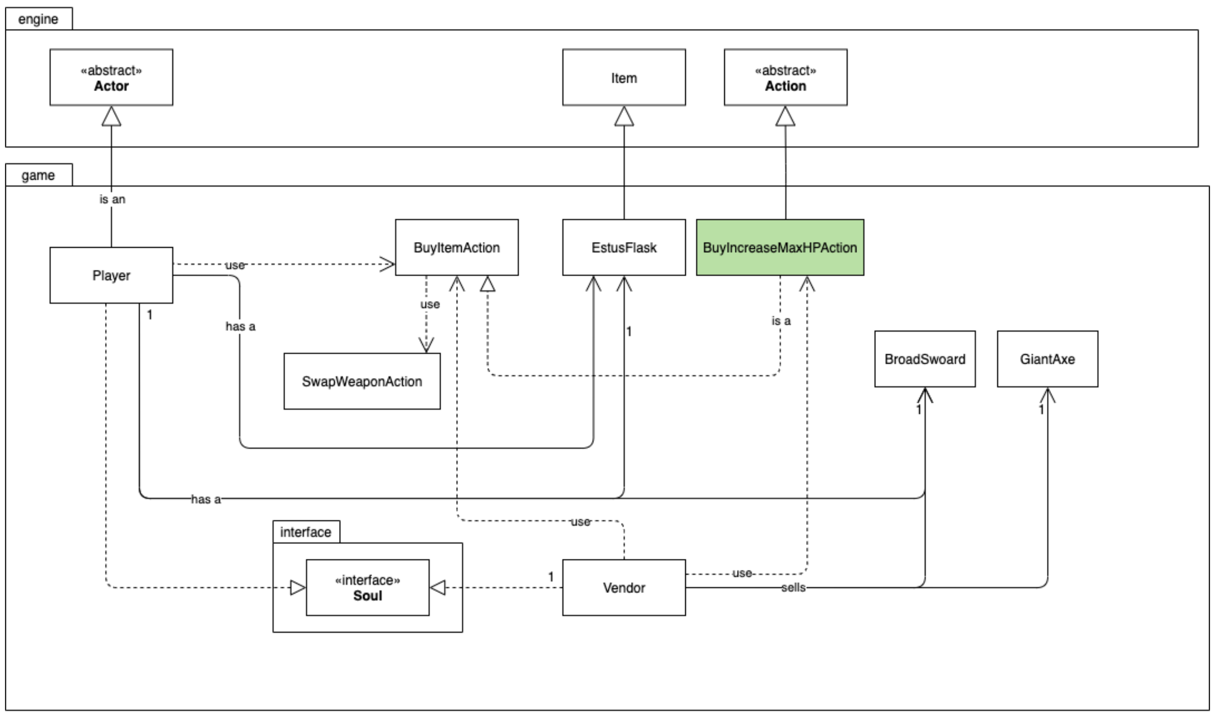
Assignment 2 Analysis and design

Group name: Lab2 Team6

Group members:
1. 31987850 Yong Liang Herr
2. 32579756 William Ho Swee Chiong
3. 31899412 Yap Yong Hong

Design rationale for UML Class Diagram - Player (Requirement 1)

Player and Vendor (Version 2)



Design rationale for UML Class Diagram - Player and Estus Flask(Requirement 1)

There are no new classes added in this UML diagram. See below for the action of drinking Estus Flask.

Design rationale for UML Class Diagram - Vendor (Requirement 8)

There is no new added class for this UML diagram.

The existing classes have covered all necessary actions for Players buying items from the Vendor. However, there are some changes to the relations between each class.

