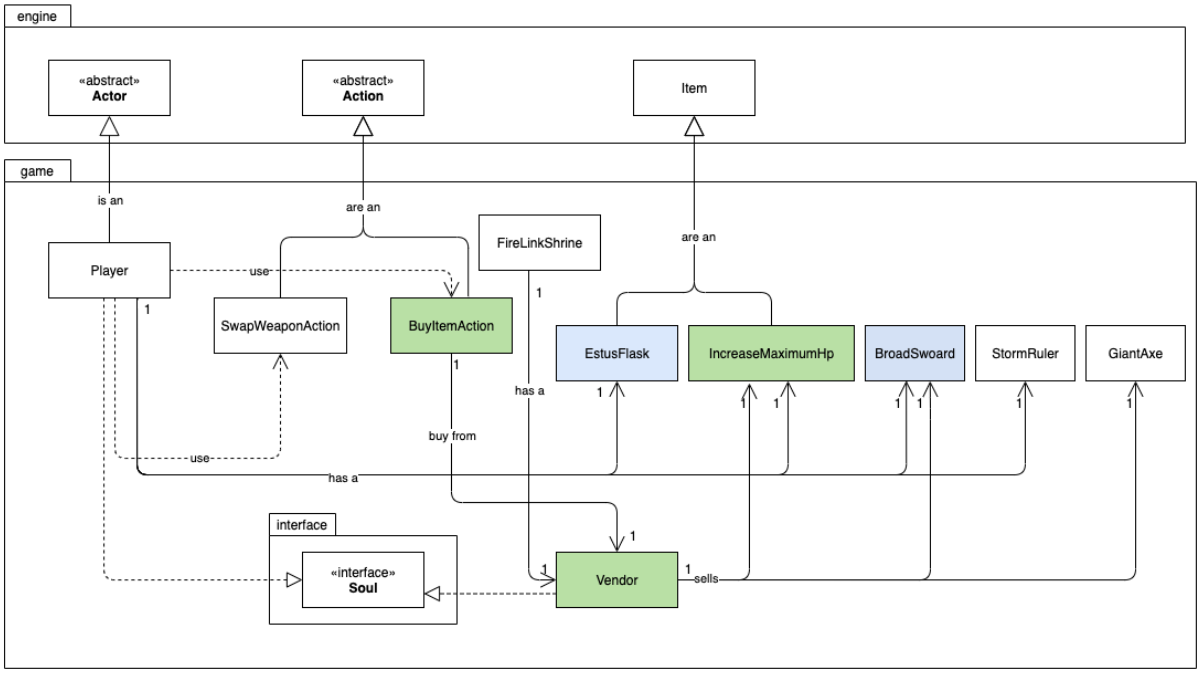
# Assignment 1 Analysis and design

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**




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

In the game, the player would be holding a weapon named StormRuler, and has a potion named EstusFlask. Since the weapon is one of the player's items in its inventory, we could conclude that there is an association relation between the player StormRuler, just like every other weapon and item. Another item is the EstusFlask. EstusFlask is extended from the Item abstract class instead of the Portable class as it cannot be dropped. EstusFlask is also an item within the player's inventory, hence, the association relation.

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

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

The Vendor is located only at the FireLinkShrine. It sells a total of three items including the IncreaseMaxiumumHp, Broadsword, and GiantAxe. To reduce dependencies between the

