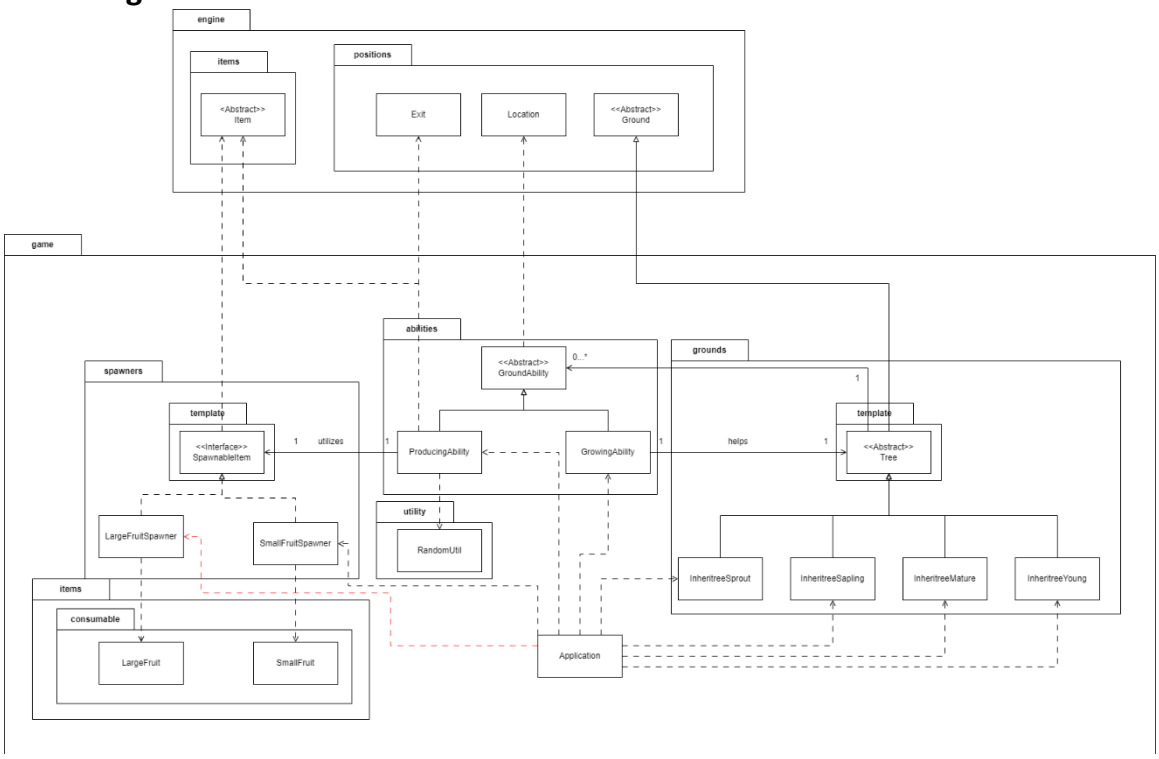
Assignment 3 Design Rationale

Requirement 4: Refactoria, Connascence's largest moon.

UML Diagram:



Design Goal:

The design goals for this assignment are to incorporating a new feature and refactoring the design of the Tree class. The new feature refers to two new Tree classes, which are InheritreeYoung and InheritreeSprout. Nevertheless, the Inheritree in Refactoria will have different growth stage. It will grow up from Sprout, Sapling, Young then Mature while the growth stage of Inheritree in Polymorphia is Sapling to Mature. The purpose of refactoring the Tree class is improving its extensibility and reducing the coupling. These design goals will be implemented while adhering closely to relevant design principles and best practices.

Design Decision:

Before implementing the new feature and refactoring the design of the Tree classes, I decided to create GroundAbility abstract. After that, I created two GroundAbility subclasses, which were ProducingAbility and GrowingAbility to represent the ability of producing fruit and growing up. In the previous design,

the relevant methods were inside the Tree abstract class such as dropFruit. However, some trees were not able to drop fruit, such as sprout and young. Therefore, I decided to move these methods out of the Tree abstract class and created relevant GroundAbility classes to represent it. The reason I changed it to class instead of interface is the logic of the method is same for all trees. If I changed it to interface, the implementations of the method are same in different Tree classes and causing code duplication. Not only this consideration but also the consideration of flexibility. Future developers could just create the subclass of GroundAbility class without modifying the existing code. Besides that, this design also improved the maintainability since the task of each Tree class become smaller and simpler. At the same time, I was required to refactor the Tree abstract class so it can store the object of GroundAbility. Considering that the abilities have different priorities, it is logical to add a new attribute to the Tree abstract class which is Map instead of arraylist. For example, the priority of the GrowingAbility is higher than the ProducingAbility.

