Design Rationale REQ1

Implementation & Why:

We first created a brand new ‘Maps’ class to store all the available maps. Creating a new ‘Maps’ class demonstrates the single responsibility principle. This class is responsible for managing maps, keeping the ‘Application’ class cleaner by delegating the map-related responsibilities to a different class. We then created a new ‘ANCIENT_WOODS’ map in ‘Maps’ class. By having a ‘Maps’ class and adding ‘ANCIENT_WODDS’ map, we are extending the functionality without modifying existing code (open closed principle). We added a new Gate inside the ‘BURIAL_GROUNDS’ map at the coordinate (25,12), this is because we want to make it harder for the Player to reach the gate. Once unlocked and used, the Player will travel through the Gate and arrive in the ‘ANCIENT_WOODS’ map at the coordinates (20,3) where the Player will be inside a building.

We then created an ‘EmptyHuts’ class that extends ‘Spawner’. ‘Spawner’ is a class that extends ‘Ground’, which will allow ‘EmptyHuts’ to spawn entities. ‘ForestKeeper’ class was also created that extends ‘EnemyActor’ and implements ‘ActorSpawn’. ‘EnemyActor’ is an abstract class that controls most of the specifications of the entity like name, displayChar, hitPoints and more. ‘ActorSpawn’ interface is to show that ‘ForestKeeper’ can be spawned and ‘ActorSpawn’ is an interface class because it can be implemented into classes easily that needs to be spawned, and does not need to be implemented for classes that do not spawn. The interface ‘ActorSpawn’ is responsible for spawning entities, this adheres to interface segregation principle by having clear and minimal methods for entities that need to be spawned. We used ‘addDroppableItem’ in the ‘ForestKeeper’ constructor so that it will have a 20% chance of dropping a ‘HealingVial’ when killed by a ‘Player’. We also override the method ‘allowableActions’ to have implement the ‘FollowBehaviour’. ‘FollowBehaviour’ was copied from a demo provided to us. In ‘allowableActions’ we check if the ‘otherActor’ has the capability type of ‘EntityTypes.PLAYABLE’, which indicates that it is a player that is within the ‘ForestKeeper’ exits, and also checks to make sure that ‘ForestKeeper’ don’t already have an existing ‘FollowBehaviour’. If it meets all conditions, ‘ForestKeeper’ will start following the ‘Player’. And we also override the interface method ‘spawn’ so that it spawns with a 15% chance. The ‘Bush’ class was implemented the same way as ‘EmptyHuts’ and ‘RedWolf’ was implemented the same way as ‘ForestKeeper’. The classes ‘EmptyHuts’, ‘RedWolf’, ‘ForestKeeper’ and ‘Bush’ adhere to the single responsibility principle as well. ‘EmptyHuts’ and ‘Bush’ classes are responsible for spawning entities in empty huts and bushes, ‘ForestKeeper’ represents an enemy actor, and ‘RedWolf’ represents another enemy actor. These classes encapsulate specific behaviours and attributes related to their respective entities.

Our use of class inheritance and abstract classes demonstrates OOP principles. For example, ‘EnemyActor’ and ‘Spawner’ serve as a base class for ‘ForestKeeper’ and ‘RedWolf’,’EmptyHuts’, and ‘Bush’ promoting code reuse and abstraction of common attributes and behaviours. We encapsulate attributes and behaviours within classes, exposing them through methods. For instance, the use of getter and setter methods allows controlled access to class attributes, promoting encapsulation.

Pros:

- Our code is easy to manage and to extend functionalities as it follows SOLID principles.
- Our code shows the Single Responsibility Principle by dividing responsibilities into distinct classes like Maps, EmptyHuts, ForestKeeper, RedWolf, etc.
- Our code shows the Open-Closed Principle by creating a ‘Maps’ class and adding the ‘ANCIENT_WOODS’ map without modifying existing code, and new maps can be added without changing existing code.
- Our code shows the Interface Segregation Principle as well as we created an ‘ActorSpawn’ interface. It is only used on entities that need to be spawned, so that classes that do not need to be spawned do not have unnecessary methods.
- The use of inheritance and abstract classes (EnemyActor, Spawner) demonstrates the principles of Object-Oriented Programming.

Cons:

- The use of multiple classes and interfaces can make our code more complex. Managing the relationships and interactions between different classes could become challenging, making our game code harder to comprehend and maintain over time.

Extendability:

- Our game code can easily add new entities. By creating new classes that use our existing classes and/or interfaces, new entities can be integrated without extensively modifying existing code.

- Adding new maps can be done by extending the Maps class, allowing for different map layouts. This Maps class can be used to add new Maps anytime.

- Our code structure allows for the introduction of dynamic features such as different weathers. New behaviours or attributes based on weather can be added to existing entities, promoting adaptability to changing game dynamics.

- The code can easily accommodate changes in behaviours for different entities. Modifying the behaviour methods or introducing new behaviour classes allows for altering how entities interact with each other and the game environment.

- The Spawner class can be extended or modified to have spawning rate.