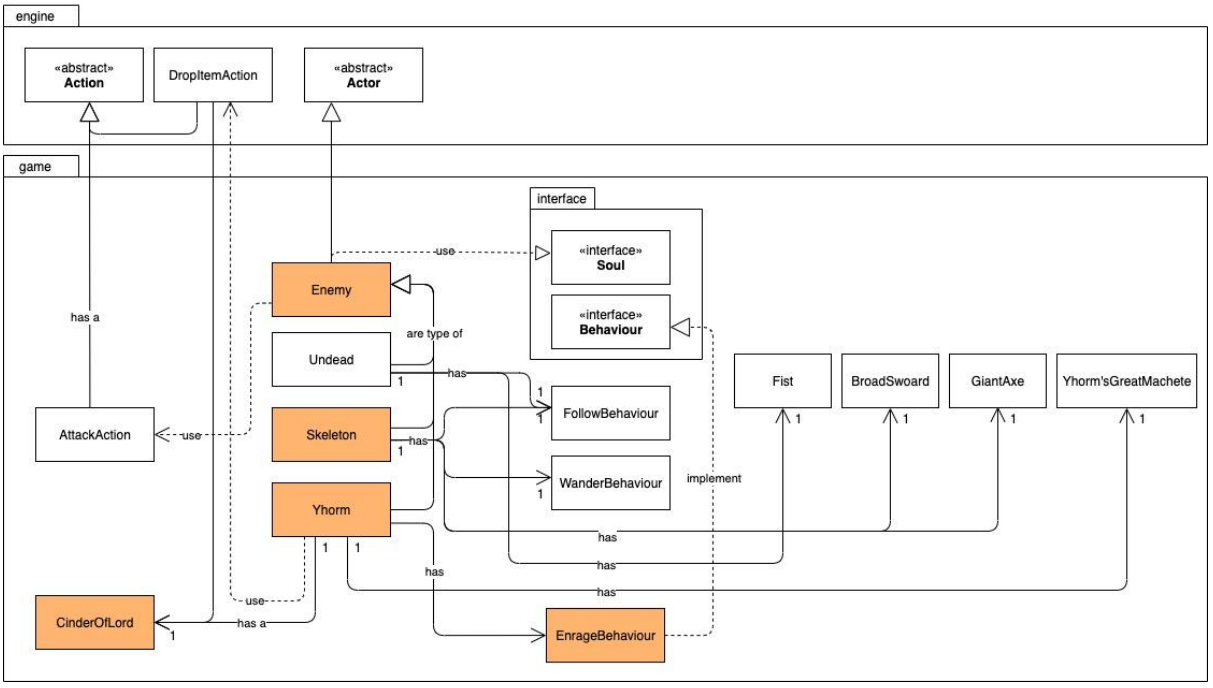
player and the vendor, we decided to extend another BuyItemAction from the Action abstract class that would handle all buying actions. That way, whenever the player tends to buy an item, it could be done through the BuyItemAction class.

The three items would be created and stored in a collection of the vendor class. Hence, there will always be an association relation between the items and the vendor. The reason for not including multiplicity is because the object would always be created when required (only three items would be created by the vendor). Whenever the buy method is called, it would simply return the item the user chose (the option should be an integer), by returning the corresponding item in the vendor's list of items. After receiving the item, BuyItemAction class would directly add them to the player's inventory.

As all bought items would be in the inventory of the player, there would also be an association relation between them. The player also has dependencies on the SwapWeaponAction, so that the player can swap its weapon when required.

