Design Rationale

By: MA_AppliedSession1_Group3

REQ 1: The Ancient Woods

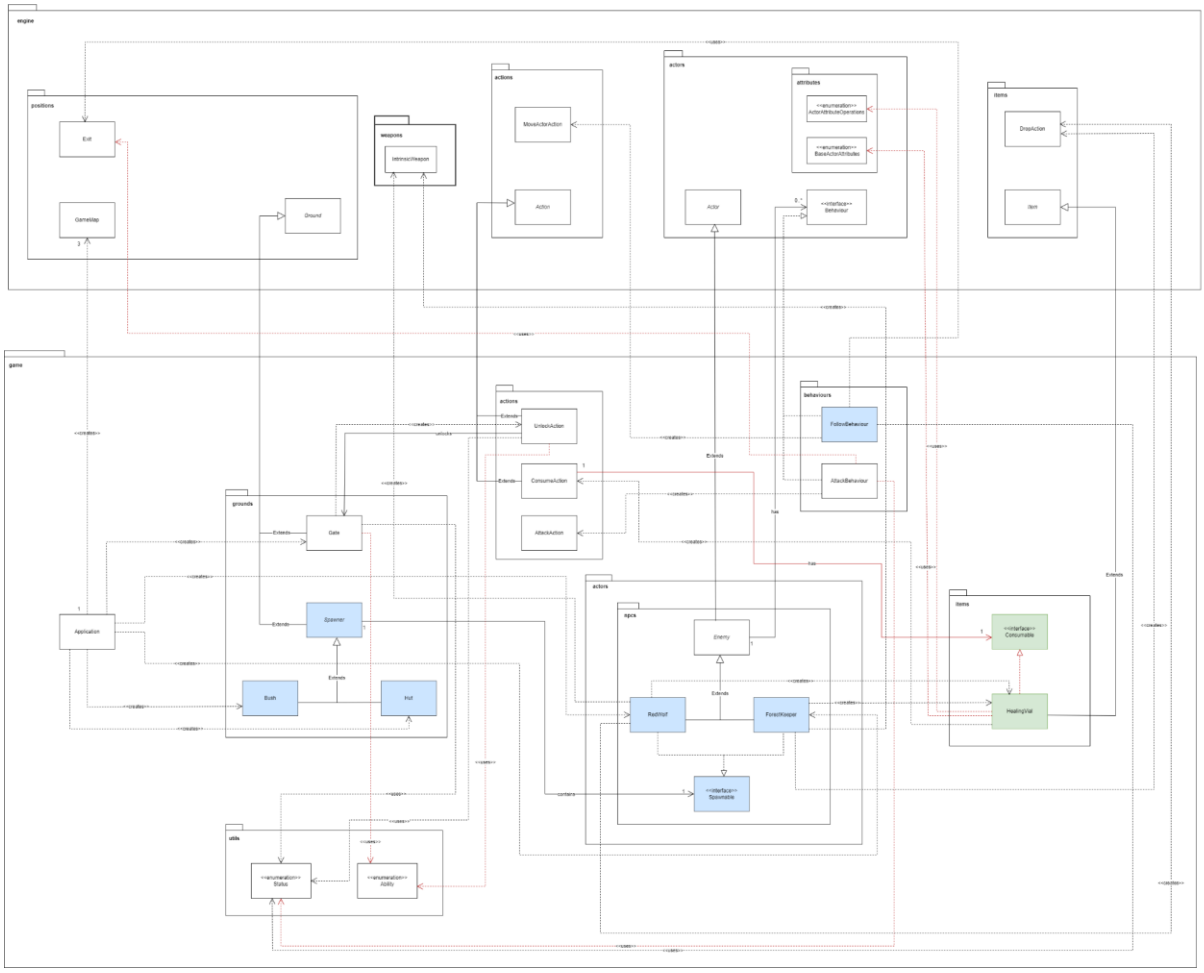

Figure 1: UML class diagram of REQ1 - The Ancient Woods

The provided diagram shows the entire system implementation for Requirement 1, The Ancient Woods. The primary goal of this design is to create a new game map, which is known as Ancient Woods, a gate to travel to the new game map, creating the Bush and Hut in the Ancient Woods to spawn new types of enemies, which are Forest Keeper and Red Wolf at every turn of the game, dropping healing vial item by an enemy once it is defeated. The Red Wolf and Forest Keeper follow the Player until either is unconscious. Thus, new classes are introduced in the extended system: ForestKeeper class, RedWolf class, Bush, Hut, an abstract class, Spawner, an interface class, Spawnable, and FollowBehaviour class. Those newly created classes will interact with the existing classes in the system.