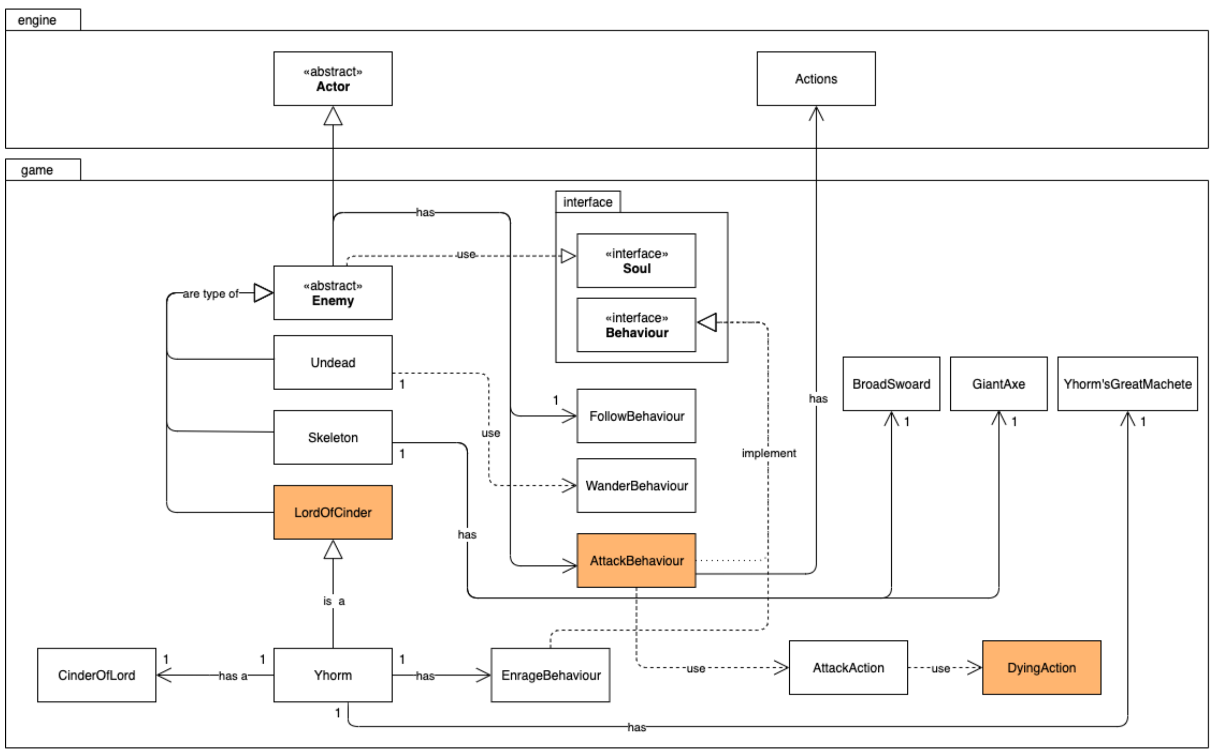
BuyItemAction follows the Single Responsibility Principle (SRP) to handle all buying actions between the Player and the Vendor. The BuyItemAction now depends on SwapWeaponAction when dropping the current weapon of the Player after buying an item. The implementation of BuyIncreaseMaxHPAction abides by Liskov Substitution Principle (LSP to ensure that buying increases maximum hitpoints performs similarly with buying other items.

The Vendor class is designed to fulfill the Open-Closed Principle (OCP) in terms of adding new weapons to the Vendor items hash map. When it is required to extend the list of the current items, we could just add it to the hashmap and it would be working as usual.

The transaction of souls is made possible by implementing the Dependency Inversion Principle (DOP). In this way, the BuyItemAction would not need to make the Player interact with the Vendor. Instead, they are insulated with a layer of abstraction but still able to perform transactions.

Design rationale for UML Class Diagram - Enemy (Requirement 4)

Enemy (Version 2)



