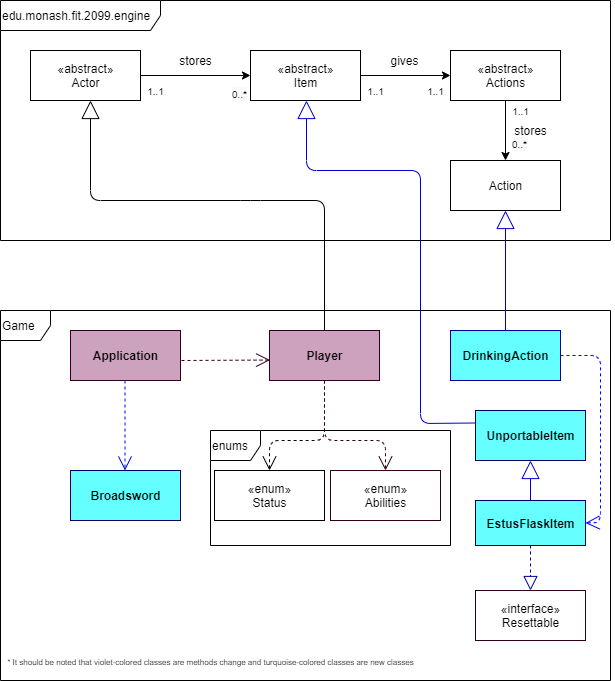
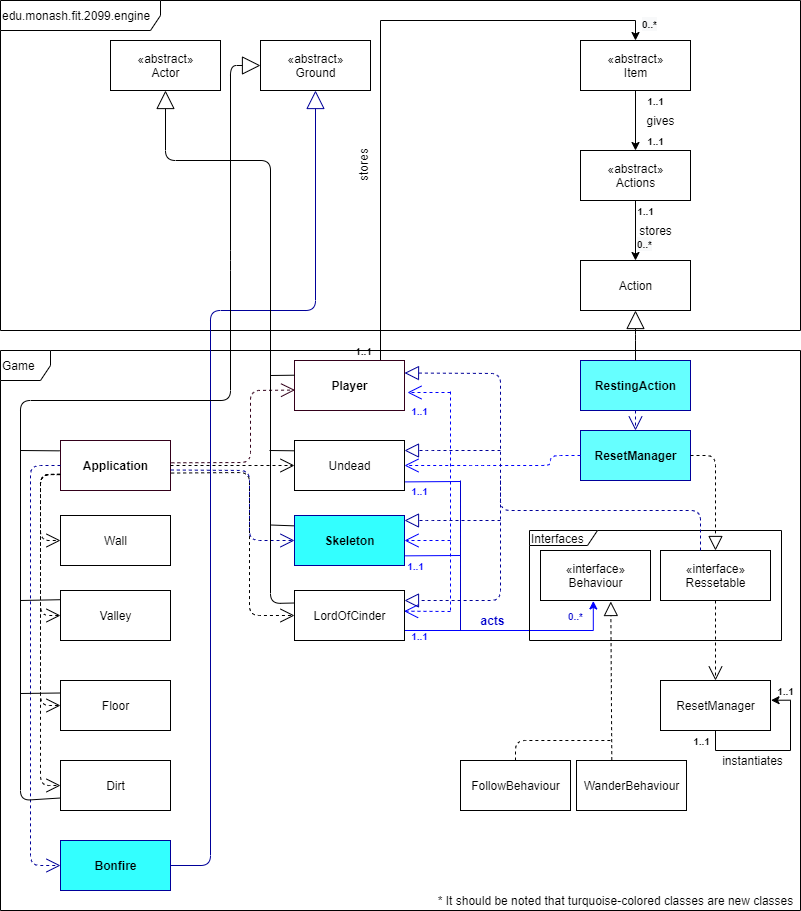
**Design Document**