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

**Enemy**




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

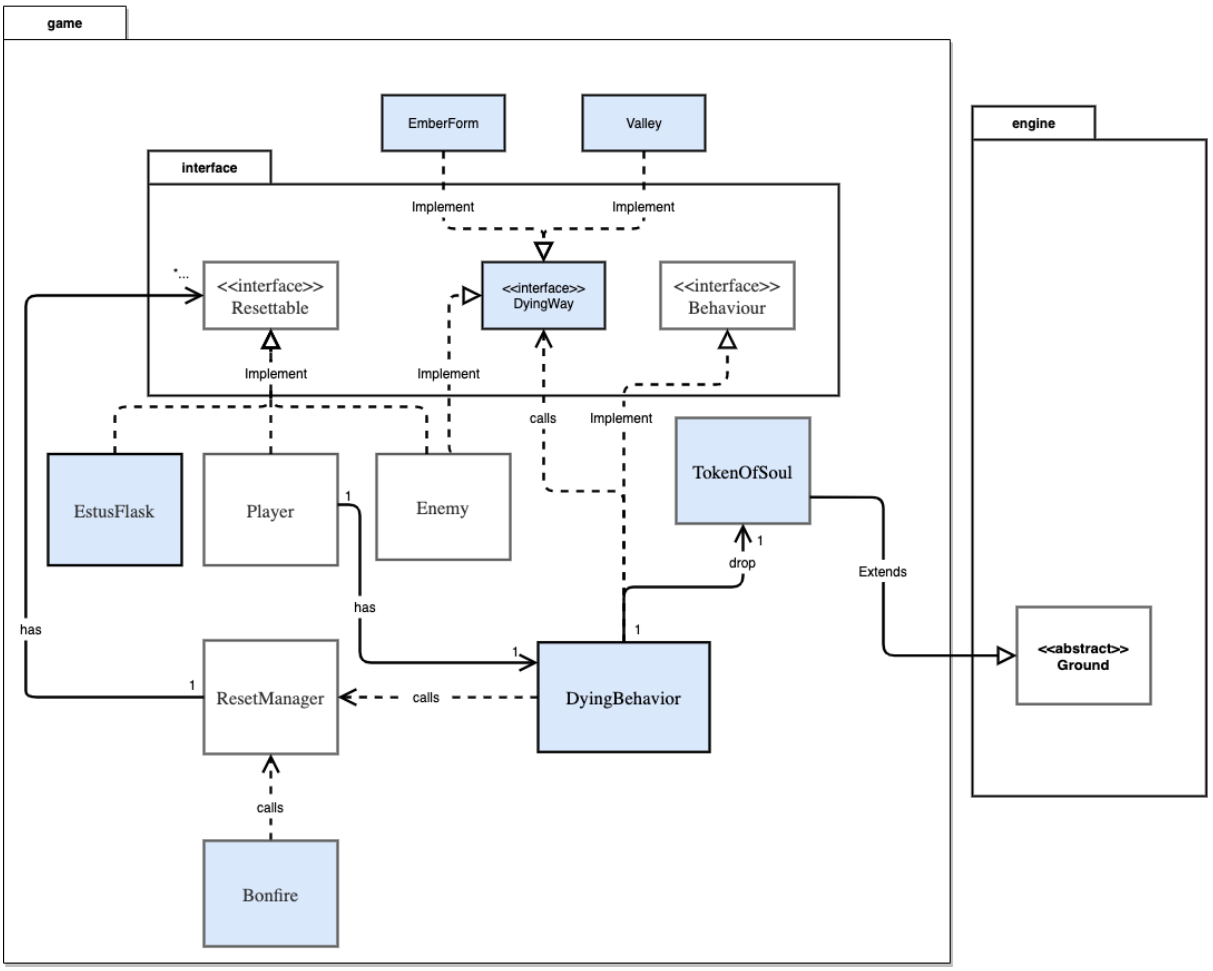
First, we have the Enemy class that is extended from the Actor class. Then, we could have all of our enemies to inherit from this Enemy parent class. The reason for this is to reduce dependencies for all enemies (skeleton, undead, etc) towards Soul and other actions like AttackAction. Instead, only the Enemy class is required to do so. As all enemies also need to be revived (see Reset UML) after the game is reset, we decided to create this Enemy class to abide by the DRY design practices.

Each enemy has their own weapon, so for every weapon they have, there would be an association relation between them and the weapon. Each enemy also has a collection of behaviours. Hence, there would be an association relation between them and their respective behaviours (FollowBehaviour for Undead, and FollowBehaviour, WanderBehaviour for Skeleton). Yhorm also has a special behaviour which is the EnrageBehaviour that would use its weapon special skill when it is in EnrageForm.

When Yhorm is defeated, a special item named CinderOfLord would be dropped. Yhorm would use the DropItemAction to perform this and since the item is portable, it is extended from the portable item class. The DropItemAction is also associated with CinderOfLord, since it is the item to be dropped by this action.

