## Requirement 2 - Traders and Runes

Requirement 2 is about Runes and the actors that control them. For example, the player holds runes, enemies drop runes upon death, and traders will buy and sell weapons in exchange for runes.

**Enemies**

The way we implemented enemies dropping runes is by having an abstract method that returns the range of runes they can drop. By default, this is (0, 0) if enemies do not drop runes, but overridden if the enemy does. This adheres to the Liskov substitution principle as all enemies will "drop" runes for the purposes of our program, but can drop varying values or none at all. Since the default method is in the abstract class, enemies that do not drop runes do not need to implement unnecessary methods; only those that do.

**RuneManager**

We adhere to the single responsibility principle as each class has a key responsibility when it comes to runes (enemies drop runes, the player has the runes, traders buy/sell for runes etc). The specific class in charge of keeping track of this is the RuneManager. This allows us to globally access the functionality that deals with Runes and allows us to change the balance of the player's runes. By adhering to the single responsibility principle, we reduce coupling for any single class in regards to managing runes.

**Tradable Weapon, and Purchasable/Sellable Interface**

The purchasable and sellable interfaces were created in order to set selling and buying prices for the weapons. Specific tradable weapons will also implement the interfaces to set a specific selling and/or buying price, as well as adding the enum of that weapon type. This enum is used in a hashmap that traders can refer to; this way, traders will know which weapons they are able to buy and sell, and this can be easily checked within their hashmap of weapon enums. This enum is used to refer to the WeaponFactory, another adherent of the single responsibility principle, to create weapon instances that are passed to the player. Traders will depend on the interface for buying and selling, hence we are applying the dependency inversion principle to reduce coupling.

**Trader and NpcManager Class**

The trader class follows the open-closed principle as it is open for extension but closed for modification. We have designed the class so that it can sell any kind of weapon in its inventory without having to modify the existing code. Additionally, we have a class NpcManager which creates Trader instances, and although these are somewhat coupled it is very easy to extend NpcManager to create other actors or other traders.

**Pros and cons**

The tradeoff with our tradable weapon design is that it can be somewhat convoluted and confusing to create new weapons and sell them to the player, on the trader's side. However, we have implemented various levels of abstraction with the dependency inversion rule as well as the open closed principle. This means that our system is overall more flexible, maintainable, and extensible, if more features are added in the future. This could include more traders which sell different weapons, more enemies which drop different runes, and so on.