Nevertheless, I also decided to create the SpawnableItem interface and the classes that implement it which are LargeFruitSpawner and SmallFruitSpawner. The approach of this design is to avoid all the fruit objects share the same state. These spawner classes' task is constructing a new fruit object. Therefore, the maintainability is improved because the task of the ProducingAbility is divided into smaller. Now the ProducingAbility just focuses on the chance and choose the place to put the fruit object while the Spawner class focuses on creating the fruit object.

In implementing the new feature, I decided to create two new classes, which are InheritreeSprout and InheritreeYoung. After that, I let these two classes extending the Tree abstract class to reuse the code of the Tree abstract class. Besides that, the maintainability is enhanced because the changes to the Tree abstract class automatically propagate to its subclasses, making it easier to maintain consistency and avoid inconsistencies across different tree types.

Pros:

1. Separation of Concerns: The design separates different responsibilities, such as producing fruit and growing, into distinct classes (ProducingAbility and GrowingAbility). This makes each class simpler and focused on a single responsibility, enhancing maintainability and readability.

2. Extensibility: By using abstract classes (GroundAbility) and interfaces (SpawnableItem), the design is open for extension. New abilities or types of

spawners can be added without modifying existing code, adhering to the Open/Closed Principle (OCP).

3. Flexibility: The design allows for different abilities to be prioritized using a Map in the Tree class. This flexibility enables easy adjustments to the behavior of tree instances by simply changing the abilities and their priorities.

4.Maintainability: Centralizing the logic for abilities in their respective classes reduces code duplication and makes it easier to update or fix bugs in one place. Changes in the Tree abstract class automatically propagate to its subclasses, ensuring consistency.

Cons:

-**Code Duplication**: There might still be duplication if different Tree subclasses share similar logic but cannot inherit it directly due to slight variations. This can lead to repeated code across subclasses.

-**Connascence**

1. Connascence of Name: The design might suffer from connascence of name if multiple classes need to know the exact names of methods or attributes in other classes. This can make refactoring more challenging since renaming a method would require changes in all dependent classes.

2.Connascence of Position: The order of the Map will affect the performance of the Tree. For example, the priority of ProducingAbility is higher than GrowingAbility.

-**Code smell**

1.Shotgun Surgery: Although the design aims to reduce code duplication, modifying certain behaviors might still require changes in multiple places. For instance, changing the logic for ProducingAbility or GrowingAbility may necessitate updates in various Tree subclasses that use these abilities.

2. Speculative Generality: If there are many abilities or spawners that are never used or rarely used, this can indicate speculative generality. This is a situation where the system is designed to handle cases that may never occur, adding unnecessary complexity.

Alternative Design:

One alternative design is creating the ability interfaces such as ProducingFruit interface, GrowingUp interface and let the Tree classes implementing it. However, just like I mentioned before, the logic of the method is same for all the tree type, which means the implementation is almost the same. Using the interface will cause the code duplication because we need to write the same code in different tree classes. Besides that, it is hard to maintain because we need to visit the tree classes that implementing the interface if the logic of the method of the interface needs some changes. Moreover, another approach for the dropping the fruit is always dropping the same fruit object. It is not logical and bad because if the fruit object involves randomness, alternative design is unable to handle it. Although it has several disadvantages, it is simple and easy to implement comparing to chosen design.

Pros:

1.Simplicity: This design is straightforward to implement because it does not involve multiple layers of abstraction or complex dependency management. Besides that, the logic for producing fruit and growing up is directly implemented in the Tree classes, making it easy to understand for someone new to the codebase.

Cons:

1.Code Duplication: Logic is repeated across multiple Tree classes, making the code harder to maintain and update.

2. Hard to Maintain: Changes require updates across multiple classes, increasing the risk of errors and inconsistencies.

-Connascence
1. Connascence of Name: If the method names for producing fruit or growing up change, all implementing classes need to be updated, increasing the likelihood of errors.

2.Connascence of Execution: The tick method will implement the method of the interface. However, the methods of different interfaces have different priorities, the order of the method inside the tick method should be correct else it will not work as our expectation.

