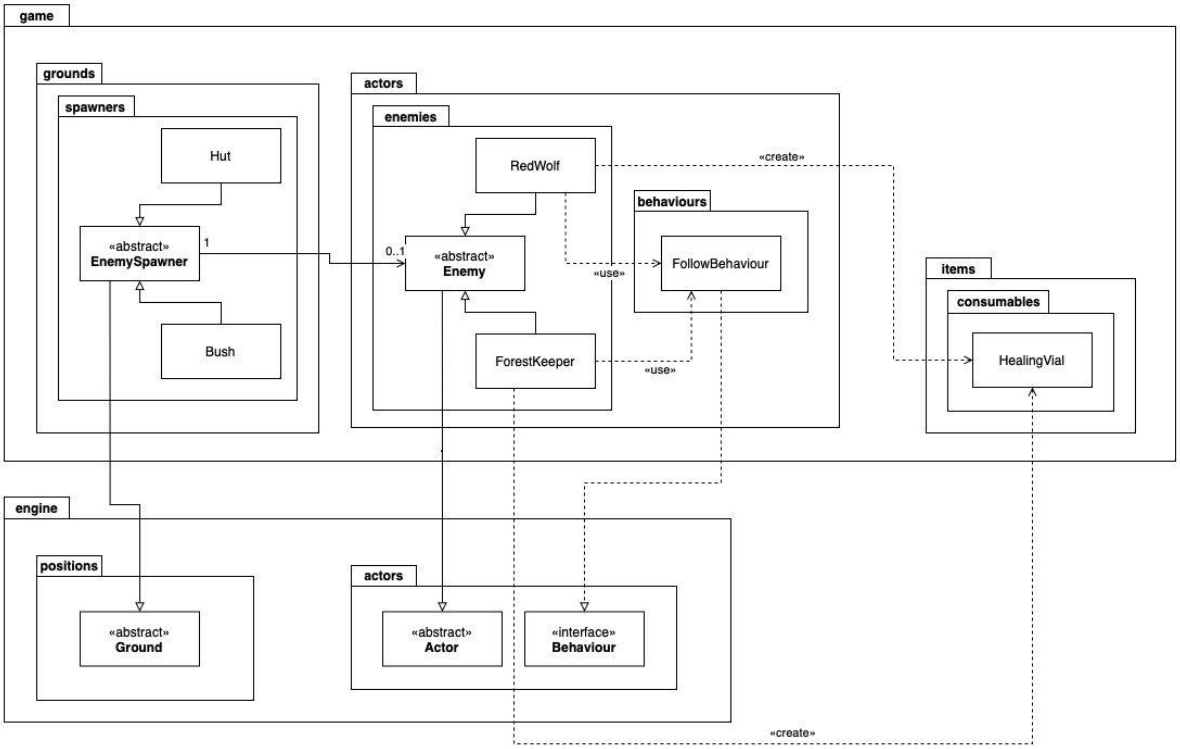

Changes from Assignment 1

We decided how we implemented the spawning of enemies, instead of needing to type the spawn rate in ‘application’, for each initialization of a spawner. We decided that the spawn rate should be an attribute of each enemy which is set upon initialization of the enemy. And inside of Enemy abstract class, we have getter and setter methods to set and return the spawn rate of the specific enemy.

The main objective for REQ1 are to add Hut, which spawns ForestKeeper, and Bush that spawns RedWolf, both of these enemies follow the player around.

The Abstract ‘EnemySpawner’ class extends the Ground abstract class. Now, it only accepts 1 argument, which is the enemy type to be spawned. This adheres to the Dependency Inversion Principle (DIP), as we rely on the abstraction of the enemy rather than its concrete implementation. ‘EnemySpawner’ class handles the spawning logic by overriding the ‘tick’ method and comparing a random number against the spawn rate of the enemy (by using the getter method).

All spawners like Graveyard, Hut and Bush extend abstract class EnemySpawner. Since all these spawners share some common attributes and methods, we abstract these identities to avoid repetitions (DRY).

Since enemies share common behaviours, We made ForestKeeper and RedWolf class extend the abstract Enemy class. Since they share some common attributes and methods, abstracting these identities avoids repetition, adhering to the (DRY). Both ForestKeeper and RedWolf can be used interchangeably in contexts where an ‘Enemy’ is expected, like when

passing the argument through the ‘EnemySpawner’ constructor, this demonstrates adherence to the Liskov Substitution Principle (LSP).

Advantages over our old implementation:
By having the spawn rate be an attribute of the enemy, we eliminate the need to repeatedly type out the spawn rate in ‘Application’ whenever we are initialising a new spawner.

Advantages over an interface implementation:
Since the spawning logic across all enemies remains the same, and the only differences are the enemy type that is spawned and the spawn rate, we can just write the spawning logic once in the abstract ‘EnemySpawner’ class and have all spawner subclasses extend it. An interface implementation would require a lot of repeated code across all the subclasses, which would violate the (DRY) principle.

Disadvantages:
Less flexibility
It may be less flexible than an interface implementation, as the spawning logic is already defined in the parent class. This means that any classes that have special spawning logic will be limited to the constraints of the parent class.