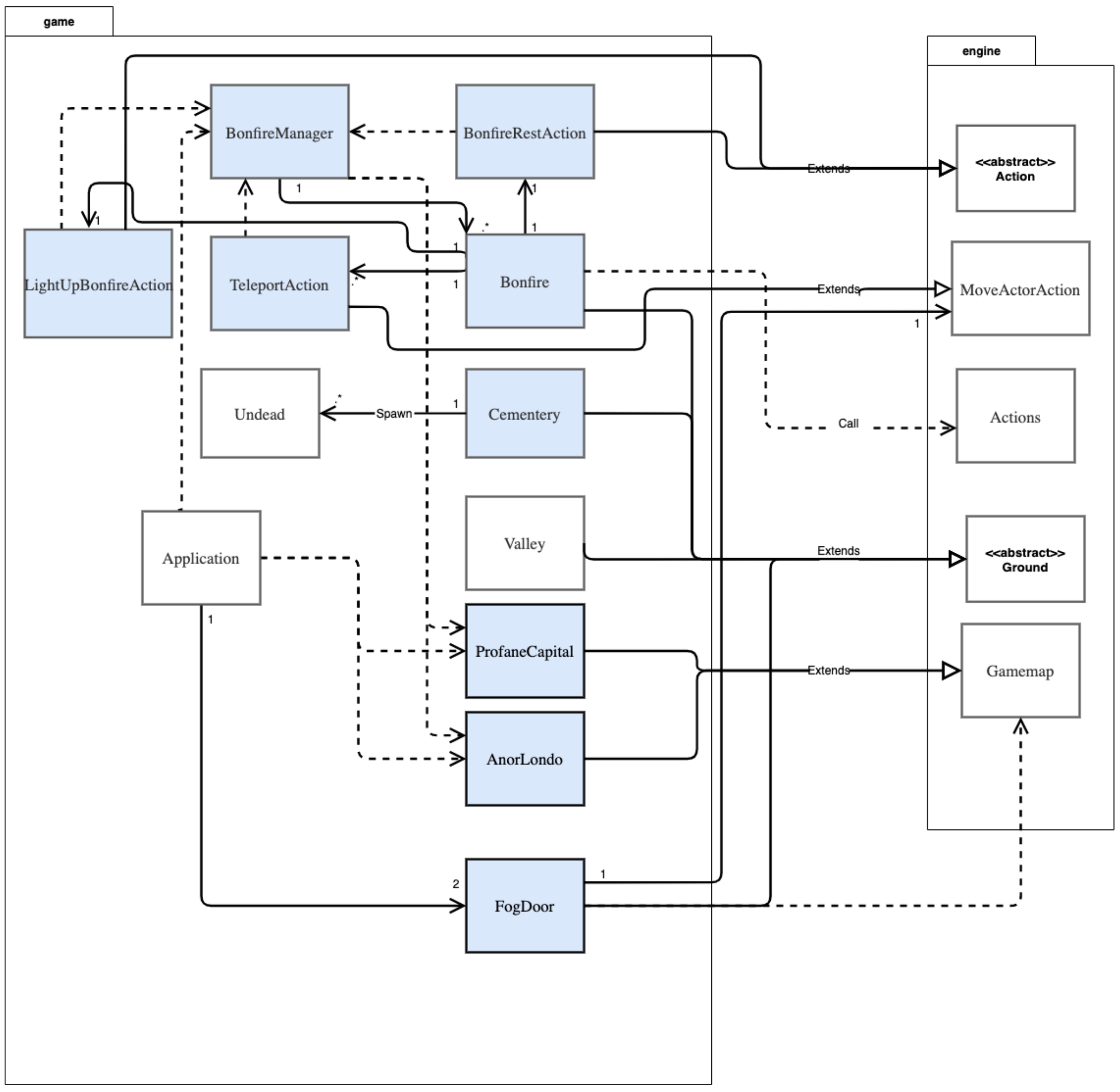
Assignment 3 Design O' Souls

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 - New Map and Fog Door, Bonfire (Requirement 1 and 2)

