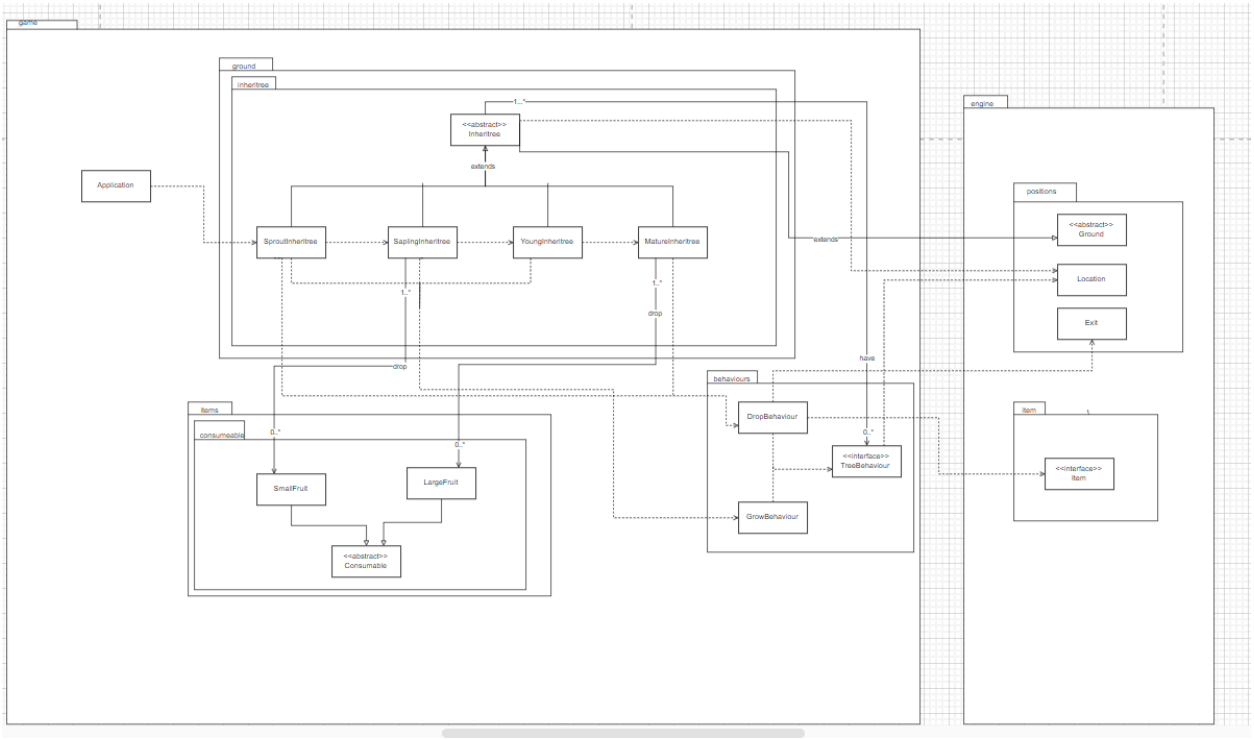
DESIGN RATIONALE

UML



DESIGN GOAL

The task's design objective is to develop a dynamic and interactive tree growth system for the game environment in Refactorio on the moon Connasence . The Inheritrees begin as sprouts and advance through different growth stages, each with unique actions like growing and dropping fruits. This approach improves gameplay by offering a realistic and captivating ecological system while following object-oriented programming principles.

DESIGN DECISION

The strategy for fulfilling this need involved creating a hierarchical arrangement of inherited classes that depict various stages of growth. Additionally, the TreeBehaviour interface was used to encapsulate the actions related to fruit growth and dropping

1. **TREE BEHAVIOUR** :The TreeBehaviour Interface outlines a standard agreement for various behaviors displayed by an Inheritree, guaranteeing the implementation of the execute method in all specific behaviors. This supports polymorphism and facilitates code reusability.

2. **DROP BEHAVIOUR** :This class is responsible for implementing the TreeBehaviour interface and managing the process of fruit dropping. It calculates the probability of fruit drops and generates the relevant fruit at a nearby position if the required conditions are fulfilled

3. **GROW BEHAVIOUR** : This class also serves as the TreeBehaviour interface and handles the functionality for tree growth . It monitors the age of the tree and updates its status when it reaches a set number of ticks.
4. **IHERITREE** : This abstract class is the fundamental class for all tree phases, using a Map<Integer, TreeBehaviour> to handle various actions based on priority in a dynamic manner.
5. **SPROUT INHERITREE -** This class depicts the first phase of the tree. Although it is unable to bear fruits, and drop fruits it has the potential to develop into a SaplingInheritree after a set number of ticks.
6. **SAPLING INHERITREE -** This class symbolizes the next phase of the tree. It has the ability to drop and spawn small fruits at a designated chance and transform into a YoungInheritree after a specific number of ticks

7. **YOUNG INHERITREE -**This class indicates the third phase of the tree. While it does not yield fruits, it has the potential to develop into a MatureInheritree after a designated number of ticks.
8. **MATURE INHEITREE -**This class represents the final stage of the tree. It can drop large fruits at a specified rate but cannot grow.

Changes Made

Previously in Assignment 2, there was an Inheritree parent abstract class which had methods to dropFruit to drop the spawned fruit and tick to grow the fruit. As there were only two stages (Mature and Sapling), these methods were overridden in subclasses to specify how many ticks it takes to grow into a new stage and the type of fruit to drop.

