background thread method; periodically calls `tick()` on inventory composite every 5 seconds while checking atomic control flag to enable clean termination without race conditions.
- **Destructor:** Implements strict cleanup sequence: stops ticker thread first, then deletes inventory composite, then factories (which cascade-delete flyweights), finally actors; prevents use-after-delete.

Implementation Challenges:

- **Memory Management Plan:** Singleton owns objects (factories, inventory, staff/customers) that other classes reference. Solution: Created sequence diagrams documenting lifelines; destructor follows strict cleanup order (thread → inventory → factories → actors).
- **Background Thread Integration:** Thread must run autonomously without interfering with main operations. Solution: Used `std::atomic<bool>` flags for race-free shutdown; thread checks flag in loop for clean termination.
- **Cyclic Dependencies:** Singleton references all classes; all classes access singleton. Solution: Forward declarations in headers, definitions in .cpp files, proper pointer management eliminates circular includes.

4.13 State

Classification: Behavioural — **Strategy:** Delegation (Object)

Intent: Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

4.13.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

- **FR-3: Plant Lifecycle Management** - Manages plant maturity states with minimum ages, growth rates; enables automatic transitions between states based on age and attributes

Supporting Requirements

- **NFR-2: Maintainability/Extensibility** - New maturity states can be added without modifying existing states or the plant class
- **FR-8: Inventory Tracking and Notifications** - State changes trigger observer notifications to alert staff of lifecycle transitions

4.13.2 Pattern Benefits

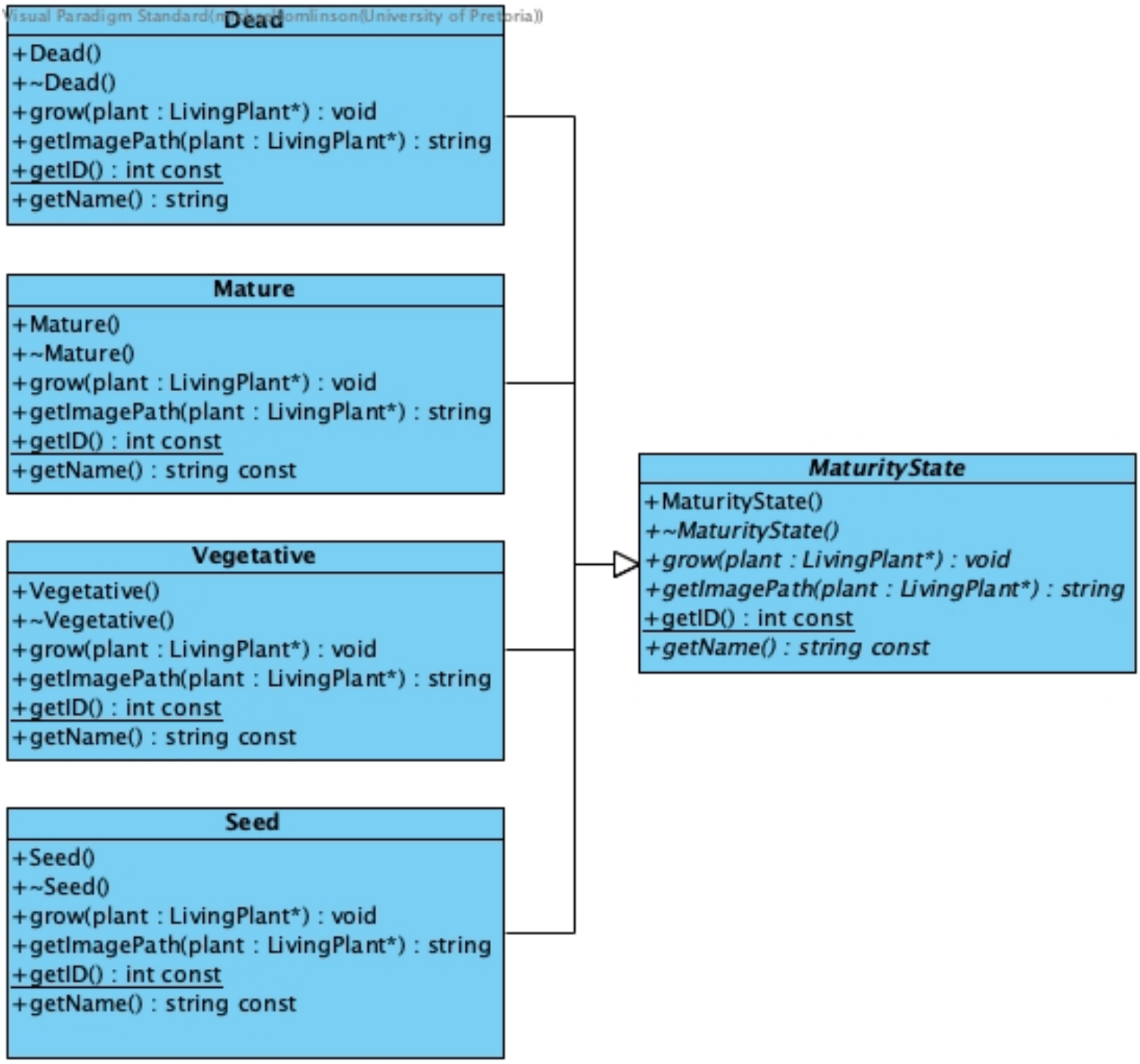
General Benefits:

- Localises state-specific behaviour into separate classes; eliminates large conditional statements.
- Makes state transitions explicit and easier to manage; enables encapsulation per lifecycle stage.

Implementation Benefits:

- Handles seasonal influences dynamically; integrates with observers for transition notifications.
- Supports price dynamics and care requirement changes per state.

4.13.3 Class Diagram



4.13.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Context	`LivingPlant`	Maintains a reference to the current MaturityState and delegates state-dependent behaviour.
State	`MaturityState` (`abstract`)	Defines the interface for encapsulating behaviour associated with a particular maturity stage.
ConcreteState	• `Seed` • `Vegetative` • `Mature` • `Dead`	Implements specific growth behaviour and handles state transitions based on plant age.

4.13.5 Implementation Notes

Important Functions:

- **grow():** Implements state-specific growth behaviour including age increment, seasonal multiplier application, water usage calculation, and health updates.

Implementation Challenges:

- **Seasonal Influence Integration:** Apply seasonal multipliers to water usage without complicating state logic. Solution: Separate multiplier application from core growth behaviour.
- **State Transition Validation:** Ensure one-way progression with terminal Dead state. Solution: Age-based thresholds with explicit condition checks prevent invalid transitions.

4.14 Strategy

Classification: Behavioural — **Strategy:** Delegation (Object)

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

4.14.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

- **FR-4: Watering Management** – Implements different watering approaches defining water amount and frequency for different plant types; allows strategy changes based on plant lifecycle
- **FR-5: Sunlight Management** – Implements different sunlight exposure approaches defining intensity and duration for different plant types

Supporting Requirements

- **NFR-2: Maintainability/Extensibility** – New care strategies can be added without modifying plant classes
- **FR-15: Staff Action Execution** – Enables staff to execute watering and care routines through strategy execution