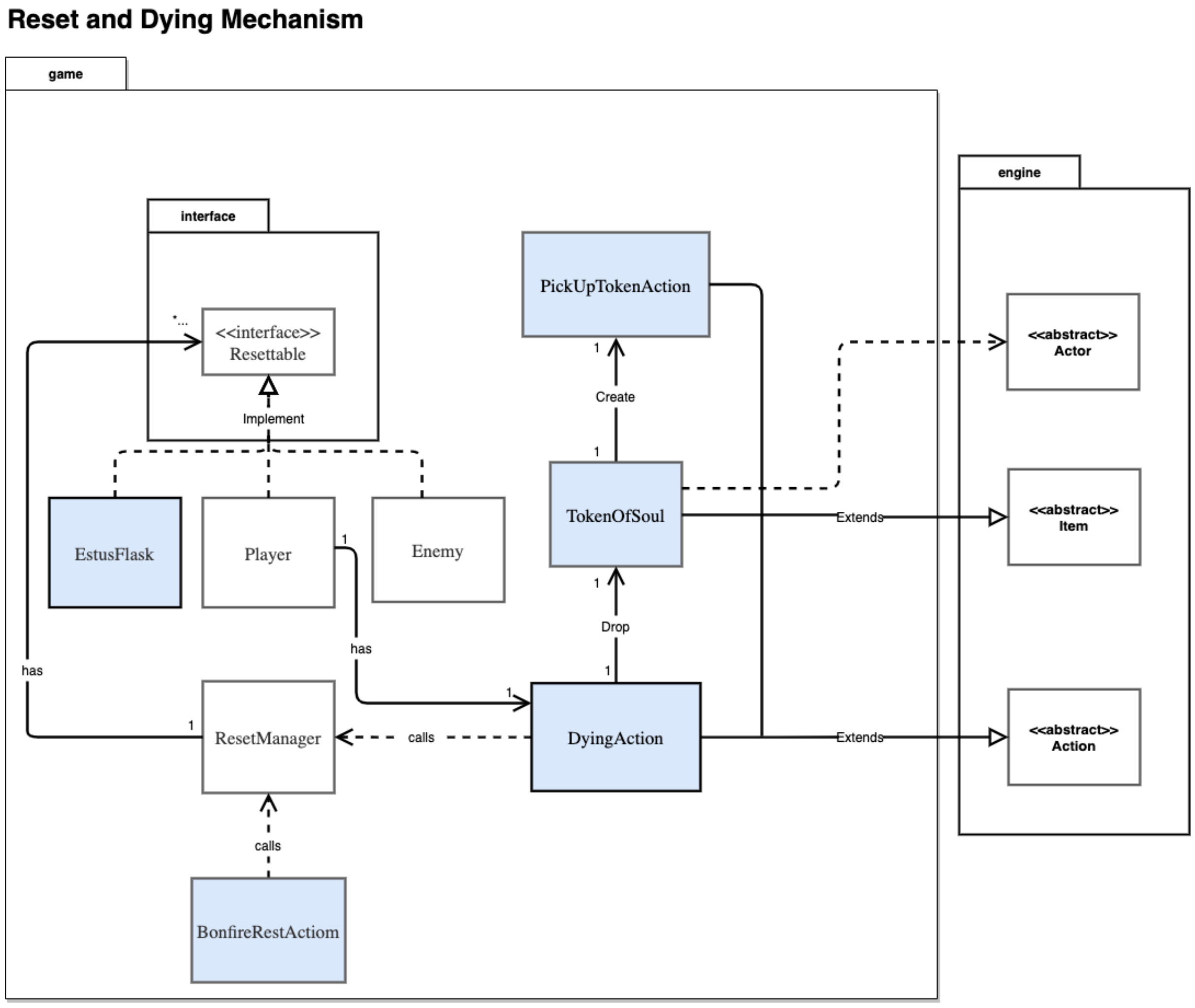
The newly added class are the ones colored in green. The changes to this graph are as the following:

First, we have the Enemy class being abstract so that every enemy like Undead, Skeleton, and LordOfCinder could inherit. We design this as so since all enemies have similar attributes like their behaviours, and also methods like adding behaviours, getting available actions, etc. Following LSP, we could expect all other methods that require an enemy would be having the same available methods and attributes. We could also greatly reduce associations and dependent relations between enemies and behaviour class. The abstract of the Enemy class also helps us to identify that only creating a specific type of enemy is possible.

Lord of Cinder is included in this new UML diagram, where Yhorm now inherits from it. The reason is that Yhorm is one of the Lord of Cinder, and there could be other Lords that could as well inherit it in the future. EnrageBehaviour follows SRP as it only handles the action return for Yhorm when it reaches enrage mode.