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

**Reset and Dying Mechanism**




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

First, we will look that there is a dependency relationship of the bonfire to ResetManager class. This dependency relationship is to allow the player to rest when the player approach Bonfire. When the player chooses to rest it will then reset the player's health point to

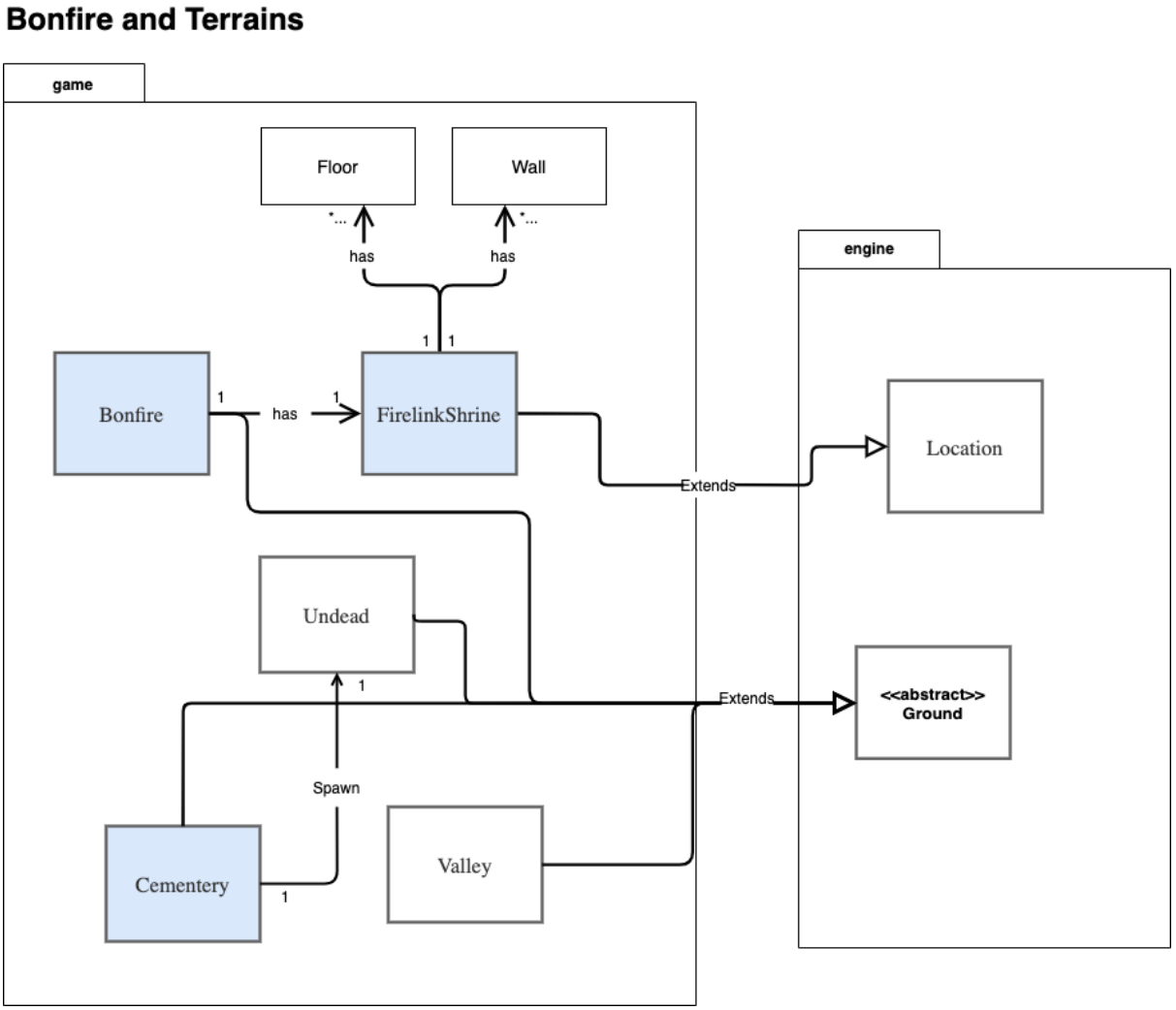
maximum, refill the EstusFlask to the maximum charge and reset the enemy health, position, and skills. The reset action will be done either when the player chooses to rest, or the player is defeated or dead in the game. The only difference between the rest feature and the dead feature is the dropping of the soul. Therefore, in this case, We let the DyingBehavior 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.

