2.3 PATTERN NAME *(Classi cation: Creational/Structural/Behavioural, Strategy: Delegation/Composition/Inheritance)*

Justification and Motivation: 1-2 sentence explanation of why this pattern was chosen for the system and where it is used.

2.3.1 Functional and Non-Functional Requirement Mapping

Primary Requirements

FR-X: Requirement Name - Brief description of how this pattern enables or supports this functional requirement; specific functionality it provides
FR-Y: Requirement Name - Brief description of how this pattern enables or supports this functional requirement; specific functionality it provides

Supporting Requirements

NFR-X: Quality Attribute - Brief description of how this pattern supports this non-functional requirement and what quality it improves
FR-Z: Requirement Name - How this pattern works with other requirements or subsystems

2.3.2 Pattern Benefits

General Benefits: General benefits of using this design pattern, independent of the specific context

Benefit 1
Benefit 2
Benefit 3

Implementation Benefits: Specific benefits and advantages this pattern brings to the Plant Nursery system

Specific benefit 1 for the nursery system
Specific benefit 2 for the nursery system
Specific benefit 3 for the nursery system

2.3.3 Class Diagram

Insert partial UML class diagram here

2.3.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Role Name 1	Class1, Class2	Description of what this role/class does in the pattern
Role Name 2	Class3	Description of what this role/class does in the pattern
Role Name 3	Class4, Class5, Class6	Description of what this role/class does in the pattern
Role Name 4	Class7	Description of what this role/class does in the pattern

2.3.5 Implementation Notes

Important Functions: List and briefly describe the key functions/methods that implement this pattern

ImportantFunction1(): Description of what this function does
ImportantFunction2(): Description of what this function does

ImportantFunction3(): Description of what this function does
ImportantFunction4(): Description of what this function does

Implementation Challenges: Describe challenges encountered during implementation and how they were addressed

Challenge 1 and how it was resolved
Challenge 2 and how it was resolved
Challenge 3 and how it was resolved

Flow Example: Narrative description or sequence of steps showing how the pattern flows during a typical operation in the nursery system. For example: "When a customer requests a seasonal plant filter..."

Step-by-step description of the flow:

1. Step 1 description
2. Step 2 description
3. Step 3 description
4. Step 4 description

Calling Example: Pseudo-code or C++ code example showing how the pattern is used in practice

```
Example code or pseudo-code showing how the pattern is instantiated and used
```

2.4 Iterator *(Classi cation: Behavioural, Strategy: Delegation)*

Justification and Motivation: The Iterator pattern provides sequential access to plant collections with filtering based on seasons or other criteria, without exposing internal collection structures. It supports traversal of nested composite hierarchies through a stack-based algorithm, allowing flexible browsing and seasonal filtering for both customers and sta .

2.4.1 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 Organization – Supports stack-based traversal through nested `PlantGroup` structures.

Supporting Requirements

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

2.4.2 Pattern Benefits

General Benefits: General benefits of using this design pattern, independent of the specific context

Hides collection structure, encapsulating traversal logic.
Supports multiple concurrent traversals via independent iterators.
Enables filtering and transformation during iteration.
Simplifies client code and eliminates index management.

Implementation Benefits: Specific benefits and advantages this pattern brings to the Plant Nursery system

Supports seasonal filtering without exposing internal lists.
Handles arbitrarily nested groups with stack-based iteration.
Extensible filtering criteria (season, type, health).
Achieves amortized O(1) `next()` via cached traversal stack.
Decouples traversal logic from business rules.