4.14.2 Pattern Benefits

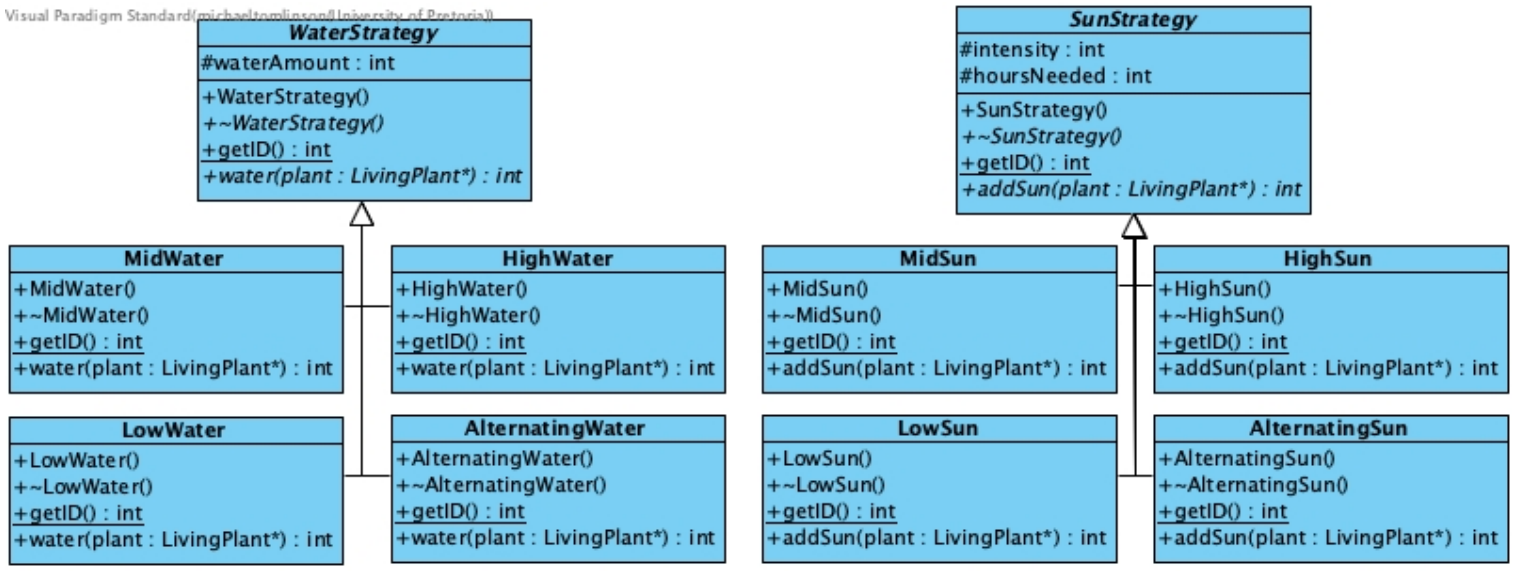
General Benefits:

- Defines family of algorithms and makes them interchangeable; varies independently from clients.
- Supports open/closed principle enabling extension without modification; eliminates conditionals.

Implementation Benefits:

- Enables dynamic care adjustments based on lifecycle; optimises memory via flyweight sharing.
- Facilitates staff actions and composite operations on plant groups.

4.14.3 Class Diagram



4.14.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Context	`LivingPlant`	Maintains references to `WaterStrategy` and `SunStrategy` and delegates care operations to them.
Strategy	`WaterStrategy` (abstract), `SunStrategy` (abstract)	Define interfaces for care algorithm families.
ConcreteStrategy	• Water: `LowWater`, `MidWater`, `HighWater`, `AlternatingWater` • Sun: `LowSun`, `MidSun`, `HighSun`, `AlternatingSun`	Implement specific care algorithms that directly modify plant's `waterLevel` and `sunExposure` attributes with varying amounts, frequencies, intensity, and duration.

4.14.5 Implementation Notes

Important Functions:

- **addSun(LivingPlant*:plant)/addWater(LivingPlant*:plant):** Concrete strategy methods implementing specific care algorithms; each strategy variant modifies plant attributes (waterLevel/-sunExposure) with different amounts, frequencies, or intensities.

Implementation Challenges:

- **Runtime Strategy Switching:** Enable dynamic strategy changes based on lifecycle. Solution: Setter methods allow runtime modification without recompilation.
- **Memory Efficiency via Flyweight:** Strategies are shared across all plants. Solution: Integrated with Flyweight factory for immutable shared instances.