-Code smell

1. Rigidity: The design can be rigid, making it difficult to extend with new functionality. Adding new behaviors or modifying existing ones requires changes to multiple classes, making the system less adaptable to change. For example, the tick method. If the tree class implementing new interface, we need to modify the tick method so it will implement the method of the new interface.

Analysis of Alternative Design:

The alternative design is not ideal because it violates various Design and SOLID principles:

1. **Open/Closed Principle:**

   - While this allows for polymorphism, it means that any changes to the logic of producing fruit or growing up require modifying each Tree class that implements these interfaces. Besides that, whenever we need to change the behavior for producing fruit or growing up (e.g., changing the algorithm or adding new features), we have to modify every Tree class that implements these behaviors.

2. **DRY:**

   - The logic for methods like dropFruit or grow would be written in each tree class even though it's the same. This creates unnecessary code repetition and increases the overall size of the codebase.

Final Design:

In our design, we closely adhere to Single Responsibility Principle, Open/Closed Principle and Dependency Inversion Principle:

1. **Single Responsibility Principle:**

   - **Application:** The Tree class focuses on its core tree properties. The other tasks such as producing fruit, growing up are handled by ability classes. Moreover, for the ProducingAbility, the task of constructing fruit object is handled by the fruitSpawner class. The task of the

ProducingAbility is deciding the chance and choosing the location drop the fruit object.

- **Why:** Adhering to SRP improves the maintainability and code reusability.
- **Pros/Cons:**
  Pros:
  -Separation of Concerns: The design separates different responsibilities, such as producing fruit and growing, into distinct classes (ProducingAbility and GrowingAbility). This makes each class simpler and focused on a single responsibility, enhancing maintainability and readability.

  - Maintainability: Centralizing the logic for abilities in their respective classes reduces code duplication and makes it easier to update or fix bugs in one place. Changes in the Tree abstract class automatically propagate to its subclasses, ensuring consistency.

  Cons:

  - **Code Duplication**: There might still be duplication if different Tree subclasses share similar logic but cannot inherit it directly due to slight variations. This can lead to repeated code across subclasses.

  - Shotgun Surgery: Although the design aims to reduce code duplication, modifying certain behaviors might still require changes in multiple places. For instance, changing the logic for ProducingAbility or GrowingAbility may necessitate updates in various Tree subclasses that use these abilities.

  - Speculative Generality: If there are many abilities or spawners that are never used or rarely used, this can indicate speculative generality. This is a situation where the system is designed to handle cases that may never occur, adding unnecessary complexity.

  -Connascence of Position: The order of the Map will affect the performance of the Tree. For example, the priority of ProducingAbility is higher than GrowingAbility.

2. **Open/Closed Principle:**

- **Application:** Abstract classes like GroundAbility, Tree, we can add new ability class or new tree type without modifying existing classes. Same goes for the Interface, SpawnableItem.
- **Why:** Adhering this principle has improved the code promotes loose coupling and reduces the risk of introducing bugs when adding new feature. Besides that, it improves extendibility as we can extend the functionality without rewriting the existing code.
- **Pros/Cons:**
  Pros:

  -Extensibility: By using abstract classes (GroundAbility) and interfaces (SpawnableItem), the design is open for extension. New abilities or types of spawners can be added without modifying existing code.

3. **Dependency Inversion Principle:**

- **Application:** The Tree class is having relationship with the GroundAbility abstract class instead of all the subclasses of GrounAbility. Similarly, the ProducingAbility is having relationship with the SpawnableItem instead of the class that implementing this interface.
- **Why:** Adhering to this principle has enhanced the extendibility. If future developers add new ability or new tree type, they could just let the new classes extending the abstract class. As long as the new ability or new tree type extending their abstract classes, they do not need to modify any existing code to add this new feature. It makes the process of extending functionality becomes easy and simple.
- **Pros/Cons:**
  Pros:
  -Flexibility: The design allows for different abilities to be prioritized using a Map in the Tree class. This flexibility enables easy adjustments

to the behavior of tree instances by simply changing the abilities and their priorities.

Cons:
-Connascence of Name: The design might suffer from connascence of name if multiple classes need to know the exact names of methods or attributes in other classes. This can make refactoring more challenging since renaming a method would require changes in all dependent classes.

Conclusion:
Overall, our chosen design provides a robust framework for adding the two new Tree classes, which are InheritreeSprout and InheritreeYoung.Besides that, our chosen design allows the Inheritree in Polymorphia and Refactoria has different growth stages. By carefully considering connascence, SOLID principles, we have developed a solution that has high maintainability, easy to extend and flexible.