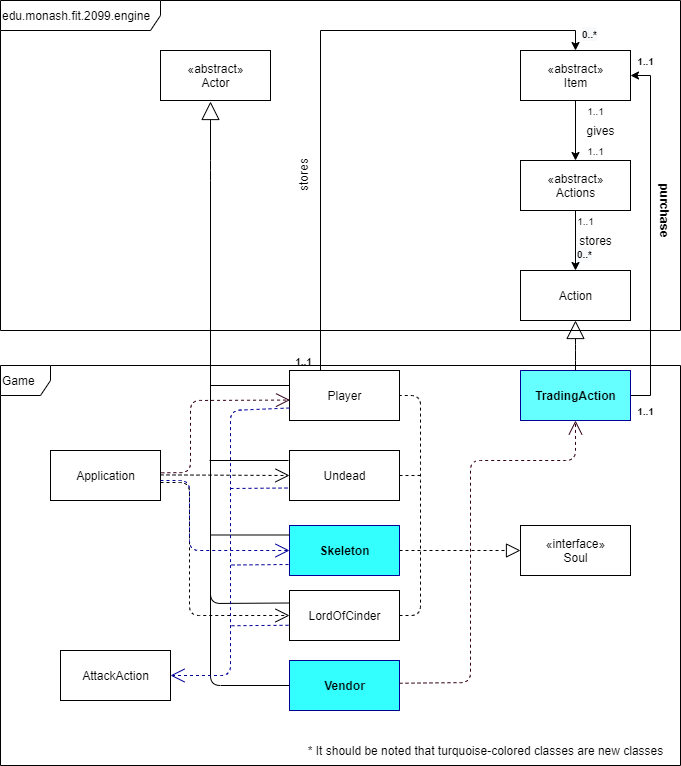
**Requirement 1**

****

**Requirement 2**

