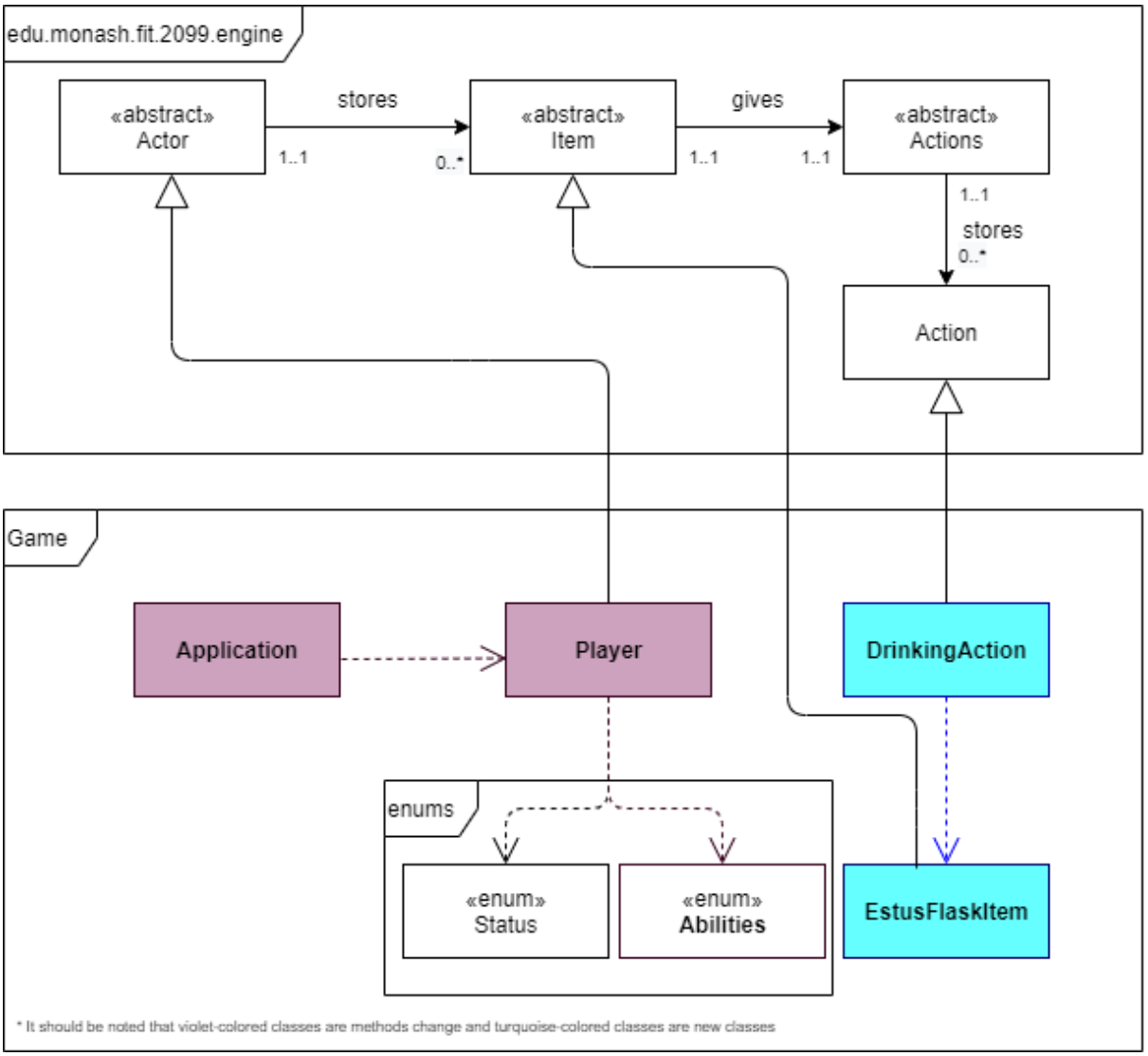
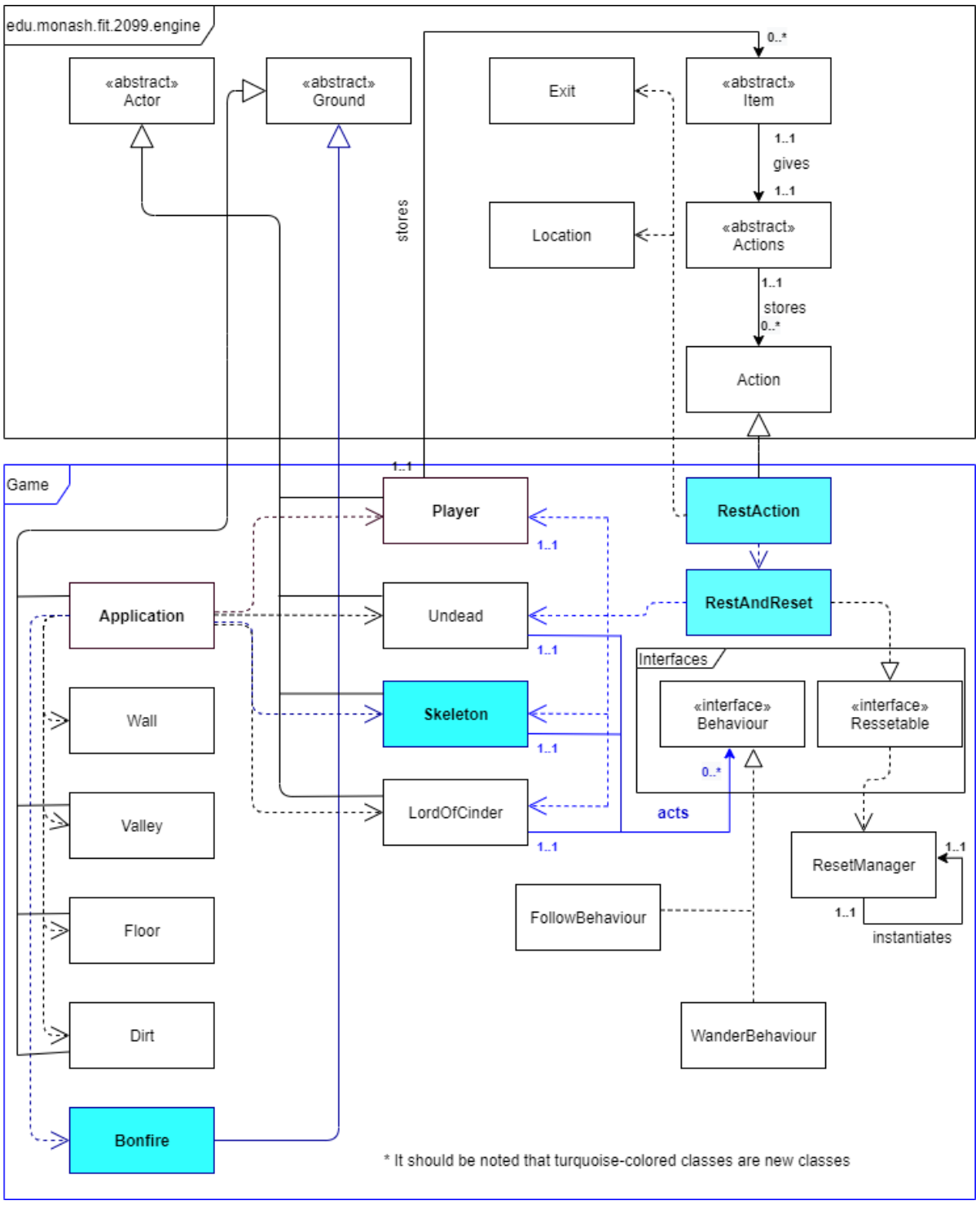
# Design Document