When the player hit point is zero or less than zero, the player will create a DyingBehavior object to handle all the dying mechanisms which will call the ResetManager class to reset those three features. Since it's also an actor's behaviour, the DyingBehavior will implement the Behavior class. Next, the difference between the dying behavior with ResetManager is that DyingBehavior will manage the dropping of Token of Soul when the player is defeated so the DyingBehavior should associate with the TokenOfSoul class. The TokenOfSoul class will extend the abstract class of ground since it is shown as a $ sign in the game map.

Lastly, the player will have different dying ways. Now, we only have 3 known dying ways that have a subtle different feature in the dying behavior. Therefore, we let the DyingBehavior class be dependent on the DyingWay interface which is being implemented by EmberForm(Burn), Valley(Fall) and Enemy(Slam).

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

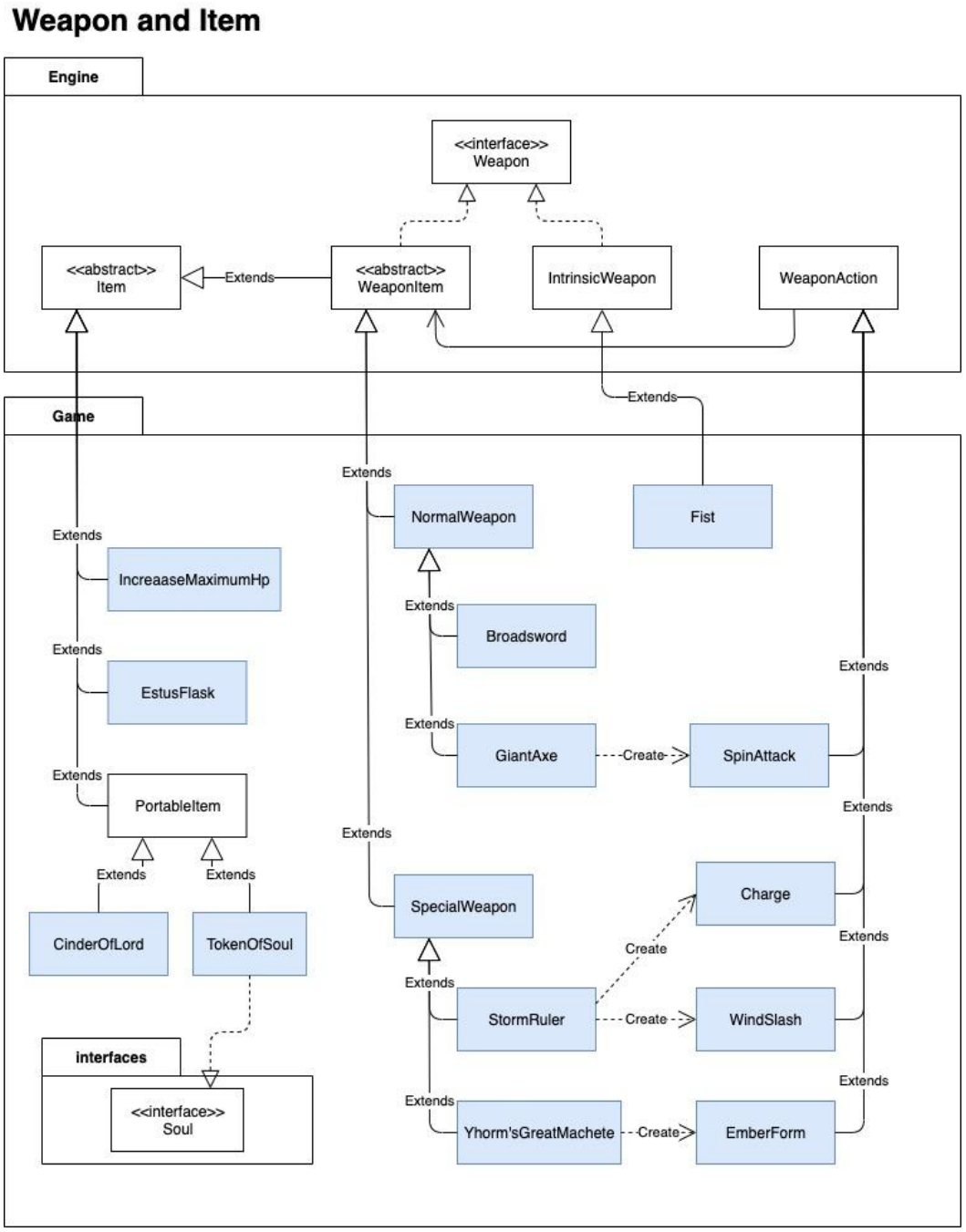



The new added classes are the one's colored in blue.

