REQ2 & REQ3 Introduction (Myint Myat Thura)

**Assignment 3 REQ 2 features extension below

Many of my design choices for REQ2 and REQ3 (which I also did) will revolve around the DRY principle. I believe that non-repetitive code is a necessity for clean and professional software. Most of my implementation decisions will mostly involve extremely good scalability/extension.

REQ2 – Trader & Runes

Runes and Weapons dropping

For runes, we can see that as per instructions they are the main currency of the game (for players). Every actor in the game will carry runes, however, only players can use them in any way they want. This allows us to see something. To best adhere to DRY principle, the goal is to reduce repetitive code. There is no better way to do this than to group the similarities and split the differences.

Brief walkthrough on implementation design – (Single Responsibility Principle, DRY principle)

To understand this more, we can see that runes can be dropped and used. Who can use the runes? ONLY the player, who can drop the runes? EVERYONE. When does everyone drop the runes? When they die. Knowing this, the most efficient way to accomplish dropping runes is to associate dropping the runes with death action. Each actor has an ActionList which stores action objects, in this case DropAction

ActionList --<<stores>>--> Action (e.g., DropAction & DeathAction)

DropAction --<<has>>--> Item

Runes extends Item

This means that when someone dies, it always calls the DropAction inside the actionlist. This is following the Single Responsibility Principle and Don't Repeat Yourself principle. By associating the dropping of runes with the death action and having a dropAction inside the actionlist for each actor, we are reducing repetitive code and ensuring that each class has a specific responsibility.

In the case of enemies with weapons dropping weapons, we could easily manage this by making runes and weapons part of their inventory. Then calling the dropAction for every item in their inventory.

Pros –

1) Associating the dropping of runes with the death action and having a DropAction inside the actionlist for each actor reduces repetitive code and follows the **Don't Repeat Yourself (DRY) principle.**

2) Having a specific DropAction for each actor follows **the Single Responsibility Principle (SRP)** and makes the code more modular and easier to understand.

3) Grouping similarities and splitting differences allows for more **efficient code and easier maintenance.**

4) No matter what the enemies have in their inventory, the rule is that everything in the inventory

will always be dropped. Therefore, adding weapons and runes to inventory would make it **extremely scalable and efficient** in terms of utilizing DropAction to its full potential.

Cons –

1) We are implementing this method with the assumption that all the enemies will drop something on death. If there was ever a function that allows enemies to drop things even while not dead, this would force us to link dropAction with other classes, not just deathAction.
2) If the enemy were to not drop anything upon death, the associative relationship between deathAction and dropAction becomes redundant.

Deciding how many runes to drop and when to decide that.

HostileCreatures --<<uses>>--> RandomNumberGenerator

For what the enemy drops, the requirement is that we have to use a random number generator with a range. However, instead of deciding the rune drops at the moment of death, at the generation of these hostiles, we can assign them a random number for runes to be dropped upon death. This way, our DeathAction calls the least number of methods possible making the code more efficient. This can be accomplished by having a dependency relationship between HostilesCreature abstract class and random number generator class.

Trader

Brief Walkthrough on implementation design – (Single Responsibility Principle, DRY Principle)

Merchant Kale will be the trader for this particular game. At the moment, merchant Kale would not keep track of purchased and sold items in an inventory. Therefore, purchasing and selling would be carried by the player with merchant Kale as a trader with prices of weapons. Trader inherits from the Actor abstract class and merchant Kale inherits from the Trader class. The player will have an association relationship with a new class called PurchaseAction and SellAction which inherits from Action abstract class. And merchant Kale will hold weapon items and runes as attributes to give details of trade prices. Each weapon will be assigned a number of runes.

Actor --<<stores>> Item

Trader extends Actor

Trader --<<has>>--> Runes

Player has the ability to interact with the trader using the PurchaseAction and choosing an item, when the PurchaseAction has been called, sellAction from the trader will be called. This back-and-forth action between the Player and the trader creates a trading exchange.

PurchaseAction && SellAction --<<uses>>--> Trader

Player --<<Uses>>-- purchaseAction && sellAction

This design follows the Single Responsibility Principle, each class will have a specific responsibility and reason to change. For example, trader will manage weapons and assign runes. While PurchaseAction and SellAction have the responsibility of handling the purchase and sell actions respectively.

Pros –

1) The use of inheritance and abstraction allows for **reusability and easier maintenance**, for example, if a change is made to the Actor abstract class, it will reflect in all classes that inherit from it.
2) The use of separate classes for different actions (PurchaseAction and SellAction) follows the **Single Responsibility Principle (SRP)** and makes the code more modular and easier to understand.
3) The use of a dependency relationship between the Player and PurchaseAction classes allows for a **clear separation of concerns** and makes it easier to add new actions in the future.

Cons –

1) As new objects and classes are added in the future, there may be a need to create additional Action classes for each new type of interaction. This could lead to an increase in the number of classes and potentially make the code **more difficult to manage.**
2) There may be **some redundancy in the code** if similar actions are performed by different classes. For example, if both the Player and Trader have similarities in methods for interacting with their inventories, there may be some overlap in functionality.

Assignment 3 REQ 2 features extension (no optional)

Cage extends SpawningGround
Cage --<<has>>-->Dog
Dog extends HostileCreature
Barrack extends SpawningGround
Barrack --<<has>>--> GodrickSoldier
GodrickSoldier extends HostileCreature

The two new spawning grounds of Stormveil Castle each spawns Stormveil Castle hostile creatures. Cage and Barrack inherits the existing SpawningGround abstract class due to similarities in functionality. This abstraction in SpawningGround provides a useful extension in features(Open-closed Principle) that does not require much modification or building functionality from scratch ,saving time and possible code repetition if they are similar. As opposed to spawning enemies from either side of the map, Dog and Godrick Soldier do not have direction restriction, hence creating a new instance is sufficient. Using enemy factory classes is only necessary when direction requirement is needed, sticking to Single Responsibility Principle. As shown in REQ 1 for attack behaviours and status for hostile creatures, Dog and Godrick Soldier will have the same attack behaviours and Stormveil Castle status to identify them in AttackBehaviour class as they are hostile creatures but only differ in stats for hit point and spawning chance.