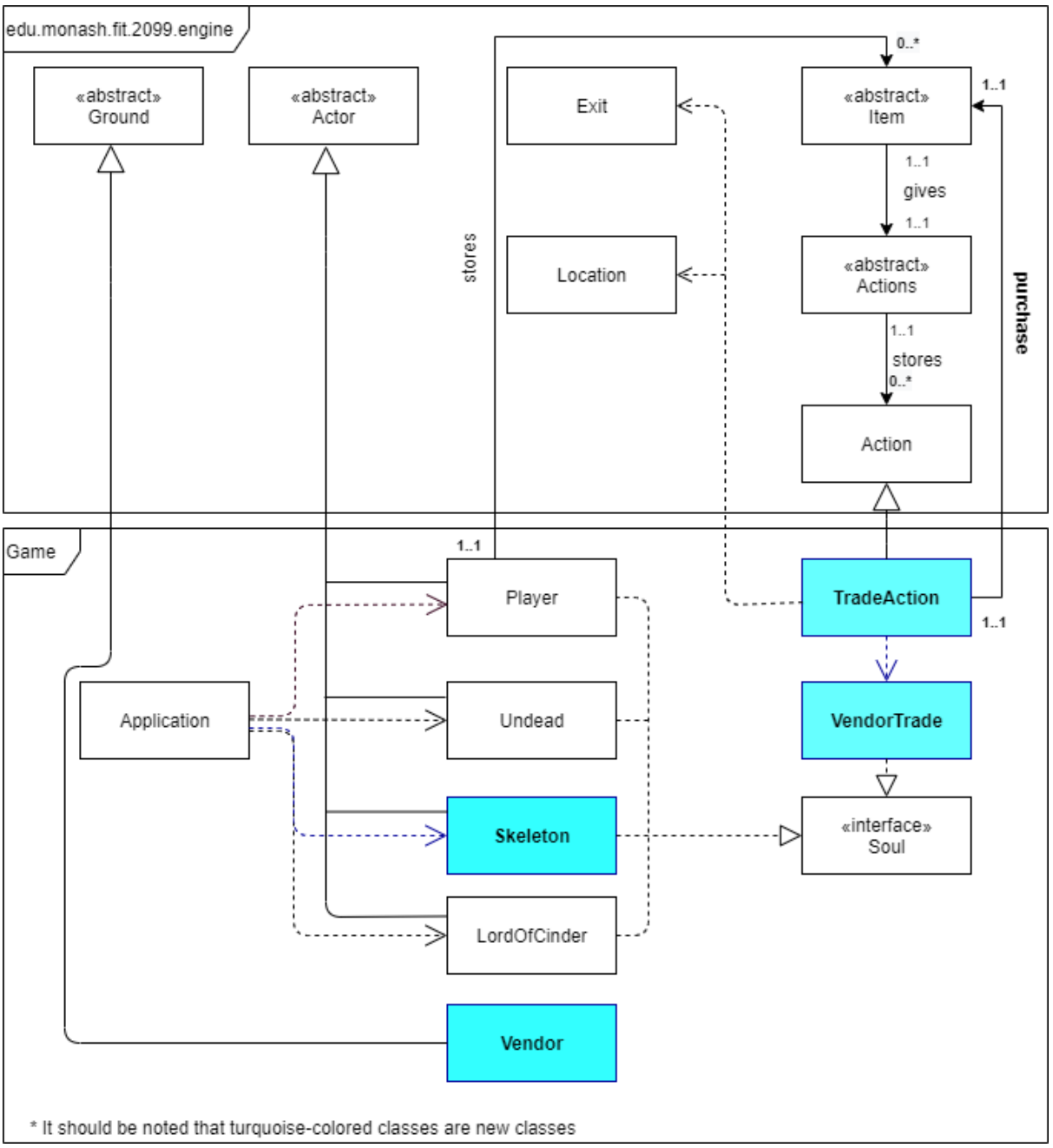
## Requirement 1




## Requirement 2

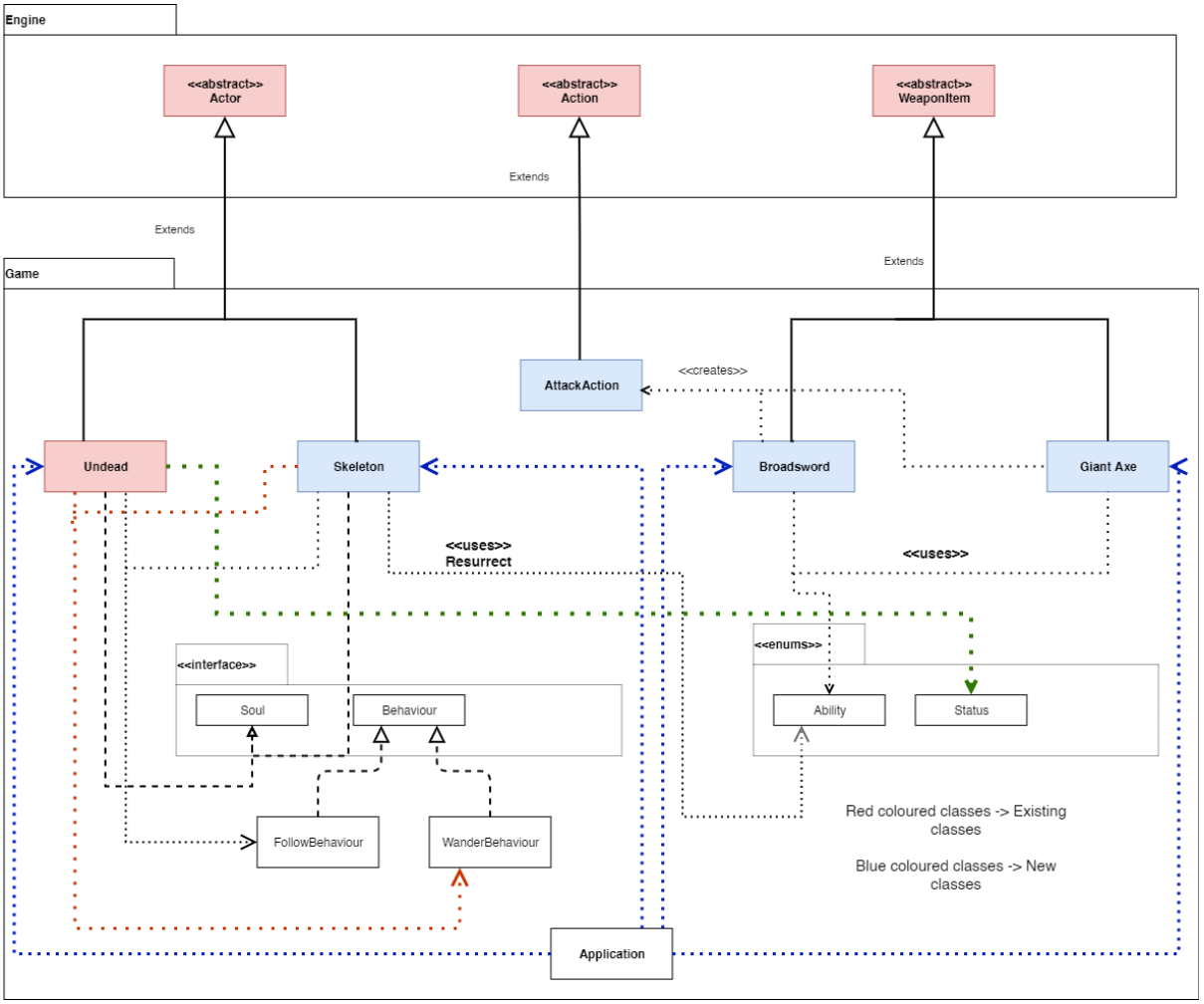



## Requirement 3

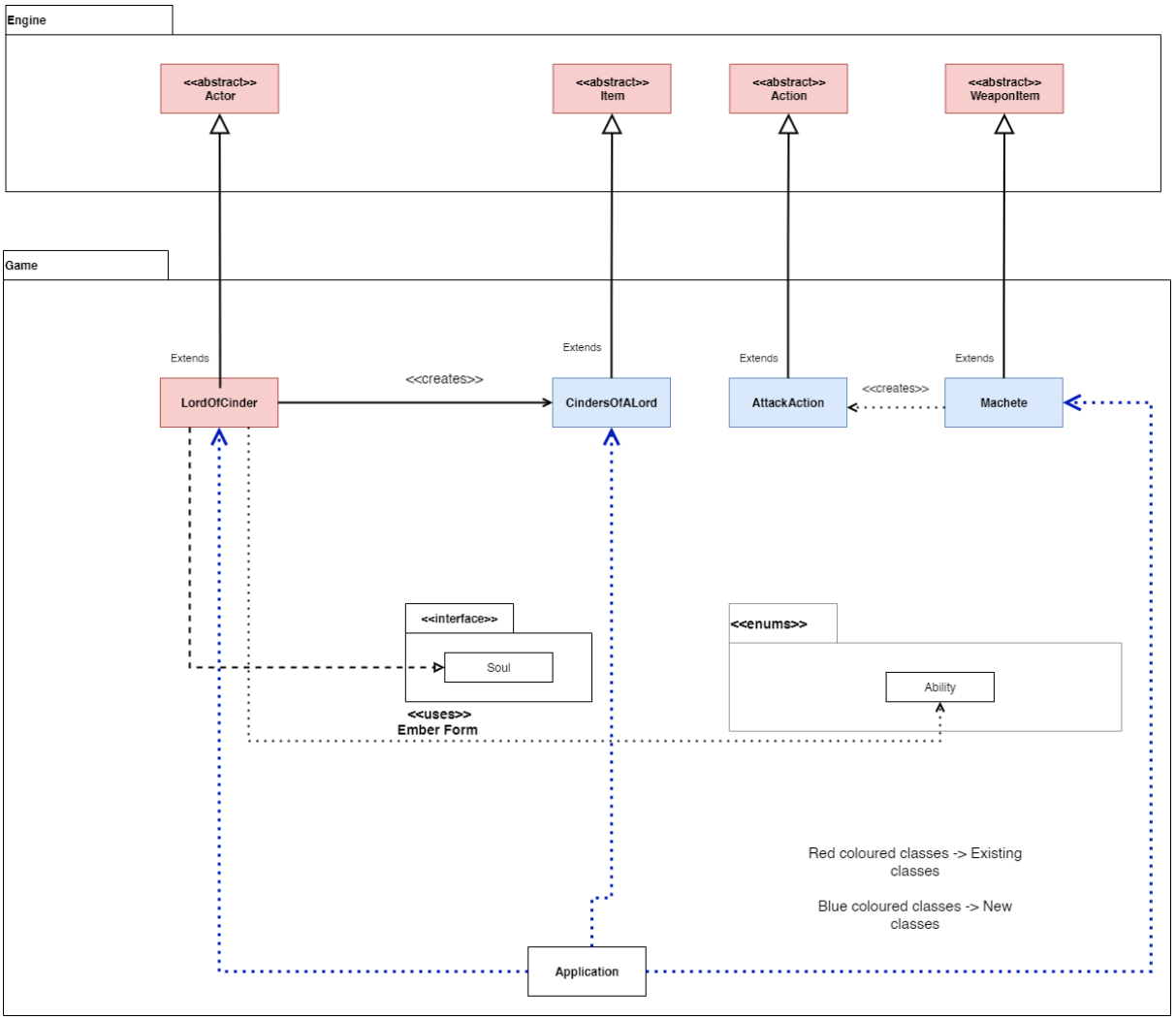



## Requirement 4

Skeleton and Undead UML