GameMap, Bonfire, FogDoor



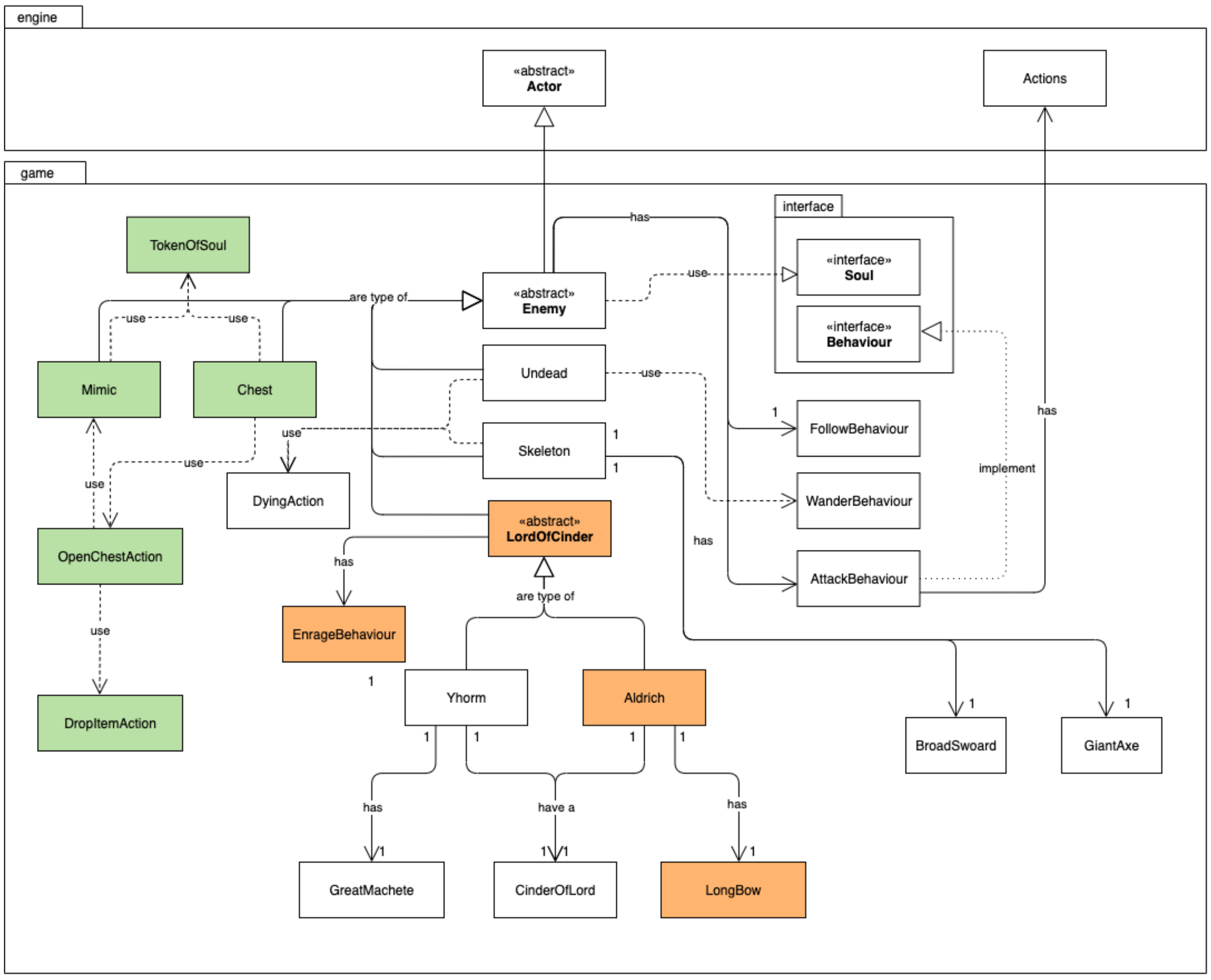
The newly added and edited class has been marked blue in the diagram.

For requirement 1, we created two new classes that extend from Game Map which are ProfaneCapital and AnorLondo. The initialisation of the game map and set up of the map is done in the new classes. These two maps will be initialised once throughout the program, so for simplicity, we use the static factory method in these two maps. We have achiched the Open Closed Principle that we can add new game maps by extension and avoid the modification. In order to move the player from one map to another, we had implemented FogDoor which extends the MoveActorAction to move the player to another map’s FogDoor. The FogDoor has a dependency relationship with the GameMap to generate the location of the Bonfire by coordinates.

