Design Rationale REQ2

Implementation & Why:

Created Runes as a child of Item as Runes can be dropped by Actors and consumed to gain wallet balance. Every instance of Runes has a set int value so that we can set the dropped value for different EnemyActors that drop runes. This follows the single responsibility principle as the value of each rune for each enemy actor is stored in the instance of the Runes.

AddBalanceAction extends the action class, where it takes a Rune in the constructor. Actor addBalance method is called to add the balance to the Actor's wallet. The rune is then removed from the actor's inventory.

EnemyActor defines the runes dropped by each enemy actor in the game. This is done so by having an int value RunesDropped in the constructor to determine how many Runes are dropped when the enemy is slain. In order to allow the slain enemyActor to drop items and runes at the same time, dropRunes and dropItems methods are created and called at the overridden unconscious method. This closely follows the DRY (do not repeat yourself) Principle because all the classes that extend EnemyActor would not need to repeat the rune dropping logic as it would be inherited.
A dropRunes method is created to instantiate a new Rune Item at the current Ground position of the enemy once slain.

In order for the EnemyActor to drop multiple items (example: Wandering undead drops OldKey, HealingVial and Runes), we have created a new class ItemDrop that has the method dropItems. This method takes the map where the item would be dropped, the actor the drops the items and most importantly the list of Items that would be dropped as well as the Item's drop chance in the form of a list. The dropItem method iterates through the list of Items and the Item drop chance simultaneously as both list index correlates. The probability of dropping the item is then calculated and the item is added to the game map if the random double is within the range of chance. Coming back to the EnemyActor class, the dropItem method would then add the items that would be dropped to the map at the Actor's current location. This adheres by the single responsibility principle as the dropItem logic is coded externally and can be easily manipulated and debugged in future implementations for better extendability.

For the player to be able to drink from Puddle and gain health and stamina, allowableActions in Puddle is overridden to allow all PLAYABLE actors (Player EntityType) to have a DrinkPuddleAction if they are standing on a puddle. DrinkPuddleAction increases the player's health by 1 (fixed) and stamina by 1% (get the Player base attribute STAMINA and increase by 0.01). The Player's stats are increased by calling the actor's modifyAttribute method.

Bloodberry is implemented as a child of the TradeableItem class as it can be sold to the Traveller. When the Bloodberry is consumed a new IncreaseMaxAttributeAction is returned. This Action uses the affected attribute, operation and amount affected in the constructor to call the actor's IncrreaseMaxAttributeAction. This adheres to DRY (do not repeat yourself) as the modification code does not have to be repeated for other items that would increase

maximum attributes as it can increase any of the attributes the actor has, just instantiate the action.

Pros:

By adhering to DRY and single responsibility principles, we have made the code more maintainable and easier to extend in the future. Code consistencies are also adhered to make the code more readable.

Cons:

Increased complexity when using a new action to change the maximum attribute of an actor instead of directly accessing and modifying using the method.

Extendability:

IncrreaseMaxAttributeAction method allows for future classes that increases an actor's maximum attribute by simply calling the action.