Design Rationale for Requirement 2

Bloodberry
For the consumption of a Bloodberry, the initial approach was to add onto the Consumable Item abstract class made in Assignment 1. This could be achieved by adding a boolean to see which attribute was getting modified, i.e. if the attribute was changed or its maximum was modified. Although this worked, the design would not be extensible in the future if more items were added with different consumption requirements, thereby violating the Open/Closed Principle.

Instead of using the abstract ConsumableItem class, a Consumable interface was created. The Healing Vial and Refreshing Flask from Assignment 1 would now extend directly from the abstract Item class, rather than ConsumableItem, and implement the Consumable interface, satisfying the Dependency Inversion Principle. The new Bloodberry class would also implement the Consumable interface (more explained under Rune). Upon consumption, the consumeItem method is called through consumeAction, and the maximum healing attribute of the actor is modified.

This approach satisfies the Single Responsibility Principle (SRP) and the Open/Closed Principle. The potential downside lies in using abstractions as they affect efficiency. Additionally, using more classes may make it more difficult for incoming developers to understand the system.

Runes

The Runes are created as an Item, implementing the Consumable interface. The constructor of runes takes in the quantity, which is an attribute to determine the number of runes an actor is carrying.

The dropping of the enemy's inventory was edited; instead of the enemy dropping all the items in the AttackAction class, the responsibility was given to the enemy abstract class to drop all their runes by overriding the unconscious method involving another actor. This avoids downcasting in the Enemy class.

Since runes did not affect the health or stamina of a player, unlike other items which were consumable, a separate wallet action had to be created for it. From this, it could be inferred that we would need to make more action classes as more consumable items were added. This meant there would be little to no benefit in having a consumableItem abstract class, as well as violating the Open/Closed Principle.

The implementation of the interface, Consumable, helped fix this issue. The ConsumeAction class calls the consumeItem method in all consumable classes, which has a different implementation in each class. This works well on scalability as if more consumable items were to be added, these may have something different occur upon consumption, which could be handled by using the interface.

The consumeAction class also follows the SRP as it is responsible only for consumable items and is efficient by not creating any more action classes for consumption, regardless of what should occur when an item is consumed.

It may be said that DRY is somewhat violated as both Refreshing Flask and Healing Vial use ActorAttributeOperations and BaseActorAttributes, but have this code in their classes.

This interaction was chosen to be shown in the Interaction Diagram as although all of the implementations of this requirement are similar, this is a newer concept and includes the Actor and the Wallet class.

Puddle

In the first draft, Puddle was implemented via a drinkAction and a tick method in the puddle class. The tick constructor of the puddle gives us the location of the puddle and is used to determine whether an actor is standing in the puddle.

If the actor's current location matches the puddle's location, the drinkAction is returned as an allowableAction.

The drinkAction was a separate method and worked exclusively for puddles on which the player stood. This also violated DRY as it would create more Actions for future similar grounds/items when only one would be needed.

Due to these issues, Puddle now implements the Consumable interface and has a consumeItem method that changes the player’s health and stamina by 1. This is a more efficient implementation as it removes the need for the DrinkAction class and is more in line with the SRP and Open/Closed Principle.

The consumable interface has a toString method, which Puddle implements so that the name of the puddle is returned to be printed to the console with the string.