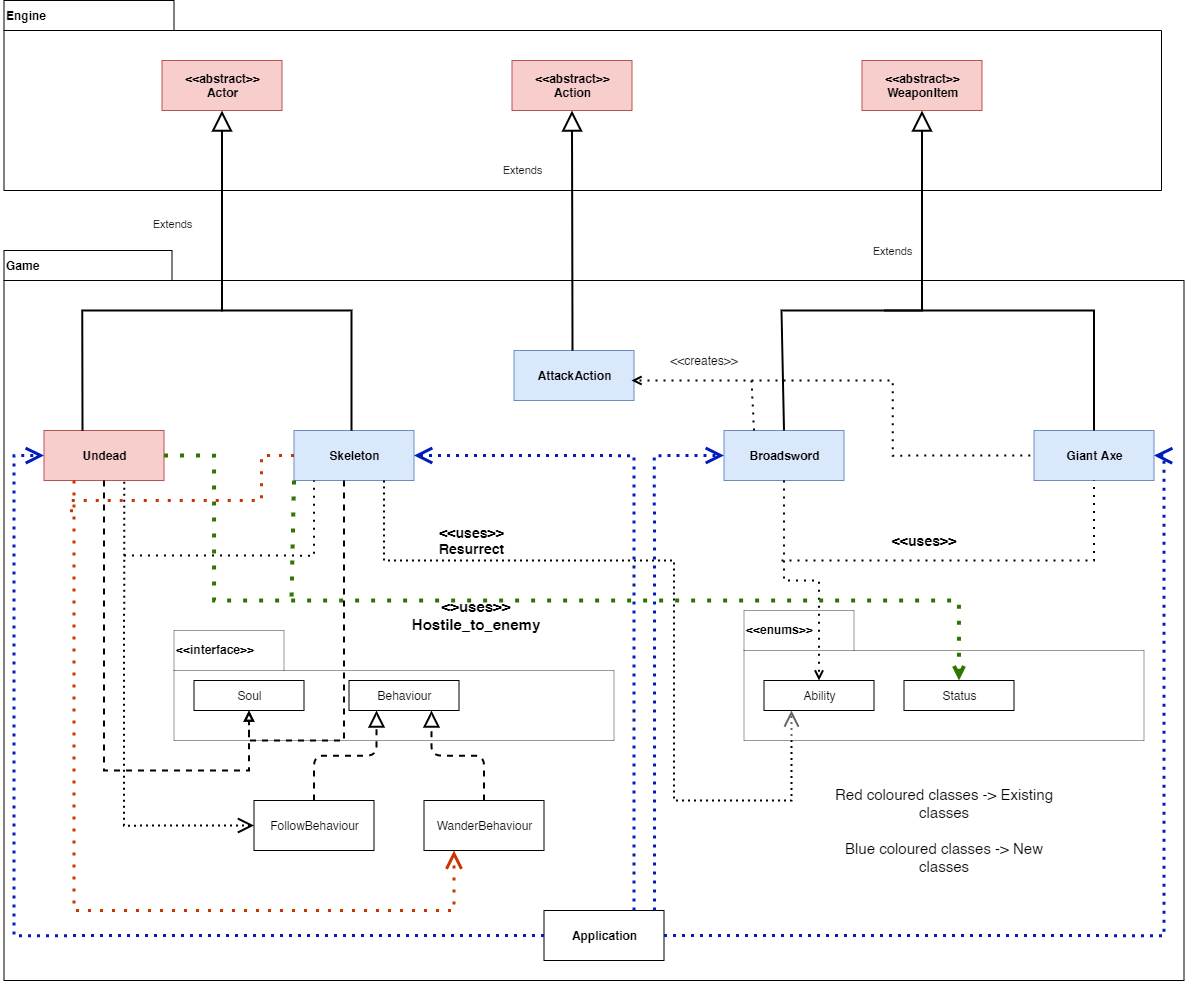
****

**Requirement 3**

