REQ 2

A. Enemies

Enemies are depended on by the “RuneManager” class in its “addRuneToPlayer” method. This method is used whenever an enemy dies based on the specifications. This follows the DRY principle as we do not have to repeat the same method for different enemies.

B. Trader

Our trader, “MerchantKale” class, extends the abstract class “Actor” from the provided engine. Having it extend this class allows us to easily insert actor into our map in application. A package called “Trader” is created in the game to group all related classes together, making for better organisation.

“MerchantKale” has a dependency to the weapons he is able to sell. In our requirements said weapons are “Uchigatana”, “Club” and “GreatKnife”. The advantage over using this instead of a sellable interface is that we are able to easily extend the amount of weapons and/or items Merchant Kale can sell by just adding it to his inventory.

“MerchantKale” has dependencies to “BuyAction” and “SellAction” as it is found in his method “allowableActions”, Which gives them the ability to sell weapons and buy weapons from the “Player”.

“BuyAction” and “Sell Action extend “Action”, which allows them to use or override any and all actions when necessary. Both Actions reside within the Action Package.

We now have a new Trader, “FingerReaderEnia”, whom extends the abstract class “Actor” from the provided engine. Having it extend this class allows us to easily insert actor into our map in application. A package called “Trader” is created in the game to group all related classes together, making for better organisation.

“FingerReaderEnia” has a dependency to the weapons he is able to sell. In our requirements said weapons are “GraftedDragon” and “AxeOfGodrick”. The advantage over using this instead of a sellable interface is that we are able to easily extend the amount of weapons and/or items Merchant Kale can sell by just adding it to his inventory.

“MerchantKale” has dependencies to “TradeAction” and “SellAction” as it is found in his method “allowableActions”, Which gives them the ability to sell weapons and buy weapons from the “Player”.

The new “TradeAction” extends “Action” in the engine, which allows it to use any override methods when necessary. Following the SRP rule.

C. Runes

The “Player” inherits the “Rune” class as the player creates an instance of rune and adds the runes to their inventory.

Our “Rune” class is managed by “RuneManager”. It contains a simple attribute, constructor and methods. 1 Manager manages many runes. Which follows the SRP.

Our “RuneManager” class has a private instance meaning that there should only be one instance of it throughout the program. Its main function is to manage the “Rune” Class. It allows the runes to be added to the player, shown to the user, and subtracted from the player. It follows the Single Responsibility Principle(SRP) as We use to rune manager to prevent rune from becoming a god class that controls everything rune related.

“RuneManager” has a hashmap that contains (key,value) pairs denoting (“Display Character”, (the lower bound of how many runes are dropable, the upper bound of how many runes are dropable). This is a very advantageous way of implementing this as it reduces the dependencies of having each monster carry runes and needing to check said runes everytime we kill a monster. This also allows us to easily extend our class whenever more monsters are added by putting new entries into our hashmap.

“RuneManager” has a dependency to the enemies “LoneWolf”,”GiantCrab” and “HeavySkeletalSwordsman” as they are used in the addRuneToPlayer method whenever necessary. This method is also dependant on the “RandomNumberGenerator” class to generate a random amount of runes based on a specific enemy. This method allows for the ease of transfer of runes from dead monsters to players.

“BuyAction” is also dependant on the RuneManager as it calls the “RuneManager” instance within its execute override method to help it manage how the runes flow from the “Players” inventory. Having a separate class from “SellAction” follows the SRP principle.

“SellAction” is also dependant on the RuneManager as it calls the “RuneManager” instance within its execute override method to help it manage how the runes flow into the “Players” inventory. Having a separate class from “BuyAction” follows the SRP principle.

A new item “GoldenRune” has been introduced in assignment 3, It extends “Item” abstract class in the engine and implements “Consumable” in the utils package. This allows us to use any methods within the “Item” abstract class and allow the player to consume the golden rune.

When a golden rune is consumed a random number of runes is generated using “Random” and is added to the players inventory. This means that the “GoldenRune” class would have to have a dependency to the “RuneManager”

We have added the Item, “RemembranceOfTheGrafted” and the weapons, “AxeOfGodrick” and “GraftedDragon” as placeholders as we are not planning on implementing the optional tasks.