When the game is started, the player will be spawned at the centre of the game map that is beside character B (Bonfire). B is inside a zone called FirelinkShrine. The FirelinkShrine is made up of walls and floors that prevent the enemy from stepping in. Therefore, the relationship between Bonfire with FirelinkShrine is association and the relationship between FirelinkShirine with Wall and Floor is also association.

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. 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.

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




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

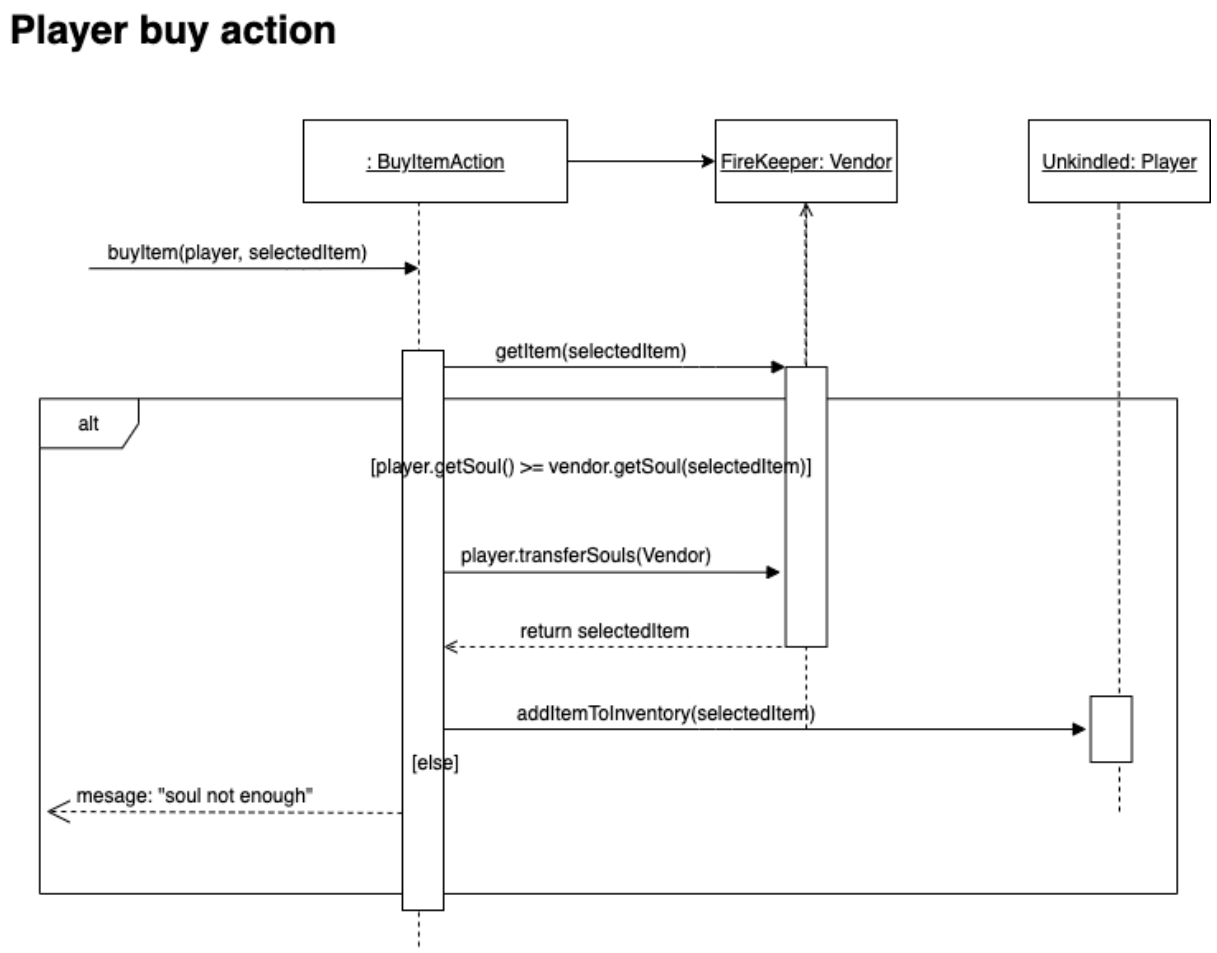
All classes that inherit the abstract class Item must be an item that can be added to the player's inventory and has methods to implement its capabilities or actions. For example, the IncreaseMaximumHp class has the capability to increase the player's maximum hit points while the EstusFlask class has the capability to heal the player. The PortableItem class is the base class for the items that can be dropped by players or enemies and picked up by players. The CinderOfLord class and TokenOfSoul class inherit the PortableItem class because they both can be dropped and picked up. Besides that, the TokenOfSoul class also