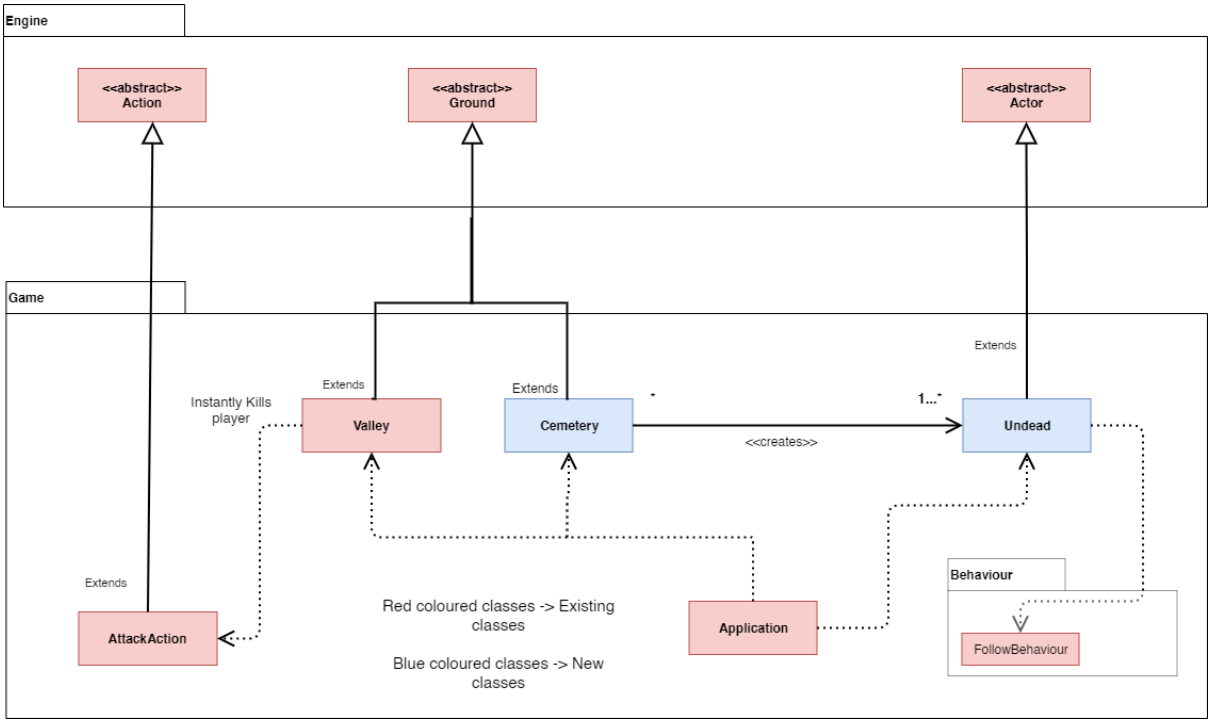

Engine
<<abstract>>
Actor
<<abstract>>
Action
<<abstract>>
WeaponItem
Extends
Extends
Extends
Game
AttackAction
<<creates>>
Undead
Skeleton
Broadsword
Giant Axe
<<uses>>
Resurrect
<<uses>>
<<interface>>
Soul
Behaviour
<<enums>>
Ability
Status
FollowBehaviour
WanderBehaviour
Application
Red coloured classes -> Existing classes
Blue coloured classes -> New classes