2.4.3 Class Diagram

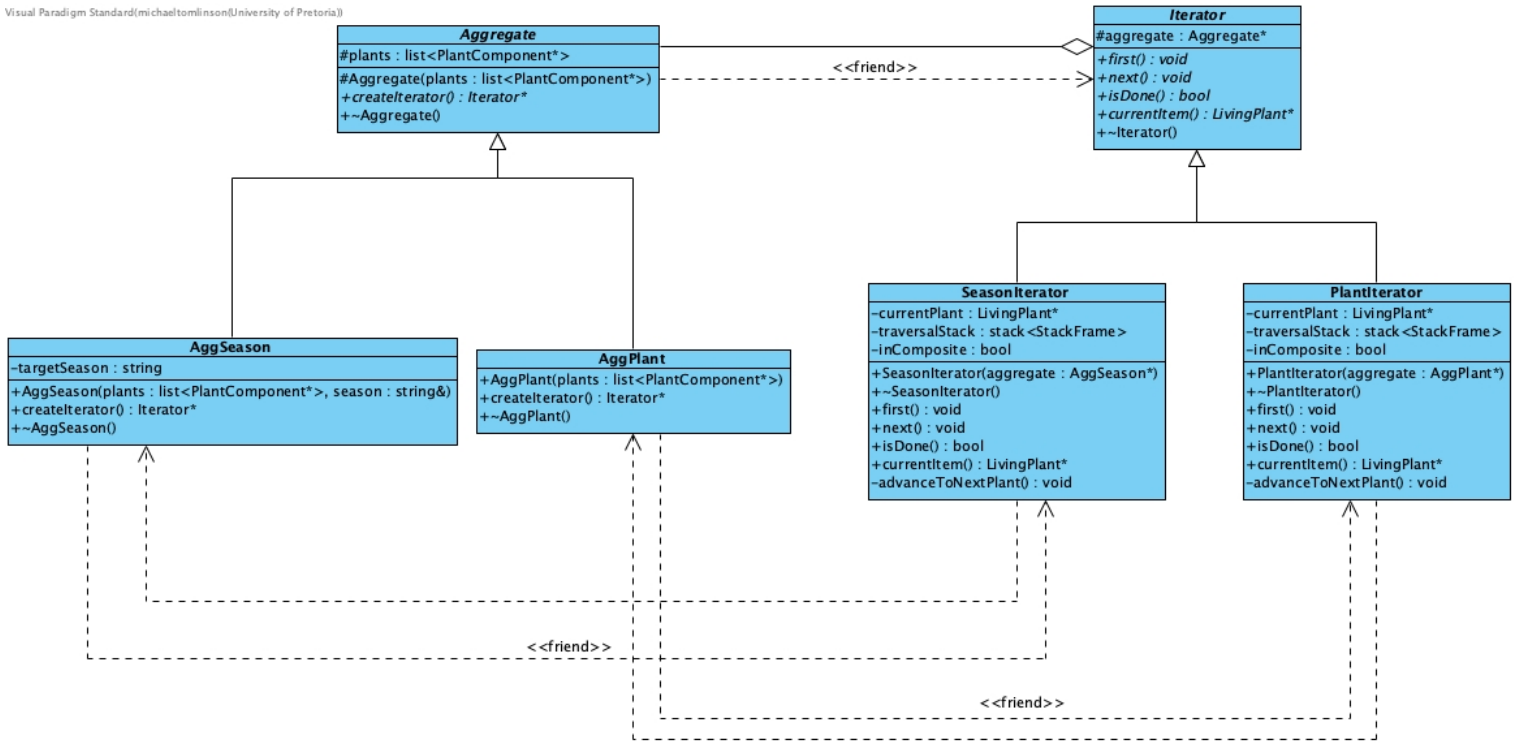

Figure 1: Iterator Class Diagram

2.4.4 Participants Mapping and Responsibilities

Pattern Role	Photosyntech Class(es)	Responsibility
Iterator	`Iterator (abstract)`	Defines traversal interface (`first()`, `next()`, `isDone()`, `currentItem()`).
ConcreteIterator	`PlantIterator`, `SeasonIterator`	Implements stack-based traversal; applies filtering logic via `advanceToNextPlant()`.
Aggregate	`Aggregate (abstract)`	Declares factory method `createIterator()`.
ConcreteAggregate	`AggPlant, AggSeason`	Create iterators; `AggSeason` uses flyweight season pointer.
Collection	`PlantGroup`, `list<PlantComponent*>`	Composite plant hierarchy being traversed.

2.4.5 Implementation Notes

Important Functions: List and briefly describe the key functions/methods that implement this pattern

first(): Initializes stack and positions at first match.
next(): Advances to next element via cached stack.
isDone(): True when traversal completes.
currentItem(): Returns cached current plant.
advanceToNextPlant(): Performs stack-based traversal.
getType(): Returns `ComponentType` enum for efficient type ID.
createIterator(): Factory in aggregates creating iterators with season filters.

Implementation Challenges: Describe challenges encountered during implementation and how they were addressed

Type Identification: Replaced costly `dynamic_cast` with enum-based `getType()` for faster static casting.
Inefficient Recursion: Replaced recursive traversal with stack-based iteration achieving O(1) amortized `next()`.
Code Duplication: Unified seasonal iterators into one generic `SeasonIterator` using flyweight season pointer.

Flow Example: Narrative description or sequence of steps showing how the pattern flows during a typical operation in the nursery system. For example: "When a customer requests a seasonal plant filter..."

Step-by-step description of the flow:

1. Client requests seasonal filter (`AggSeason(collection, seasonFlyweight)`).
2. `createIterator()` returns configured `SeasonIterator`.
3. `first()` pushes root frame to stack and calls `advanceToNextPlant()`.
4. Stack-driven traversal explores nested groups and filters matching `LivingPlant`s.
5. `isDone()` signals completion when stack empties.

Calling Example: Pseudo-code or C++ code example showing how the pattern is used in practice

```
// Unfiltered traversal
AggPlant allPlantsAgg(collection);
Iterator* it = allPlantsAgg.createIterator();
for (it->first();  it->isDone(); it->next())
    std::cout << it->currentItem()->getName() << std::endl;

// Seasonal filtering using Flyweight
auto spring = SeasonFlyweightFactory::getInstance()->getSeason("Spring");
AggSeason springAgg(collection, spring);
Iterator* sit = springAgg.createIterator();
for (sit->first();  sit->isDone(); sit->next())
    std::cout << sit->currentItem()->getName() << " (Spring)" << std::endl;
```