For requirement 2, we created a BonfireManager class to store and bind the location of the bonfire with bonfire. In this class, it is also responsible for updating the last interacted Bonfire location. We have achieved the SRP that BonfireManager has only one responsibility but a few features. Now, let us look at the Action that Bonfire can perform. The actions include Bonfire Rest Action, Teleport Action and LightUpBonfire Action. For RestAction and LightUpBonfire action, we need to update the last interacted bonfire with the current Bonfire, therefore, it shares dependency relations with the BonfireManager class to keep track of the last interacted Bonfire. On the other hand, when teleport action is performed, we need to update the last interacted Bonfire to the target Bonfire. Therefore, it also has a dependency with the BonfireManager class. Besides, teleport action also extends from the MoveActorAction class to perform the actor teleport function. In this feature, we achieved the Open-Closed Principle that we can add more Action to the Bonfire without any modification of the program.

Design rationale for UML Class Diagram - Aldrich the Devourer (Requirement 3)

Enemy (Version 3)



The newly added/edited class are the ones in orange.

For this requirement, we have Aldrich added to the UML diagram. But first, let us discuss the changes we made for its parent class - Lord of Cinder. In this UML diagram, we have to change the Lord of Cinder class to be abstract. This is because we do not need to create Lord of Cinder objects, instead, we create specific Lord of Cinder children's enemy objects like Yhorm and Aldrich. Besides, as both Lord of Cinder's children would activate enrage behaviour, we decided to include the behaviour attribute and related methods in the parent class instead to prevent code length smells. Thus, we can observe there is a new association relation between the Lord of Cinder class and EnrageBehaviour.

Since the design of Lord of Cinder has already abided by the Liskov Substitution Principle in the previous design, Yhorm and Aldrich would act similarly, indicating that all Lord of Cinder unique behaviours like enrage behaviour implementations would be applied to Aldrich as well.

Design rationale for UML Class Diagram - Mimic/Chest (Requirement 4)

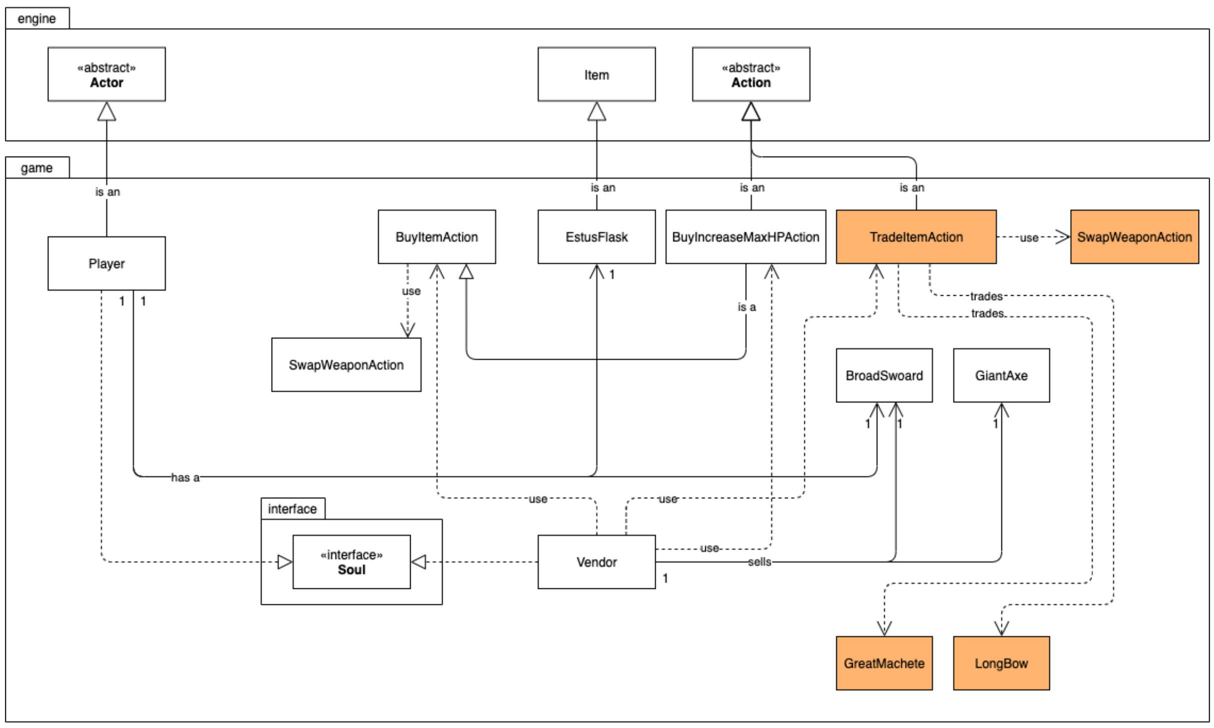
The newly added/edited class are the ones in green.

