REQ4 – A Guest from Another Realm

This REQ4 uses the improvements from the Combat Classes that I have previously talked about in the REQ3 rationale. This made the entire implementation more efficient and abiding by OOP principles.

Death Action has also been reimplemented in a way, so that the Player drops runes at the previous location before death and not at the location of death. This makes it so that it can be implemented well with death from cliff.

Do note the multiple changes and improvements that we have made from A2 implementation.

Astrologer

The Astrologer weapon, since we did not do the optional, uses the Uchigatana weapon from the Samurai class.

Summoning Sign

The requirement states that the Summoning sign must find an appropriate exit so that the ally/invader may be spawned safely. So, instead of finding the exits of the summoning sign, we basically made it so that the player must be directly on top of the summon sign to use it. Then we used the available exits of the player to spawn the ally/invader which basically does the same thing without writing extra code, this allowed us to reuse the code given in the engine therefore making it more efficient.

Ally and Invader

Ally and Invader works the same except that the Invader has so much in common with hostile creatures in the way that they attack Player, so the only difference is that instead of directly inheriting Invader from Actor, we inherit it from the Hostile creature class that we created. Ally will inherit directly from actor.

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

Ally and Invader are to be spawned by the Summon Sign, the summon sign uses the random generator class to give each a 50% chance of spawning. When they are spawned, they use our newly created ClassManager singleton class to select a random role for them, then they are given the health points and they are given the appropriate weapons. We use the same ClassManager singleton class for both Player and these

summoned creatures. This makes it so that our code is efficient and reuses the code. They differ a bit from summon creatures though, because they do not disappear when the player rests at site of grace we have to implement a different method for them in the ResetManager singleton class. At the creation of Ally and Invader, they are passed as instances to the ResetManager class. Then we make it so that they are only removed when the player dies. Therefore, we are then again making use of the existing classes for the code.

Pros:

1. **Code Reuse (Inheritance):** I used the same ClassManager *singleton* class for both Player and these summoned creatures to make my code efficient and reuse the code. This is an example of the *inheritance* principle in OOP, where a new class can inherit the properties and methods of an existing class to promote code *reuse*.
2. **Efficiency (Polymorphism):** At the creation of Ally and Invader, they are passed as instances to the ResetManager class. This allows for efficient code reuse by making use of the existing classes for the code. This is an example of the *polymorphism* principle in OOP, where objects of different classes can be treated as objects of a *common superclass or interface*.
3. **Simplified Management (Encapsulation):** By implementing a different method for Ally and Invader in the ResetManager *singleton* class, I made it so that they are only removed when the player dies. This is an example of the *encapsulation* principle in OOP, where we hide the internal details of an object and only expose what is necessary to *reduce complexity and increase maintainability*.

Cons:

1. **Limited Flexibility (Open-Closed Principle):** By using the same ClassManager *singleton* class for both Player and these summoned creatures, we may limit future *flexibility* if we decide to add additional functionality or attributes to either class. This goes against the *Open-Closed Principle* in OOP, which states that software entities should be open for extension but closed for modification.
2. **Difficulty in Testing (Dependency Inversion Principle):** The use of singletons for the ClassManager and ResetManager classes can make it difficult to test and can introduce global state into my application. This goes against the *Dependency Inversion Principle* in OOP, which states that high-level modules should not depend on low-level modules but both should depend on abstractions.
3. **Tight Coupling (Law of Demeter):** By passing Ally and Invader as instances to the ResetManager class at their creation, we may introduce tight coupling between these classes. This goes against the *Law of Demeter in OOP*, which

states that an object should only communicate with its immediate neighbors and should not have knowledge of the inner workings of other objects.

Ally--<<Extends>>--Actor

Invader--<<Extends>>--HostileCreature

ResetManager--<<Uses>>--Ally || Invader

Player--<<Uses>>--GameReset

Player--<<Uses>>--SummonAction

SummonAction--<<Creates>>--Ally || Invader

Ally || Invader--<<Uses>>--ClassManager