implements the Soul interface because the tokenOfSoul object is considered as an instance of soul that can transfer its soul to the player that picks it up.

The NormalWeapon class and SpecialWeapon class inherit the WeaponItem class because they can be used as a weapon. The NormalWeapon class is the base class for weapons that can be used by both players and enemies. Hence, the Broadsword class and the GiantAxe class inherit the NormalWeapon class. On the other hand, the SpecialWeapon class is the base class that can only be used by a specific actor. Hence, the StormRuler class and Yhorm'sGreatMachete class inherit this class because the StormRuler can only be used by players while the Yhorm'sGreatMachete can only be used by the Lord of Cinder. The aim of this design is to ensure that the usage of each class is clear.

Some of the weapons have active skills while some do not. The SpinAttack class, Charge class, WindSlash class and EmberForm class are the active skills of weapons and all of them inherit the abstract class WeaponAction. When a player wants to use the active skill of a weapon, the weapon object will give the player its skill. For example, if a player wants to use the spin attack of giant axe,  the giantAxe object will give the player the spinAttack object. Therefore, the player does not need to know about the SpinAttack class. This design implies the Reduce Dependency Principle because it breaks the dependency relationship of the Player class and the SpinAttack class.

**Sequence diagram 1 - Player buy action from vendor**

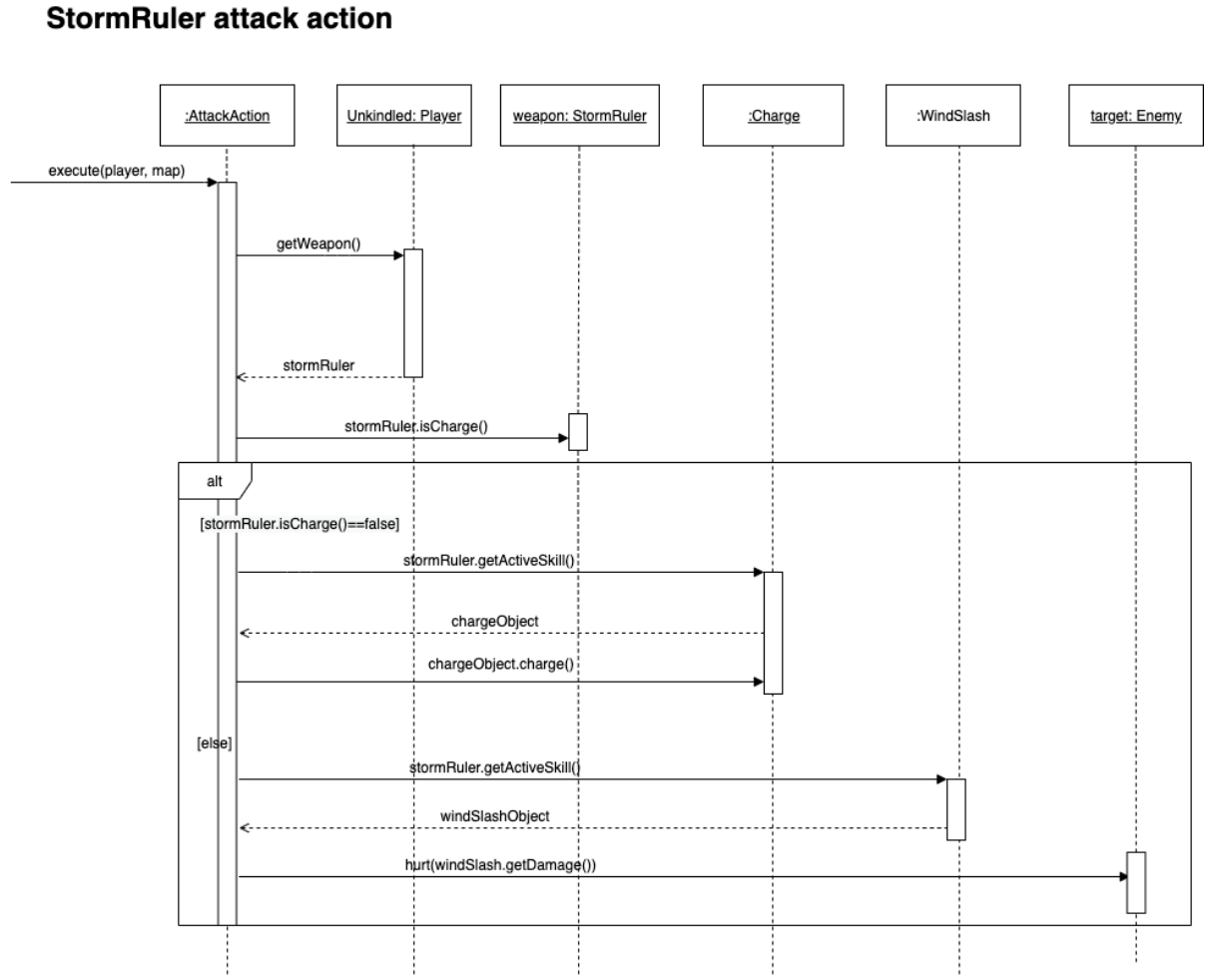



When the Player - Unkindled approaches the Vendor - FireKeeper, an option of BuyItemAction will appear in the menu. When the action is called, the Player can then select the item he wants from the menu displayed. The BuyItemAction class would then obtain the selected item from the vendor, and check if the souls of the Player are sufficient to exchange for the selected item.

If the condition is met, the player's soul would be transferred using the transferSoul method to the vendor. In return, the vendor would transfer back the selectedItem instance back to the BuyItemAction class. It would then add the item to the player's inventory list through the addItemToInventory() method.

Else if the condition is not met, it would prompt a message in the console, informing the user that the soul is not enough.

**Sequence diagram 2 - Storm Ruler attack action**




When the player performs the attack action through the AttackAction class, the class would use the getWeapon() method from the player to retrieve the current weapon of the player. Assuming that the current weapon is the StormRuler, it would then check whether the StormRuler is charged.

If it is not charged, StormRuler would get its active skill using the getActiveSkill() method. In this case, the getActiveSkill() method would return a Charge object. The player can use the object's charge method to charge StormRuler.

Else if the StormRuler is fully charged (when it has three charges), the isCharge method would return true. In this case, the StormRuler getActiveSkill() method would return a WindSlash object. This object can then be used to perform charged damage on its enemy through the player's hurt() method.