Mimic has a 50% chance of appearing on the map when the chest is opened. As Mimic is also an Enemy, we could simply extend it from the Enemy class and it would be able to perform all Enemy behaviours such as FollowBehaviour as AttackBehaviour. Following the Liskov Substitution Principle, we could expect Mimic to act as how other enemies behave. Mimic also uses Token of Soul by adding a random number of it to its own inventory. Hence, we can observe that there is a dependency relation between Mimic and Token of Soul.

Chest utilize OpenChestAction to handle all actions after opening the chest (following SRP), including dropping random number of token of souls or placing a Mimic. Since Chest itself contains random number of Token of Soul, we could see that it also has a dependency relation with it. For the action of dropping the Token of Soul in the OpenChestAction class, we have the DropItemAction to handle this. Hence, the dependency relation between OpenChestAction with DropItemAction.

Design rationale for UML Class Diagram - Trade Cinder of Lord (Requirement 5)

Player and Vendor (Version 3)

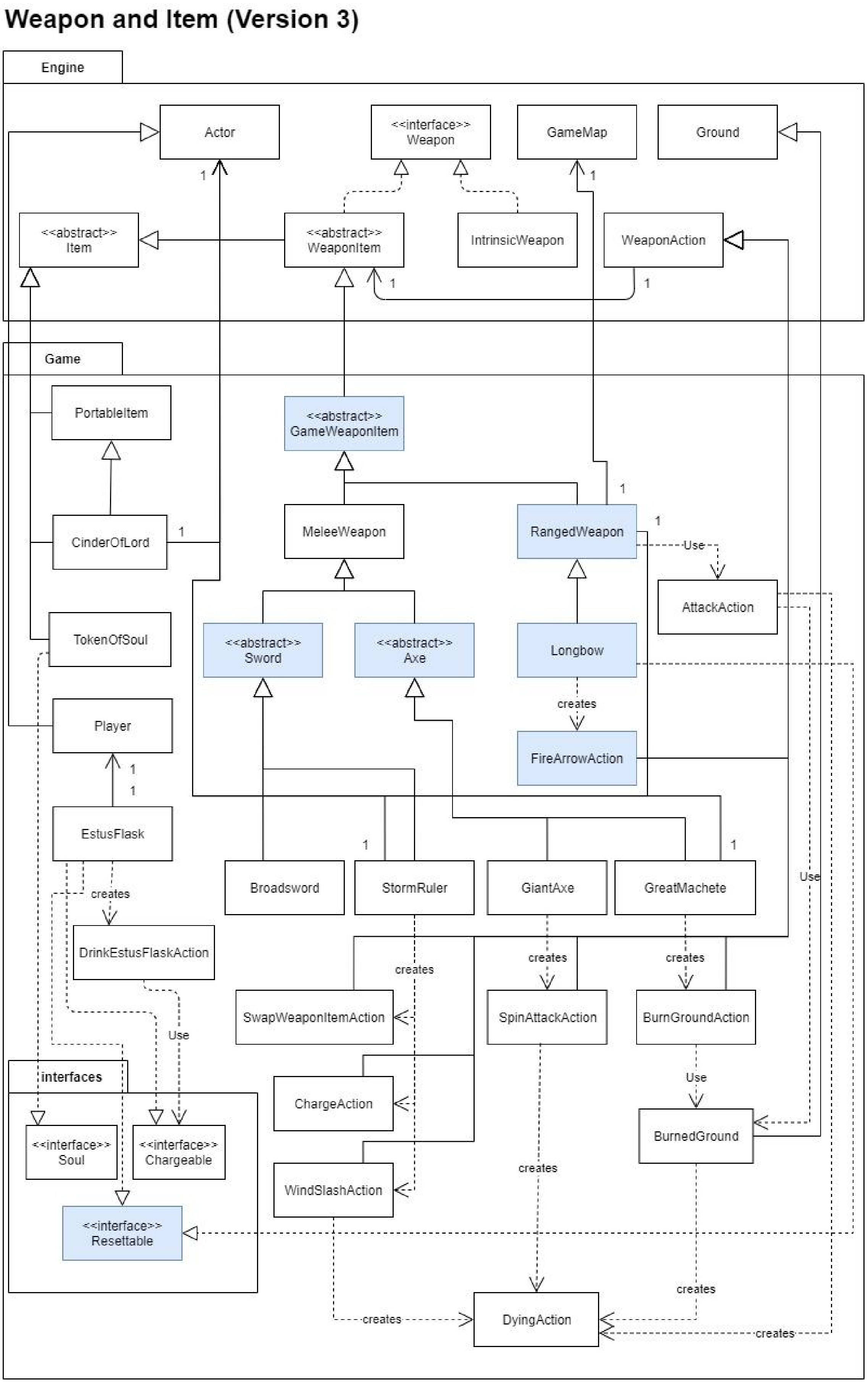


The newly added/edited class are the ones in orange.

For the trading of Cinder of Lord, TradeItemAction is used to handle all trading actions between the Player and the Vendor, following the Single Responsibility Principle (SRP).

We can see that there are dependencies between TradeItemAction and both Lord of Cinder's weapon and SwapWeaponAction. The reason is that TradeItemAction uses SwapWeaponAction to perform the weapon swapping action, and these weapons are created upon the Vendor finished checking which Cinder of Lord the user is providing.

Design Rationale for UML Class Diagram - Weapon (Requirement 3)



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

Firstly, we added the abstract class GameWeaponItem. The function of this abstract class is to disable the drop item action of the weapon and act as a parent class for all weapons in this game as all weapons cannot be dropped.

