Requirement 2

This UML class diagram will showcase the trading system of the game using runes that the players will have which is the currency in this game and how a player can obtain runes. The goal of this design is to allow the players to trade weapons such as buy or sell weapons by using runes and to get runes from enemies while having modularity and reusability.

To begin with, a rune is an item so it should extend the item abstract class to inherit all attributes of an item. A Rune Manager class has also been created to manage any operation that involves runes such as increase, decrease rune amount due to trading. This is because it is not the player's responsibility to handle any rune operation but the system. Hence why there is also a need to have an association between the Rune class and the Rune Manager Class. The Rune Manager should associate itself to ensure that there is only one instance of it only because there should be only one manager handling the rune operations just like in real life where there is only one manager in a store.

Since a player can sell weapons to gain runes, buy weapons with runes and store runes, they need a rune manager and an inventory of items. So the Player class has an association with the Item and Rune Manager class. It associates with Item rather than Rune since a player can store other items. If we let the Player class associate with every item then we will have multiple dependencies between the Player class and the items which extends the Item class and it is not recommended. So to reduce coupling we associate the Player class with the Item class instead. This design will obey the Dependency Inversion Principle since it breaks down the dependencies between Player and the items. Moreover, it prevents any violation of the Open-Closed Principle as we do not need to use an if-else statement to check the type of item. Furthermore, they will also have a dependency to a BuyWeaponAction and SellWeapon class since they can buy/sell weapons that extend the action abstract class since these actions are also a type of action and the action abstract class has all the common attributes of an action.

Next, the trader that trades weapons are actors too so they should extend the actor class to inherit all attributes of an actor. A trader is also able to trade weapons which allow players to buy or sell weapons so they will have a dependency to a BuyWeaponAction and SellWeaponAction class that extends the action abstract class since these actions are also a type of action and the action abstract class has all the common attributes of an action. Thus, we can adhere to the Do Not Repeat principle. Alternatively we can create a single action called tradeAction that does both the trading procedure but to adhere to the Single Responsibility Principle, it is best to separate the 2 trading actions into separate classes. Lastly, the trader in this system also cannot move nor be able to be attacked, attack or die. Therefore, the trader class has a dependency on DoNothingAction but not to AttackAction or DeathAction. Alternatively, we can create an interface called Mortality so we can differentiate who can die or not.

Players can sell their weapons in their inventory so it has an association with the WeaponItem Class. The player has a collection of WeaponItems instead of the weapon class to reduce coupling between the classes and prevent violating the Dependency Inversion Principle. Additionally, if there are additional weapons we do not need to fix the Player class to accommodate the new weapons

Next, when enemies die they will give runes to players, so they should have a dependency to a EnemyDeathAction class. Furthermore, Heavy Skeleton Swordsman will drop its weapon that can be sold to a trader after death so they should have a dependency on the DropWeaponAction class. But other enemies should not have it since they don't have any weapons to drop. Subsequently, players are able to pick up said drop weapon to sell so players will have a dependency to PickUpWeaponAction class. Lastly, the EnemyDeathAction, BuyItemAction, SellItemAction will be dependent on Rune Manager since these actions involve runes.

This UML class diagram provides modularity, extensibility and reusability by using inheritance and polymorphism as stated above. However, the disadvantage of this design is that there are too many dependencies which is a weak relationship. We can solve this problem by creating an association between the entities and the action class through a manager class which will handle the actions. Consequently, this design follows Single Responsibility Principle, Dependency Inversion Principle and Do Not Repeat Yourself Principle.