A new AttackBehaviour is created to handle all actions that could cause damage to the other actor like AttackAction and WeaponAction. This is again following SRP, where it assists the Enemy class to choose the attack actions to perform next when detecting a Player. Within the AttackAction, a DyingAction is used to handle the situation when the target that is being attacked is dead, grouping all necessary processes such as transferring souls, removing itself together. This helps to maintain the conciseness of AttackAction, and it as well follows SRP.

Design rationale for UML Class Diagram - Reset and Dying Mechanism (Requirement 6)



The new added classes are the one’s colored in blue.

First, we will look that there is a dependency relationship of the BonfireRestAction to ResetManager class. This dependency relationship is to allow the player to rest when the player approaches Bonfire. When the player chooses to rest it will then reset the player’s health point to maximum, refill the EstusFlask to the maximum charge and reset the enemy health, position, and skills. The reset action will be done either when the player chooses to rest, or the player is defeated or dead in the game. The only difference between the rest feature and the dead feature is the dropping of the soul. Therefore, in this case, We let the DyingAction class and Bonfire class both call the ResetManager class to perform the reset mechanism.

Next, we move on to the ResetManager class. The ResetManager class will contain a list of Resettable instances to perform the reset function. Therefore, in this case, we will make the EstusFlask class, Player class and Enemy class to implement the Resettable interface to allow it to be stored to the Resettable list in the ResetManager and perform the reset mechanism. In this feature, we followed the SRP for Estus Flask, Enemy and Player class. Although for enemy and player classes it handles a lot of functions, it still handles only one responsibility.

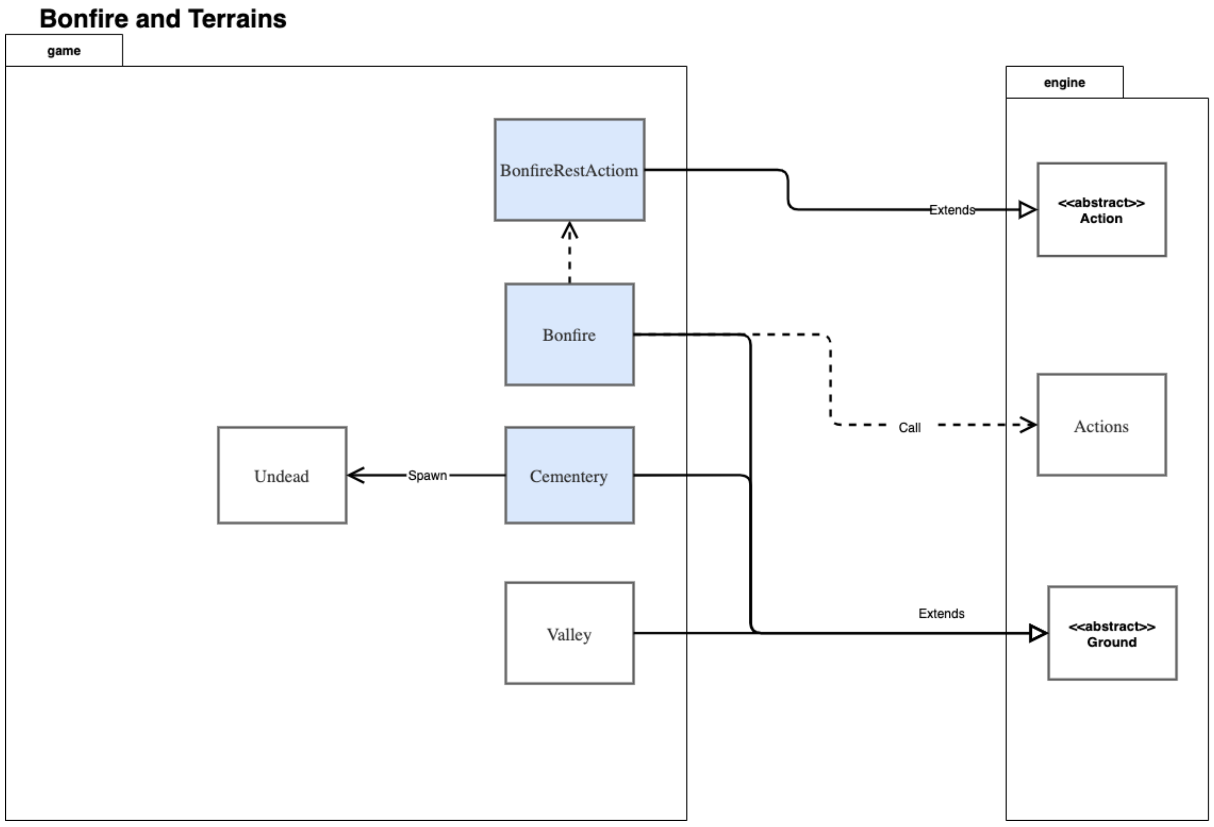
There is one method that we can override from the Resettable interface which is ResetInstance. In that class we reset the attribute for these classes to allow it to be called when we loop through the resettable list in the Reset Manager class. The resettable features achieve the OCP, by allowing us to make extensions for the class that need to be reset by implementing the Resettable interface.

