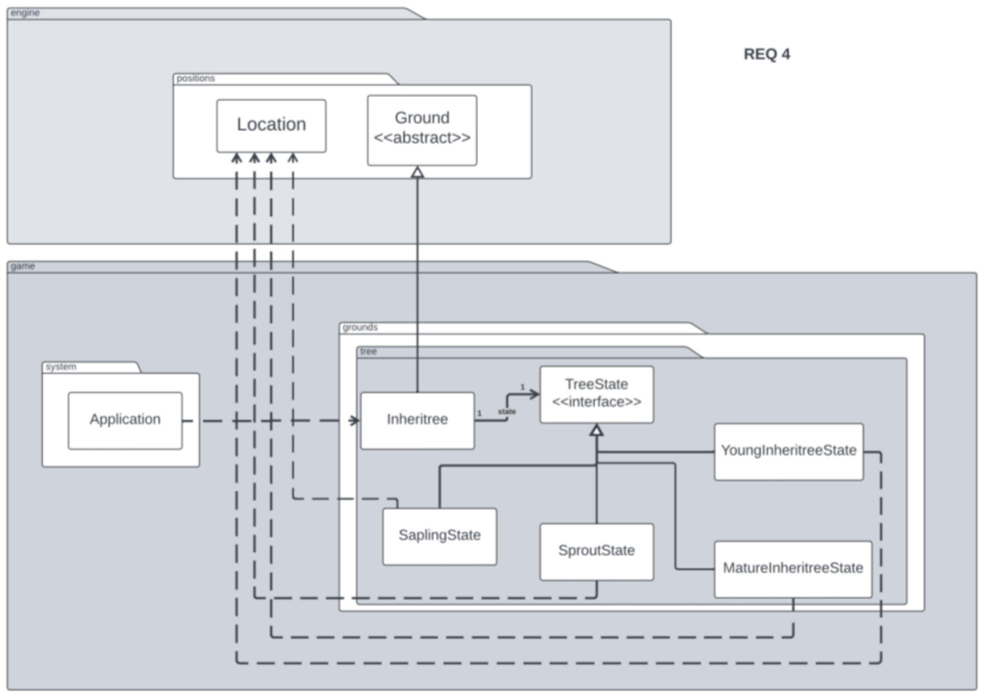
Design Rationale for REQ4: Refactorio, Connascence’s largest moon

Overview

The refactoring requirement focuses on enhancing the functionality of the Inheritree in the game, ensuring it can transition through multiple growth stages (Sprout, Sapling, Young Inheritree, Mature Inheritree) and implement the behaviors associated with each stage. The UML diagram below illustrates the relationships and interactions between the core components necessary to fulfill this requirement:



UML Diagram Explanation

Engine Package:

Positions Folder:

Location Class: Represents a specific position or location in the game world.

Ground Abstract Class: An abstract class representing any type of ground in the game. This class is inherited by specific types of grounds.

Game Package:

System Folder:

Application Class: Represents the main application or the game system that runs the game logic.

Grounds Folder:

Tree Folder:

TreeState Interface: Defines the contract for different states of an Inheritree. Each state must implement methods for handling ticks and returning display characters.

Inheritree Class: Represents the tree that can transition through various states. It holds a reference to the current state (an implementation of TreeState) and delegates behavior to this state.

SproutState Class: Implements TreeState and represents the initial "sprout" stage of the tree.

SaplingState Class: Implements TreeState and represents the "sapling" stage of the tree.

YoungInheritreeState Class: Implements TreeState and represents the "young" stage of the tree.

MatureInheritreeState Class: Implements TreeState and represents the "mature" stage of the tree.

Principles and Concepts

Single Responsibility Principle (SRP):

Each class in the design has a single responsibility. For example, the MatureInheritreeState class only manages the behavior of a mature Inheritree, while the SaplingState class only manages the behavior of a sapling. This separation ensures that changes in one state do not affect others, simplifying maintenance.

Open/Closed Principle (OCP):

The design allows for easy extension without modifying existing code. New tree stages or types of trees can be added by creating new classes that implement the TreeState interface. This design supports scalability and future enhancements, such as adding new states or behaviors without altering the existing structure.

Liskov Substitution Principle (LSP):

The design ensures that instances of Inheritree can be substituted with any state that implements the TreeState interface without altering the program's correctness. This principle ensures that derived classes enhance functionality without changing the existing behavior expected by the client.

Interface Segregation Principle (ISP):

The TreeState interface is small and focused, providing only the necessary methods (tick and getDisplayChar) that all states must implement. This keeps the interface lean and specific to the needs of tree states.

Dependency Inversion Principle (DIP):

The Inheritree class depends on the abstract TreeState interface rather than concrete state classes, promoting flexibility and reducing coupling. This allows Inheritree to work with any state that adheres to the TreeState interface.

Connascence

Connascence of Type:

By using the TreeState interface, the design minimizes direct dependencies on specific classes, making it easier to change implementations without affecting the system. The Inheritree class interacts with the TreeState interface, allowing for flexible state management.

Connascence of Position:

The design minimizes connascence of position by ensuring that changes in one part of the code do not necessitate changes in another. For example, the Inheritree class does not need to know the specifics of each tree state, as it only interacts with the TreeState interface.

Connascence of Name:

Consistent naming conventions improve readability and maintainability. The state classes (SproutState, SaplingState, etc.) are named according to their respective stages, making the code intuitive and easy to understand.

Advantages of the Design

Modularity:

The system is highly modular, with each class and interface having a clear responsibility. This makes the codebase easier to understand, maintain, and extend. Each state class manages its specific behavior, ensuring clean separation of concerns.

Flexibility:

The use of interfaces and abstract classes allows for easy addition of new features without modifying existing code. For example, new tree stages or behaviors can be added by creating new classes that implement the TreeState interface.

Maintainability:

Adherence to SOLID principles ensures that the system is easy to maintain. Each class is small and focused, making it easier to understand and modify. The clear separation of responsibilities reduces the risk of introducing bugs when changes are made.

Testability:

The modular design and use of interfaces make the system easy to test. Each component can be tested in isolation, and mock implementations can be used to test interactions between components. This promotes thorough and effective testing practices.

Disadvantages of the Design

Complexity:

Using multiple interfaces and abstract classes can introduce complexity, especially for new developers who are not familiar with the design patterns used. This can be mitigated with proper documentation and adherence to the KISS (Keep It Simple, Stupid) principle where possible.

Overhead:

The design may introduce some overhead due to the increased number of classes and interfaces. However, this is a trade-off for the benefits of modularity and flexibility. The performance impact is typically negligible in most game scenarios.

Extensibility

Adding New Tree Stages:

The current design achieves the Open/Closed Principle (OCP) effectively. For example, if a new tree stage called "Ancient Inheritree" needs to be added, a new class AncientInheritreeState can be created implementing the TreeState interface. This new class can then be integrated into the growth cycle without altering the existing Inheritree or other state classes.

Adding New Types of Trees:

To add a new type of tree, create a new class that inherits from Ground and implements the TreeState interface. This allows the game to support different types of trees with unique behaviors, further enhancing the game's complexity and depth.

Conclusion

This refactored design adheres to the principles of DRY, SOLID, and minimizes connascence, ensuring a robust, maintainable, and scalable implementation. The design effectively addresses all relevant requirements, providing a clear path for future extensions and modifications. By focusing on the principles and concepts taught in the unit, this design achieves a balance between complexity and flexibility, ensuring a high-quality codebase that can evolve with the game's needs.