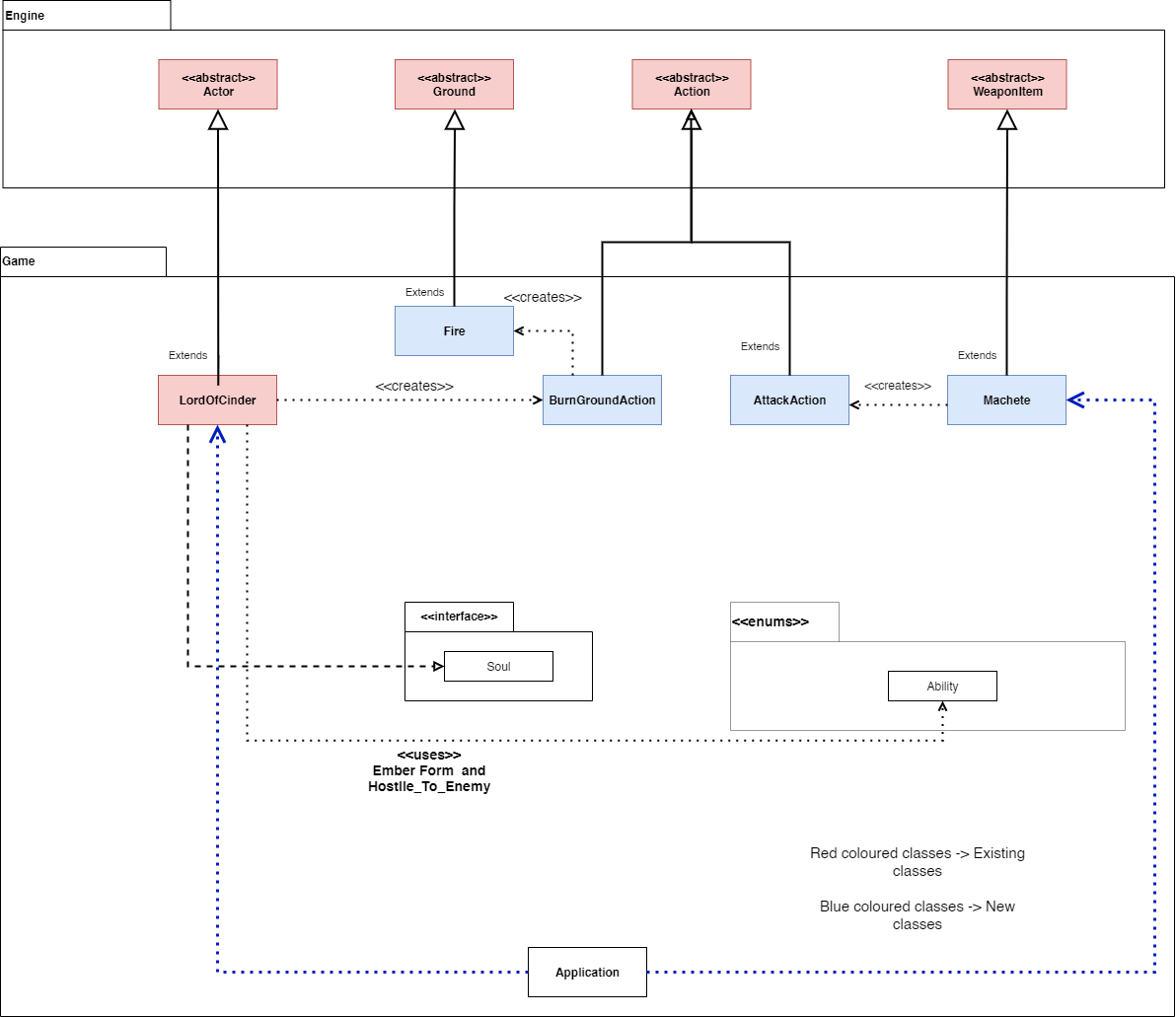
****

**Requirement 4**

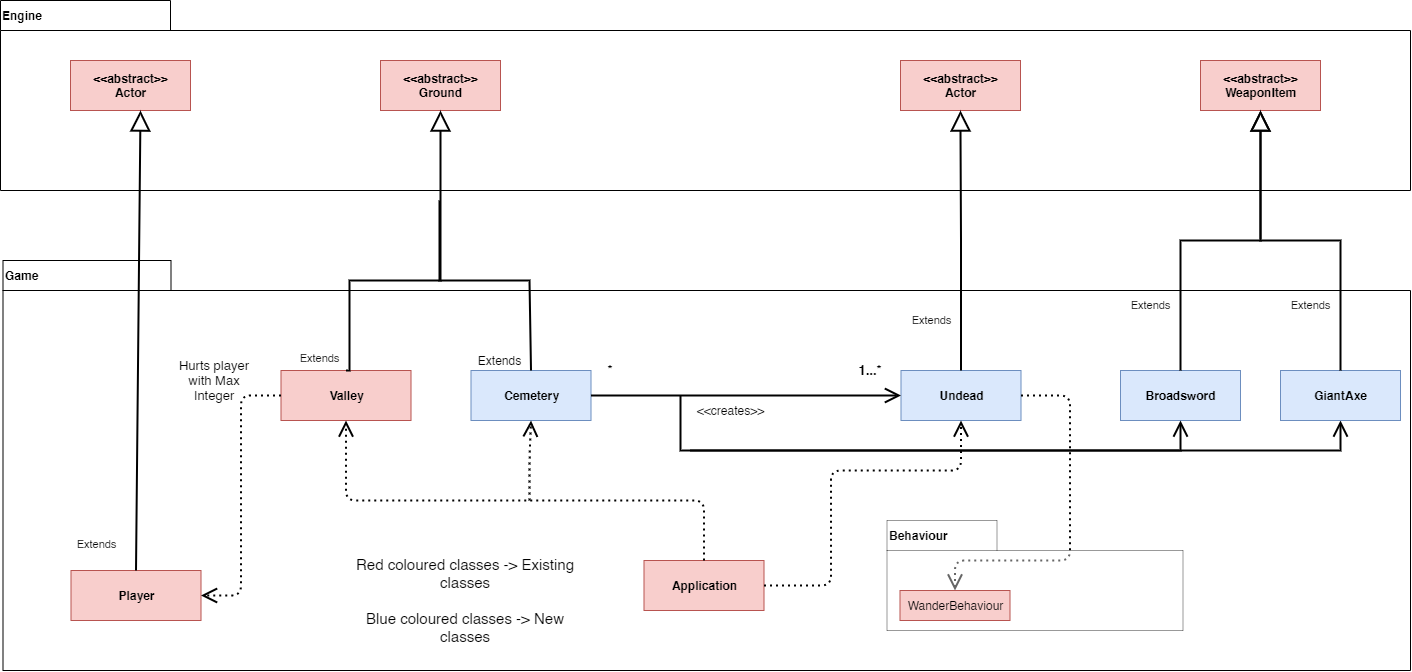