Lord Of Cinder (Boss) UML



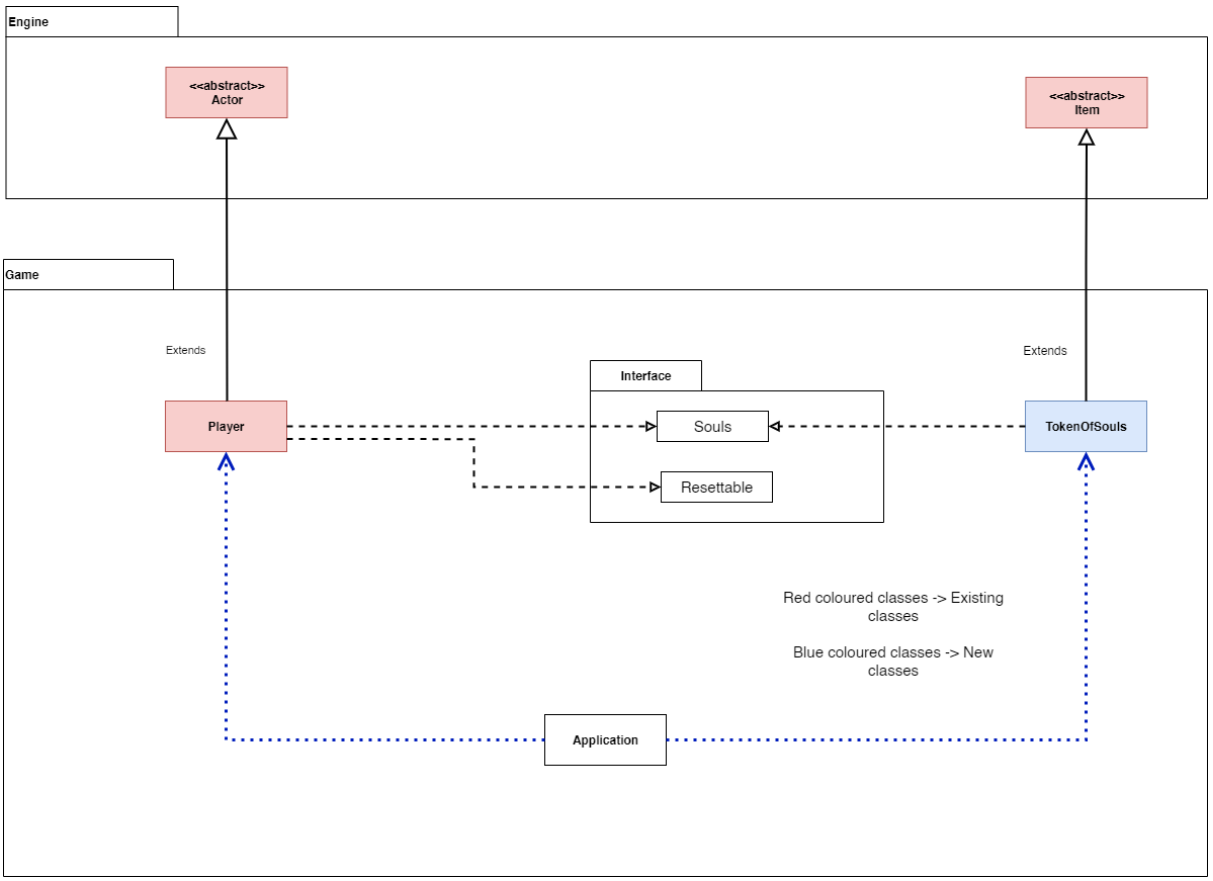
## Requirement 5

Terrain

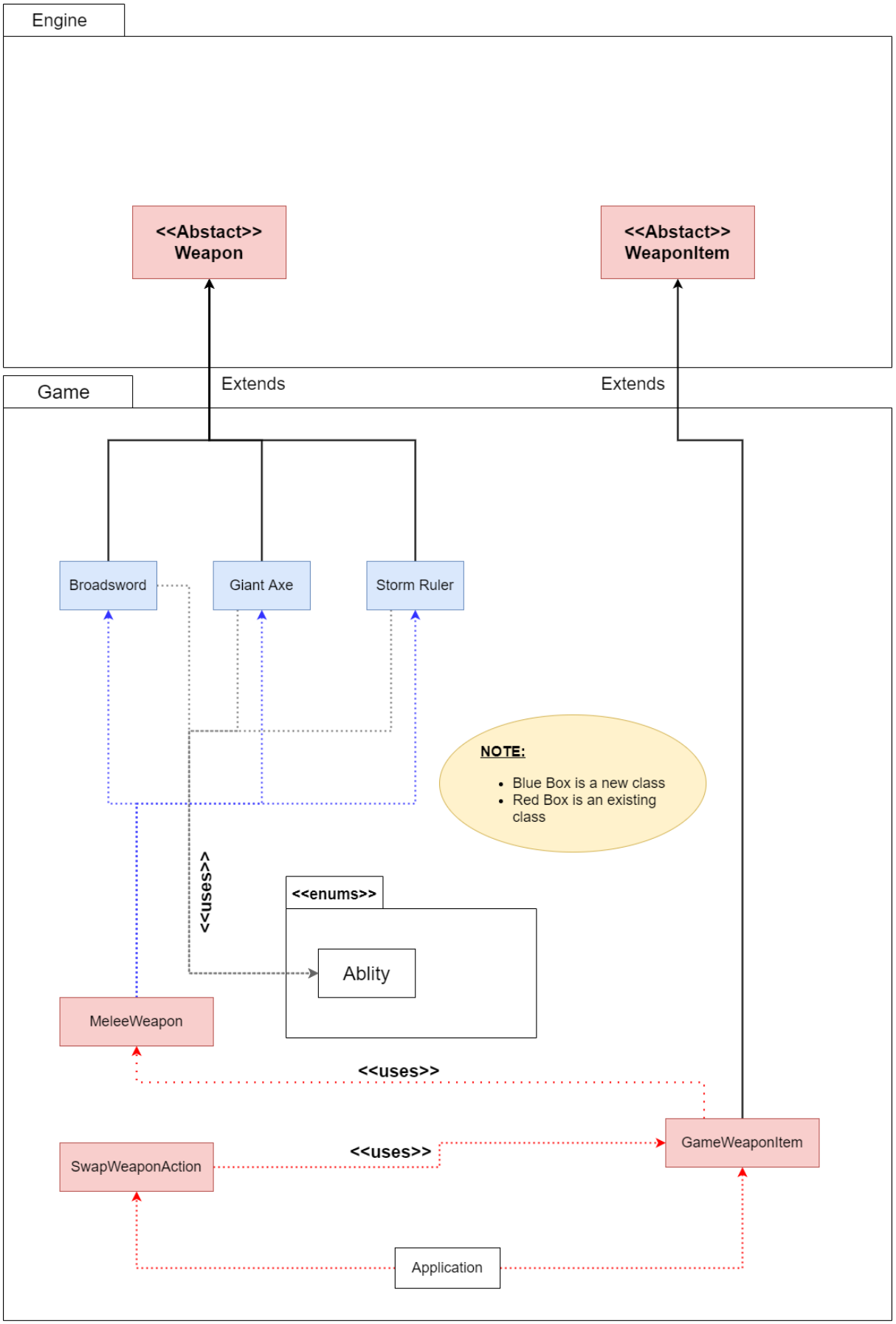



## Requirement 6

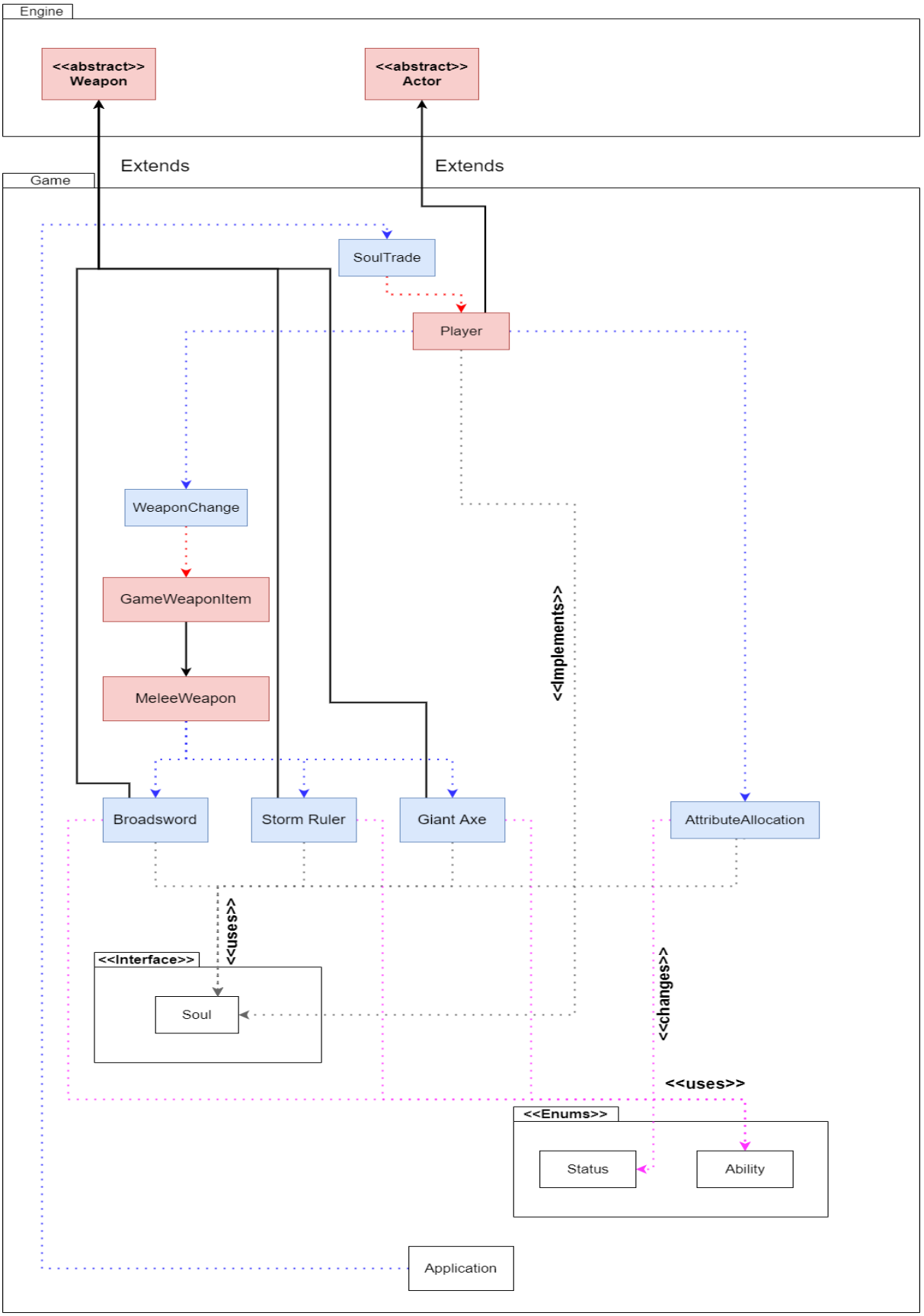
Souls and resetting


Engine
<<abstract>>
Actor
<<abstract>>
Item
Game
Extends
Extends
Interface
Player
Souls
Resettable
TokenOfSouls
Red coloured classes -> Existing classes
Blue coloured classes -> New classes
Application