However, Assignment 3 introduced new tree stages: Sprout and Young, which grow into Sapling and Mature but do not drop or produce fruits. To handle this complexity, the TreeBehaviour interface was created, along with two behavior classes: DropBehaviour and GrowBehaviour.

The previous Inheritree class was updated to include a TreeMap to help prioritize the behaviors. Each behavior is now added to the respective classes, thereby adhering to SOLID principles. This redesign ensures that the tree growth and fruit-dropping logic is cleanly separated and easily extendable.

ALTERNATE DESIGN

Creating separate classes for each stage of the tree, without deriving from a common superclass, presents an alternative design. This method permits independent modification and enhancement

Analysis of Alternative Design

This alternative design would violate several design principles:

1. **DRY PRINCIPLE:** This method entails replicating identical code in numerous classes, increasing the likelihood of mistakes and adding complexity to maintaining the codebase.
2. **Open/Closed Principle**: Extending functionality would necessitate modifying existing classes instead of extending them, conflicting with the principle of being open for extension but closed for modification.
3. **Single Responsibility Principle**: Failure to separate functionalities (common vs unique) results in each class having more than one reason to change, leading to complications in updates and testing.

Final Design

The implementation follows the following principles:

1. **Single Responsibility Principle (SRP)**: Each class was implemented with only one specific objective. For example, the DropBehaviour class solely handles the fruit dropping logic, while the GrowBehaviour class manages the growth logic.
2. **Open/Closed Principle (OCP)**: The design allows for the addition of new behaviors without modifying existing code. For instance, new tree stages or behaviors can be added by extending the Inheritree class and implementing the TreeBehaviour interface.
3. **Interface Segregation Principle (ISP)**: Each behavior is encapsulated in its own class that implements the TreeBehaviour interface. This ensures that classes are not forced to implement methods they do not use, promoting cleaner and more maintainable code

Connascence

1. **Connascence of Name (CoN)**: The execute method name in the TreeBehaviour interface must be consistently used in all implementing classes.
2. **Connascence of Type (CoT)**: The Inheritree class must correctly interpret the types of behaviors to execute them properly.
3. **Connascence of Timing (CoT)**: The growth of the tree is time-dependent, based on ticks.
4. **Connascence of Algorithm (CoA)**: The drop logic algorithm in DropBehaviour must be consistent to ensure the correct probability of fruit dropping.
5. **Connascence of Value (CoV)**: The drop chance value and the type of fruit to drop must be correctly set in the DropBehaviour class.

Pros

1. **Modularity:**
    - Each stage of the Inheritree (Sprout, Sapling, Young, Mature) and their behaviors (DropBehaviour, GrowBehaviour) are encapsulated within their respective classes. This

modular design allows for easy management and extension without interfering with existing components.

2. **Extensibility:**
    - The design adheres to the Open/Closed Principle (OCP), allowing new stages and behaviors to be added without modifying existing code. This promotes a flexible and scalable system that can easily accommodate new features and changes.
3. **Adherence to SOLID Principles:**
    - The design follows key SOLID principles, such as Single Responsibility Principle (SRP) and Open/Closed Principle (OCP), making the codebase more maintainable, understandable, and less prone to errors.
4. **Dynamic Behavior Management:**
    - The use of a Map<Integer, TreeBehaviour> in the Inheritree class allows for dynamic and prioritized behavior management, making the system adaptable to various in-game scenarios.
5. **Abstraction with TreeBehaviour Interface:**
    - The TreeBehaviour interface abstracts different behaviors, enabling flexibility in extending and modifying tree behaviors without changing the core logic.
6. **Scalability:**
    - The design is inherently scalable. New stages and behaviors can be integrated seamlessly, allowing the game to evolve and grow in complexity without major

Cons

1. **Increased Complexity:**
    - The use of multiple classes and interfaces increases the overall complexity of the system, which can make it harder for new developers to understand and contribute to the codebase.
2. **Performance Overhead:**
    - Each behavior and stage transition involves additional processing, which can lead to performance overhead, especially in a large-scale game environment.
3. **Maintenance Effort:**
    - **The** modular and extensible nature of the design requires ongoing maintenance to ensure all parts of the system remain compatible and function as intended.

Future Extensions

The current design provides a robust framework that can be easily extended to accommodate additional stages or behaviors for the Inheritree classes. Here are some potential future extensions and how the current design facilitates them:

Additional Tree Stages: New stages can be added easily . By implementing new classes that inherit from Inheritree and defining the specific behavior through additional GrowBehaviour or DropBehaviour instances, the new stages can be integrated without modifying existing code.

Different Fruit Types: If new types of fruits need to be introduced, the DropBehaviour class can be extended or modified to handle different fruit types based on new conditions or tree types. This can be achieved by passing the fruit type as a parameter when creating new DropBehaviour instances.

Dynamic Behavior Addition: Additional behaviors can be introduced and prioritized dynamically by extending the TreeBehaviour interface. This flexibility allows for new interactions or actions to be added, enhancing the tree's role in the game.

Conclusion

In conclusion, the design of the Inheritree growth system combines inheritance, interface implementation, and dynamic behavior management to create an engaging and flexible gameplay experience. By adhering to design principles and considering connascence, the system is efficient, scalable, and maintainable, paving the way for future enhancements and optimizations. This approach not only improves the gameplay experience but also aligns with best practices in software design.

.