REQ3 – Godrick The Grafted

Before I go on, there are some major changes to talk about that are not directly related to A3. These are great changes that shaped the overall assignment. First are Runes, runes were implemented in a sloppy way back in A2, but we have made it so that it suits the assignment instructions and the principles of OOP. Runes for this reason did not work back in A2, the player was unable to pick up runes or gain runes from enemies, this was fixed by creating an interface and a *Rune Handler* to do this.

Combat Classes, back in A2, we implemented this with just sloppy if-else statements, however this has been changed and Combat Classes now have their own *classes*. They use an interface and a *ClassManager* which allowed for Allies, Invaders and Player to implement each class better.

Now onto the real tasks of REQ3 in A3.

Axe of Godrick and Grafted Dragon

These weapons were implemented just like any other weapons, but they have slightly changed features when it comes to buying and selling. This will be talked about more in the section of new trader Enia.

Golden Runes

Golden Runes were to be scattered on the ground and gives a player the ability to generate random runes in a given range upon consumption.

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

Golden Runes just like the other normal Rune implements an *interface* called *RuneHandler*. *RuneHandler* has one method called *action*. There will be another class called *RuneManager*, this class is basically a singleton implementing class which will handle all the rune-like objects and perform actions accordingly. For example, for normal runes, it would take the amount of runes that object holds and then add it to the player's total rune. Then it will be removed from the inventory. This makes it so that the player will always have updated runes, but it will never hold a rune object in the inventory to avoid further complication; consider if there were 100 rune objects in the inventory. We are not going to be using them all, we just need the rune amount, so why not take them out? For Golden runes, we are simply just going to generate a number of runes based on the range that is given, then we will simply remove it from the inventory.

GoldenRunes--<<Implements>>--RuneHandler

GoldenRunes--<<Uses>>--RandomNumberGenerator

RuneManager--<<Uses>>--GoldenRunes

Player--<<Uses>>-- RuneManager

Pros:

1. **Single Responsibility Principle:** I implemented the RuneHandler interface and the RuneManager singleton class to promote the *Single Responsibility Principle* by separating concerns and ensuring that each class has a specific responsibility.
2. **Encapsulation:** I used RuneManager to handle all rune-like objects and perform actions accordingly to promote *encapsulation* by hiding the implementation details of how runes are managed from other classes.
3. **Simplified Inventory Management:** I removed rune objects from the inventory after their amount has been added to the player's total rune count to help simplify the inventory management and avoid complications.

Cons:

1. **Difficulty in Testing:** The use of a singleton for the RuneManager class can make it difficult to test and can introduce global state into my application.
2. **Limited Flexibility:** The removal of rune objects from the inventory after their amount has been added to the player's total rune count could limit future flexibility if I decide to add additional functionality or attributes to rune objects.
3. **Unpredictability:** The generation of a random number of runes for Golden Runes could introduce unpredictability into the game, which may not be desirable in all cases.

Remembrance of the Grafted

Remembrance of the Grafted is an item that forces the new trader to implement a new type of trading, *exchange*. It can also be sold for 20000 runes. Elaborated on section for new trader.

Finger Reader Enia

Finger Reader Enia is a new type of trader that allows you to purchase 2 new weapons

called Axe of Godrick and Grafted Dragon. But the difference is, instead of runes, it will only accept Remembrance of the Grafted to be *exchanged* for.

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

This new trader implements the same methods of Kale but instead of having the same items, it will only have two items. Exchange action is a new action that has been created, basically instead of acting on runes, it acts on the items that exist in the player's inventory. This is linked with how we implemented runes, because we will be holding no runes, we can basically assume that the only items that we will have in our inventory is Remembrance of the Grafted, this makes it easy to check if the item is in our inventory. Instead of using interfaces and such, we simply call the size of the inventory, if it is more than 0, then we are sure that we have Remembrance of the Grafted in the inventory. We chose this implementation because it requires nothing else and we are able to reuse the existing methods in the Player class, making the code extremely efficient. The trader will still implement the WeaponTrading interface and the singleton class TradingManager.

Enia--<<Extends>>--Actor

Enia--<<Uses>>--TradingManager

ExchangeAction--<<Extends>>--Action

Enia--<<Uses>>--ExchangeAction

Player--<<Uses>>--ExchangeAction

Pros:

1. **Code Reuse (Inheritance):** I chose this implementation because it requires nothing else and I am able to reuse the existing methods in the Player class, making the code extremely efficient. 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. **Simplified Inventory Management (Encapsulation):** By assuming that the only items that we will have in our inventory are Remembrance of the Grafted, I made it easy to check if the item is in our inventory. 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*.

3. **Efficiency (Polymorphism):** The trader will still implement the WeaponTrading interface and the *singleton* class TradingManager, allowing for efficient code reuse. 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.

Cons:

1. **Limited Flexibility (Open-Closed Principle):** By only having two items and assuming that the only items in our inventory is Remembrance of the Grafted, we may *limit future flexibility* if we decide to add additional functionality or items to the game. 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 a singleton for the TradingManager class 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 relying on the size of the inventory to determine if Remembrance of the Grafted is in our inventory, we may introduce *tight coupling* between the trader and the Player class. 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.