# Requirement 7: weapons UML



# Requirement 8: Vendor UML



# Design Rationale

## Requirement 1

Part of the extension requirements on the legacy code provided are displaying the player and its properties on the i/o properly, thus, it is common sense that a modification is implemented on the Player class and its associating operating classes. It was mentioned to acquire the player's health / hit points (HP) in the console. Being said, a getter method is required to access the player's HP, subsequently, displaying it on the console (e.g. ***public void displayHP();***).

Furthermore, a player when 'spawned' into the game map should be **equipped with a 'Broadsword'**. Fortunately, in the one of the given legacy code and classes, there's an abstract class 'Actor' that has the method, '***public void addItemToInventory(){};***'. With that in hand, the mentioned method can be used after first initializing the Player into the game map to add a broadsword into the player's item list, thus equipping it. This is done in the *Application* class.

An Estus flask is also an item of the game and its sole purpose is to heal the player when used. Before getting into how to provide implementation of healing players, an estus flask can be treated as an extension to the Item class. So, not only the parent class (i.e. Item class) methods are inherited but new methods can be implemented in the child class (i.e. ***EstusFlaskItem***) and used. For example, actors who are holding onto the flasks can activate allowable actions such as healing the respective actor (i.e. The player). Not only that, the *EstusFlaskItem* class can be initialised with an **integer variable that keeps count of the amount of Estus flask** yet to be consumed by the player.

Speaking of action, a 'Player' can only execute an action if and only if it's allowed (i.e. Is the action in the allowable action list, also an *Actions* class type). Hence, an action that performs consuming an Estus flask should be created, namely, ***DrinkingAction***. The *DrinkingAction* class may have methods that override its parent class' methods (i.e. **public String Execute();**) ,and will implement it to use methods in *EstusFlaskItem* (Therefore, the UML diagram displays a generalization relationship between *Action* and *DrinkingAction* class.). Moreover, *DrinkingAction* class may also use methods from the *EstusFlaskItem* class to heal the player. Other worthy mentions to be implemented with the *EstusFlaskItem* such as initialising a variable of its class type to the *Player* class so that the number of charges would be tangible with the Player throughout the run time.

## Requirement 2

One important element that has not been implemented to the legacy code is a **Bonfire**, or a **Firelink Shrine**. It was said that in order to **reset the game softly and reset positions, health, behaviours and amounts of enemies**, as well as the **player's health**, the mentioned can all be done just by interacting with the Shrine on the game map, represented by the character *'B'*.

It has been said that the Bonfire may be interacted with if the user gives the right corresponding hotkey when his or her player is located directly on the Bonfire or adjacent to it. Therefore, by analysis of its properties, where it stays in one place in the duration of the game running, as well as that it may be stepped on, shares a lot of commonality of what a ***Ground* class** would possess. Being said, a *Bonfire* class can be created which is an extension to the abstract parent class, *Ground*.

A reset on the game may happen due to a few reasons (e.g. Player dies from his or her environment or the player decides to rest at the firelink shrine). Thus, having an interface class of *Resettable* may be a good idea to allow for different reset implementations, so that the game may act (i.e. Reset) accordingly based on the event of the game that has happened (e.g. At some events, items on the ground should not be removed.). So, an implementing class of ***RestAndReset*** can be created for the event that the Player decides to rest. The content of this class will also be created such a way that resettable instances (e.g. All the actors involved) are added into the resettableList in *ResetManger*, followed by implementation of message sending to these instances to reset (i.e. **resetInstance(){};**) these actors properly (which is precisely why the actors of the game are shown to be dependent on the *RestAndReset* class).

However, an approach where the user can trigger the rest action has not been implemented and ***RestAction*** precisely will be another class to provide execution of the *RestAndReset* class, as well as the implementation of condition checking (i.e. If the player is on top of the Bonfire or adjacent to it, which is why *Exit* and *Location* class are in a dependent relationship with the *RestAction* class.) It's also dependent to *RestAndReset* class so that the implementation can be used to heal the player, as well as resetting the environment with its many hostile enemies.

## Requirement 3

Another important element that has not been implemented to the legacy code is a vendor that does soul transactions with players, as well as transfering/dropping of souls when any of the actors are defeated.

Since every actor in the game is involved in combat, subsequently involved in transferring and dropping of souls, they all implement the interface class *Soul*. On the other hand, a vendor can not be attacked and can only stay in one place. Therefore, it is suitable to have a

*Vendor* class extending the *Ground* class because a *Ground* class does not have attributes such as HP and it stays in one place in the *GameMap* class.

As shown in the UML diagram, a *VendorTrade* class is created to implement the *Soul* interface class to implement a proper souls transaction methods in the *Soul* interface class. A 'Player' can only execute an action if and only if it's allowed (i.e. Is the action in the allowable action list, also an *Actions* class type). Hence, an action that performs trading with a vendor should be created, namely, ***TradeAction***. The *TradeAction* class may have methods that override its parent class' methods (i.e. public String Execute();) ,and will implement it to use methods in *VendorTrade* (Therefore, the UML diagram displays a dependent relationship between *TradeAction* and *VendorTrade* class.). Moreover, *TradeAction* class may also use methods from the *VendorTrade* class to adjust the soul balance appropriately.