When the player hit point is zero or less than zero, the player will call the DyingAction object to handle all the dying mechanisms which will call the ResetManager class to reset those three features. The DyingAction will extend the Action class. Next, the difference between the dying behavior with ResetManager is that DyingAction will manage the dropping of Token of Soul when the player is defeated so the DyingAction should associate with the TokenOfSoul class. We have achieved SRP for Dying Action, each class in the Dying and Reset Mechanism only has one responsibility. For example, DyingAction, there are plenty of functions in that class. However, it still handles one responsibility which handles the Dying Action when the actor dies.

In order to let the pick up action appear on the menu console when the player approaches the TokenOfSoul, we need to implement one PickUpTokenAction to handle the pick up action. In this class, we remove the token of soul from the map and also directly transfer the amount of soul in token of soul to the player.

In this case, we can see that the difference between Bonfire Rest mechanism, and Dying mechanism is that Dying will drop the token of soul but Bonfire Rest does not. Therefore, we implement the token of soul inside the dying mechanism instead of Reset Manager to reuse the ResetManager Code for Bonfire Rest Action.

Design Rationale for UML Class Diagram - Bonfire and Terrains (Requirement 2 & 5)



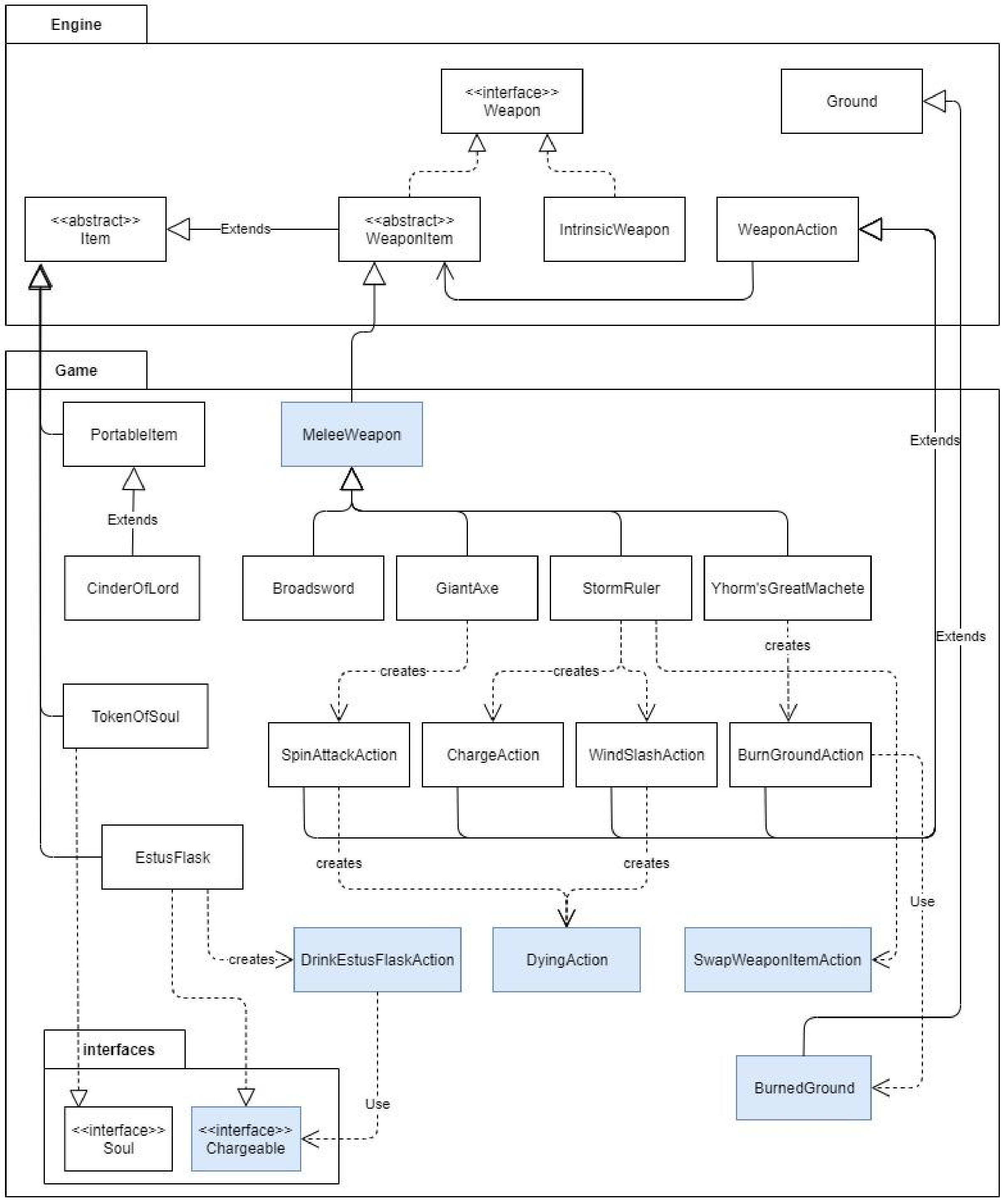
The new added classes are the one’s colored in blue.

When the game is started, the player will be spawned at the centre of the game map that is beside character B (Bonfire). When the player approaches the Bonfire, the rest option will need to appear in the menu console. Therefore, we need to create the Bonfire Rest Action to create the allowable action @see Dying and Reset mechanism. By following SRP, each Bonfire class and BonfireRestAction class both handle one responsibility.