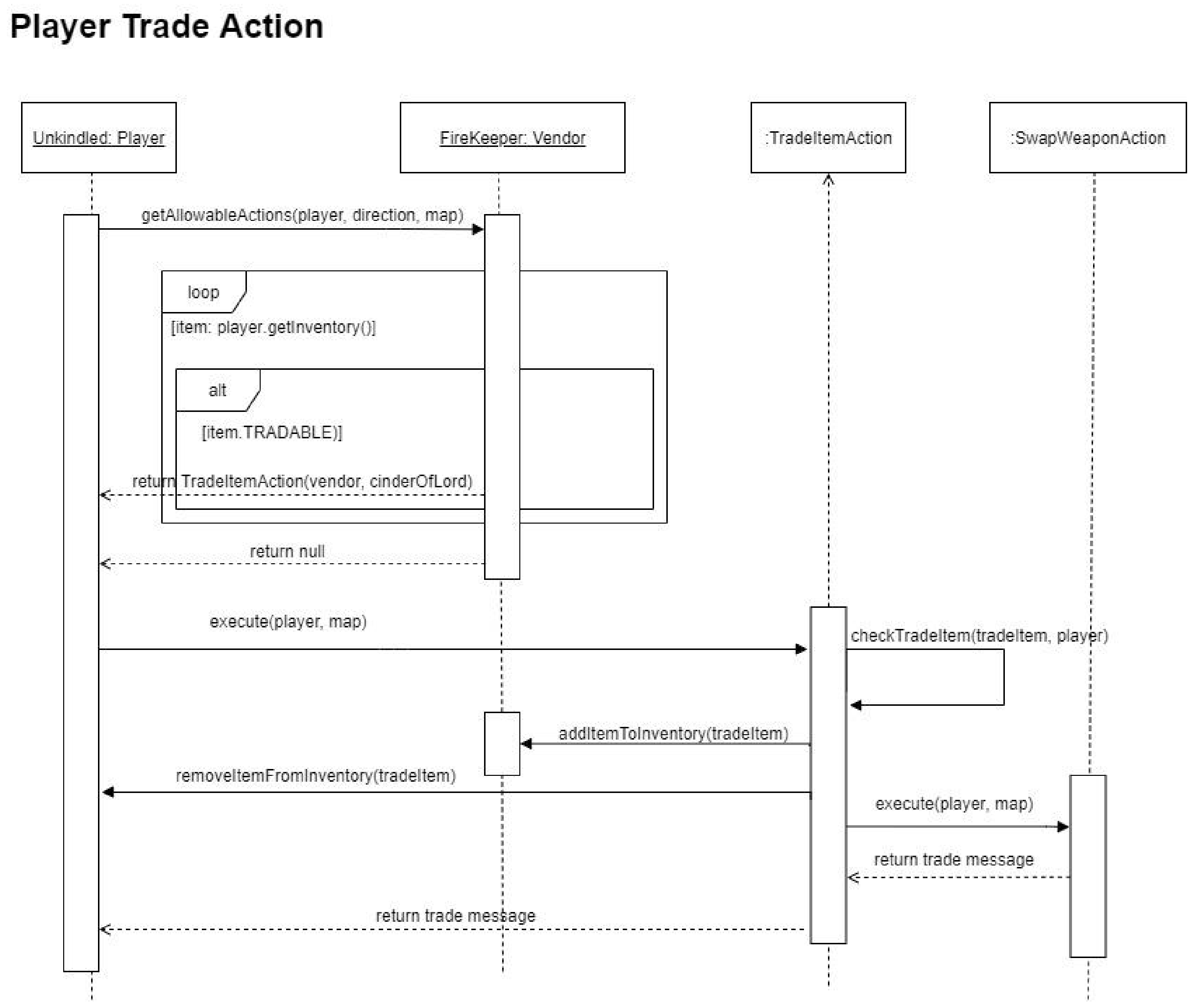
Next, we also added the class RangedWeapon. The difference between the class RangedWeapon and the class MeleeWeapon is it takes an extra parameter in its constructor that represents the range of the weapon. This design applies the Single Responsibility Principle because the weapons that extend the class MeleeWeapon can only attack the enemies in its surrounding while the weapons that extend the class RangedWeapon can attack the enemies within its range.

There is an association relationship between the class RangedWeapon and the class Actor and the class GameMap because the ranged weapon needs to know the location of its holder and find the enemies within its range.

The class Longbow that extends the class RangedWeapon also implements the interface Resettable so that its skill can be deactivated when the player dies. This design applies the Interface Segregation Principle.

Besides that, we also added the abstract class Sword and the abstract class Axe. The difference between these two classes is the weapons in the Sword class have the critical strike as its passive skill while the weapons in the Axe class do not. This design applies the Open Close Principle because it allows us to add more weapons and extend their functionalities from these two classes.

Design Rationale for Sequence Diagram - Player Trade Action (Requirement 5)


Player Trade Action
Unkindled: Player
FireKeeper: Vendor
:TradeItemAction
:SwapWeaponAction
getAllowableActions(player, direction, map)
loop
[item: player.getInventory()]
alt
[item.TRADABLE)]
return TradeItemAction(vendor, cinderOfLord)
return null
execute(player, map)
checkTradeItem(tradeItem, player)
addItemToInventory(tradeItem)
removeItemFromInventory(tradeItem)
execute(player, map)
return trade message
return trade message

When the player - Unkindled approaches the vendor - FireKeeper, it will call the getAllowableActions from the vendor. If the player has the cinder of lord, it will return a list of actions including the TradeItemAction to the menu options. If the player chooses to execute the TradeItemAction, it will first check the cinder of the lord item whether it is from the Yhorm or the Aldrich, then it will add the weapon to the player's inventory accordingly by executing the SwapWeaponAction. After that, a trade message will be returned to indicate the status of the trade action.