## Requirement 4

For this requirement,we will need to implement enemies for the game.In order to fulfill this task,the design will implement these given enemies by extending it from the **Actor** abstract class (Undead and Skeleton both extends Actor) as the **Actor** class contains most of the necessary methods to allow the enemies to function.Each enemy will also be given behaviours that define their characters such as the Undead and Skeleton classes given both the **FollowBehaviour** and **WanderBehaviour**.In addition,each of the enemies given will have different abilities and status.To elaborate further, Undead would be given a Status where on each turn the Undead would have a 10% of dying when not following the player while on the other hand,Skeleton will have a 50% chance of resurrecting itself once killed (although it won't be revived more than once) which will be implemented through the ability class.

As for weapons certain enemies can use,these would be extensions under the WeaponItem class.The application would create the weapons given by the requirement which are the **BroadSword** and **GiantAxe** and then add them to the enemies inventory by using the **addItemToInventory** method.We do this instead of giving to the enemies from the start so that we would be able to reuse the weapons for other situations next time allowing reusability.Furthermore,the weapons will be able to create an **AttackAction** to initiate an attack where in this scenario, it would be the enemies weapon attacking the Player in the game.

Next, the requirement also asks for the implementation of a boss enemy "Lord Of Cinders".We can once again use the **Actor** class for this since most of the basics are

already there.The Boss enemy will have a designated weapon Machete which we will make a **Machete** class as an extension of **WeaponItem**.Moreover, the weapon will be given to the boss by again adding it to the boss inventory using the **addItemToInventory** method.The boss will also have an ability (Given through the Ability interface) which allows it to increases weapon hit rate.Once the boss is defeated,it will create a new item instance called CindersOfALord which is an extension of the Item abstract class that the player will be able to use in the future.

Moreover,the enemy classes will implement a Soul interface.This is so that we can transfer souls to the player when these enemies are killed by the player.Each enemy will be given a fixed amount of Souls depending on what type of enemy which will be transferred to another Souls instance (In this case the instance the Player has) when killed.

## Requirement 5

In order to implement the features given,we will need to make the Valley Class (which is already provided) and Cemetery class where both will extend the Ground class since most of the methods are suitable for this situation.Starting off with the Cemetery class,the cemetery represented as a C on the map will be given the task to spawn (create) the Undead based on a 25% probability where the class will signal the Application to place an Undead on at cemetery if successful.The Undead class similar in Requirement 4 will have the same set of behaviours and status with the only difference being how it is spawned (created).

Moving on,we will also need to implement a Valley class (which is already made but at the moment does nothing right now) that is represented with a + symbol on the map.The player will instantly die if touching the valley simulating that the player has fallen into the valley.This is done by **Action abstract** having the Valley class creating an Attack Action (an extension of the class) if in the same location as the player.The Attack Action will attack the player with damage equal to the player's health resulting in killing the player.This design is much easier than creating a whole new class focusing on killing the player when we can just reuse a class (In this case, **AttackAction** being useful for this) and add some methods tailored to the Valley class

## Requirement 6

An important feature that the game must have is soft reset/dying in the game.What needs to be done is for **RESET** features to be executed when the player dies as well as moved to the Bonfire location (Firelink Shrine).This is done by using the Resettable Interface where we would register all instances (**registerInstance**) to be softly reset such as Player's

health and soul including bringing the Player back to the last save location which is the Bonfire.In addition when the player dies,the player's soul count will be reduced to 0 using the Souls Interface provided since the methods are useful for this situation rather than making an entire new class for this.Not only that,an item class known as **TokenOfSouls** extended from the **Item** abstract class will be created when the player dies.

After the item is created,all of the Player's current soul will be transferred to the item using again the **Souls** Interface provided and the item will be placed on the location the player died,or in the situation where the Player falls into a Valley it will be behind the location instead.Once the player steps on the item location,the item will be picked up and all souls within the item will be transferred to the player as a result.In a scenario where the player dies before the item is collected,a new item will be made on the new death location while the old item will be removed from the game.This can be done by having a boolean check if a token already exists or not and removing the current token for a new token

## Requirement 7

One of the most important items in this game is that the player is able to use different types of weapons that are in the game. In order to equip the weapon, player need to use the SwapWeaponAction class to invoke GetWeapon(). However, there are a few restrictions on equipping weapon, therefore GameWeaponItem class which is an extension of WeaponAction Abstract is needed to state what are the restrictions of equipping weapon. Next, we implement the weapon damage, weapon attack speed into an existing MeleeWeapon class. This will ensure that every weapon has differences apart from the weapon skills.

Finally, we're moving to the Weapon Abstract extension which are Broadsword, Giant Axe, and Storm Ruler. All of the 3 extension is a new class which will uses the ability such as passive and/or active skills from the <<enums>> Ability class.

## Requirement 8

The game will have a trading system which was explained in the **Requirement 3** which will be using soul to trade with. The **Player** will have 2 option on what to trade souls with, one option is to change their weapon and the other option is to upgrade their stats which will be affecting their attribute points. So, 2 new class will be needed, **WeaponChange** class which will have the same **GetWeapon()** function as **SwapWeaponAction()** class and **AttributeAllocation** class which will **Change** player **<<enums>> Status** . Just like what we did on **Requirement 7** to change weapon the restrictions inside the **GameWeaponItem** class is needed, when the restrictions are not breached then the **Player** can get a new weapon which will affect their **<<enums>> Ability** because different weapons have different **Skills**. Both of the new Classes (**WeaponChange** and **AttributeAllocation**) will use the **<<Interface>> Soul** that the **Player** stores.

# Sequence Diagrams