In this game, there are two types of terrains scattered on the map. The first type of terrain is Cemetery. The cemetery has an association relationship with the undead to handle the 25% rate of the spawn of undead in the cemetery. The second type of terrain is the valley, when a player steps on a valley, the player will be killed instantly. Enemies are not allowed to step in the valley, therefore, we set the condition to prevent the enemies from stepping into the valley and killing them. Therefore, since all the terrains are made up of grounds on the map, therefore, both of the terrains will be extended from Ground abstract class. The cemetery has achieved the OCP, such that we are able to make extensions to spawn other creatures in the cemetery.

Design Rationale for UML Class Diagram - Item and Weapon (Requirement 3 & 7)

Weapon and Item (Version 2)



The new added classes are the one's colored in blue.

One of the significant changes we made is we added a Chargeable interface for items that have charges. This design applies the Open Closed Principle because it allows other items that have charges to implement this interface and extends its functionality if we added those items in the future.

Besides that, this design also applies the Dependency Inversion Principle because it allows other classes to add or deduct the number of charges of the items by using the method in this interface and hence achieves the condition that high-level modules should not depend on low-level modules. For example, by implementing this design, the DrinkEstusFlaskAction class is using the method of the interface to deduct the number of charges of Estus Flask instead of using the method in the EstusFlask class. It also reduces the dependency of the EstusFlask class and the DrinkEstusFlaskAction class.

The second difference is that all weapons are now extending the MeleeWeapon class. This design applies the Liskov Substitution Principle because the MeleeWeapon class is responsible for disabling the drop item action and all weapons should extend this class as all the weapons in this game cannot be dropped.