To achieve the goal of the design, we decided to introduce a new Spawner class as an abstract class that extends the abstract Ground class from the engine. All the grounds in the game that can spawn different enemies during each turn will extend this Spawner abstract class because they share common attributes and methods. Grounds in the current implementation, Bush and Hut, will extend this Spawner abstract class as both the Bush class and Hut class can spawn enemies. Since we treated both Bush and Hut classes as types of ground, and both can spawn different enemies, we plan to create a new abstract Spawner class that extends the abstract Ground class, as we considered the Spawner abstract class as a new type of ground which spawn the enemies. Besides, this approach ensures that when additional requirements are introduced in future game implementations that apply to all spawner classes (ground classes that can spawn enemies), these new features can be added to the abstract Spawner class, preventing code repetition (DRY). Importantly, this approach adheres to the Open/Closed Principle (OCP) since the Spawner abstract class extends the functionality of the abstract Ground class from the engine without modifying existing code, as well as Liskov Substitution Principle (LSP), where a subclass can do anything its base class do and it does not disrupt the behaviour of our system. Additionally, the abstract Spawner class has an association relationship with the Spawnable interface class to spawn different enemies at certain chances during each turn of the game. This design choice reduces multiple dependencies (ReD) on various classes that may be implemented in the future, such as the RedWolf and ForestKeeper classes.

We also introduced two new enemies' classes into the extended system, RedWolf class and ForestKeeper class. Both classes extend with the existing Enemy abstract class, mentioned in the previous assignment 1, since both share common attributes and methods. This design choice aligns with the Don't Repeat Yourself (DRY) principle. Since all enemies, regardless Red Wolf or Forest Keeper in the Ancient Woods, can follow the player, it means they share the same behaviour: follow behaviour. Therefore, we had introduced a new FollowBehaviour class, which implement the Behaviour interface class from the engine. The FollowBehaviour class will handle any follow actor actions that are executed by the enemies in Ancient Woods, so this FollowBehaviour class creates MoveActorAction class to perform follow actor action, indicating there is a dependency relationship between FollowBehaviour class and MoveActorAction class from the engine. Our design is not only can reduce multiple dependencies between enemies and their behaviours, but it also aligns with the Dependency Inversion Principle (DIP) where follow behaviour depends on abstractions instead of concrete implementations. We also have the FollowBehaviour class to have a dependency relationship with the Status enum class to check every actor it encounters before performing any actions. Therefore, Red Wolf and Forest Keeper can check the status of an actor they want to follow is accurate, which is the Player. Alternatively, if the Status enum class does not be used, we will need to implement if-else statement and instanceof operator (checking class type) in FollowBehaviour class to ensure that enemies follow the which involves extra dependencies and violates the Open/Closed Principle (OCP) because when new enemies are added, the existing code of the

FollowBehaviour class have to be modified to ensure the condition of enemies not following other actors except the Player holds true at all times. Besides, the FollowBehaviour class has dependency with the Exit class to determine its surrounding contains any actors. In addition, the RedWolf and ForestKeeper classes also create (dependency relationship) DropAction and HealingVial since they will drop healing vial once defeated.

Moreover, the Spawnable interface class, which will be implemented by RedWolf and ForestKeeper classes, is introduced. We decided to design Spawnable as an interface class as it stores a collection of abstract methods for spawning new enemies itself to be implemented in the Enemy subclasses in the later implementation. Instead of defining an abstract method to spawn itself in the Enemy abstract class, which is potentially leading to unnecessary implementations in some subclasses, we can have Enemy subclasses that can spawn itself, which are introduced in future game implementations, can implement it. This design decision aligns with the Interface Segregation Principle (ISP), where the classes that implement this Spawnable interface only need interface methods. This approach also makes our system more extensible and maintainable.

The Application class creates (dependency relationship) a new GameMap class and Gate class, so the Application class has a dependency relationship with the GameMap class and the Gate class. Besides, the Application class also creates (dependency relationship) the Bush class and Hut class as well as the instance of RedWolf and ForestKeeper class.

On the other hand, our design may increase the system's complexity as many new classes are introduced and increase the developers' difficulty if any errors occur during development.

Modification done on classes created in Assignment 1 for REQ 1

A few changes have been made to classes from the previous Assignment 1 interacting with new classes in the requirement 1, specifically, the Healing Vial class and the Consumable interface class. In the last assignment, we have Consumable class as an abstract class. To make our system more extendibility and satisfy this requirement, we decide to modify Consumable class to an interface class. This interface will be implemented by classes that only need to be consumed. This aligns with the Interface Segregation Principle (ISP). Besides, HealingVial class that extends from Consumable abstract class previously will now be implementing the Consumable interface class since it is a consumable item. HealingVial class will no longer extend from the previous Consumable abstract class, now it extends the Item abstract class from the engine directly.