Forest Keeper and Red Wolf Implementation

To implement the two new enemies "Forest Keeper" and "Red Wolf" I created new classes that extend the EnemyActor abstract class that aligns with the Liskov Substitution Principle (LSP) as the child classes of EnemyActor all have the same functions. A new abstract class was not created for the following enemies even though they share some duplicate code in how the FollowBehaviour is added, this is because creating a new abstract class just to add a new Behaviour to a type of enemy would have created an excessive number of abstract classes. The child classes have a method added to add a follow behaviour when they get within range of the player. The FollowBehaviour implements the Behaviour interface where it gets its method from, by implementing an interface with only essential methods, the design adheres to the Interface Segregation Principle (ISP), making the code maintainable and easy to extend. Newspawners were created to spawn the new enemies including the RedWolfSpawner and the ForestKeeperSpawner this aligns with the Single Responsibility Principle (SRP) as these two classes are made to just handle the spawning of their respective enemies. I believe this makes the code a bit easier to read but another approach that could have been taken is having one spawner class that takes an enemy to spawn as a parameter, however this could put more work in the application class which would make the code potentially less extendable and the code less straightforward to read. Wealso created two new ground types to hold the new spawners Hut and Bushes this also aligns with the Single Responsibility Principle (SRP) and they extend the Spawning ground abstract class to receive all of its functionality (LSP) an alternative to this practice of extending new spawning grounds from the SpawningGround abstract class would be to make the spawningGround a concrete class and in the application file setting the correct display char ('h' or 'm') with setDisplayChar however I think the advantage of our practice was it doesn't rely on a god class and it makes the code easier to understand when each type of spawning ground has its own class.

Runes Implementation

For requirement 2, "Runes" extends the game package "Item". The "Item" class is the blueprint for subclasses which extend from it such as "Runes". This means the subclass can be used wherever the base class is used without needing to make any changes which aligns with the Liskov Substitution Principle (LSP) . This is because these classes have the same structure and functions which can be overridden if necessary. This ensures the code is easier to extend in the future. In order for an enemy actor such as "Forest Keeper" to drop "Runes" after it dies, "Runes" is implemented in the unconscious function in the enemy actor classes Forest Keeper, Red Wolf, Wandering Undead, and Hollow Soldier. This enables each enemy actor to drop a unique runes amount; however, the code is a bit repetitive as each unconscious function has the same overall structure just with different variables and items. Additionally, the abstract EnemyActor class aligns with the Open/Closed Principle (OCP) as abstraction easily allows for code to be extended and not modified. Having the EnemyActor class allows all the enemy actors to have common functionalities and attributes whilst its subclasses can contain new functionalities specific to the particular enemy such as ForestKeeper and its specific unconscious function. A downside to having an abstract class is that it is the only abstract class that its subclasses can extend. “Consumable” interface follows the Interface Segregation Principle (ISP) as it is a small interface which only contains one method which is to consume anything the actor is given. An interface is useful as any class can implement as many interfaces as they require while small interfaces can continue following SRP by having a single purpose. In addition you can implement as many interfaces as required in any class. However, implementing the interface can be expensive. Finally, “ConsumableAction” aligns with the Single Responsibility Principle (SRP) as classes are created to have one responsibility. The ConsumableAction class is created only to consume an item used by the actor, it has no other responsibilities. This also means that if another item was created it can easily extend ConsumableAction without interfering with any other classes.

Trader Implementation

For requirement 3, a new abstract ‘Trader’ class was made. Much like the ‘EnemyActor’ abstract class, this new ‘Trader’ class specifically accounts for all actors that are able to buy and sell items in the game. A new ‘Traveller’ class extends from this ‘Trader’ class. It is important to separate such traders from enemies as despite both being actors in the game, they perform vastly different actions. For example, the traveller does not wander around the map or attack the player. This further aligns with the Single Responsibility Principle (SRP) as the ‘Traveller’ class handles all of the traveller’s responsibilities and nothing more. Additionally, the use of a ‘Trader’ abstract class allows our code to be easily extendable. This also aligns closely with the Open/Closed Principle (OCP) as should new traders be created, our code can be easily extended, but not modified. We chose to include this abstract class as the requirement specifications note that different traders may sell items for different prices, strongly suggesting that new travellers will be added in the future. This further sets our code up to continue to follow the DRY principle, as if new traders are added to the game, attributes common to all traders only need to be coded once in the abstract class. Nonetheless, abstract classes can be quite expensive. Should new traders not be added in the future, adding an abstract class may be considered redundant. It may add unnecessary complexity to the code and make our code more difficult to navigate and understand.

Great Knife and Giant Hammer Implementation

For requirement 4, two new concrete classes GreatKnife and GiantHammer were made the extend the preexisting WeaponItem abstract class, this aligns with the Liskov Substitution Principle (LSP) as the concrete weapon items share all of the functionality of their parent class. Because they can be traded with the traveller GreatKnife implements the BuyableItem and SellableItem interface whereas the Giant Hammer only implements the SellableItem interface because it cannot be bought from the trader, it is important that the classes don't implement methods they don't use, having unnecessary methods could make the class to big and over complicated so this aligns with the Interface Segregation Principle (ISP). There are two new concrete classes extending the Action abstract class these are the GreatSlamAction and the StabAndStepAction these are both designed to handle the two new actions in the requirement and align with the Single Responsibility Principle (SRP) as each new class is created specifically to handle the new skills of both the weapons. Overall the advantages of this design is that it aligns with the previous design. A downside to our implementation is that the BuyableItem interface class is not very extendable. This is because if new Trader actors are created with a unique buying function for each item, it is not easy to change the interface method to include these changes without modifying existing code.