Skeleton and Undead



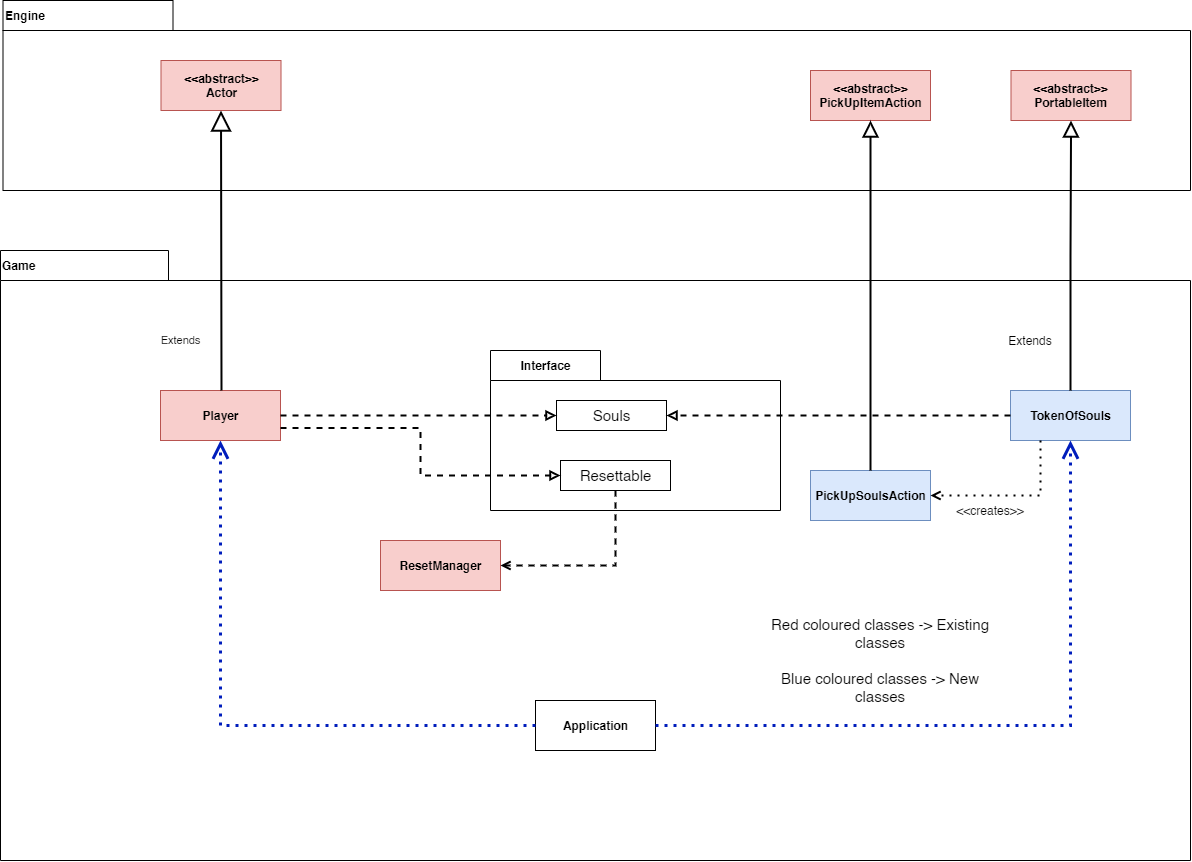
Lord Of Cinder



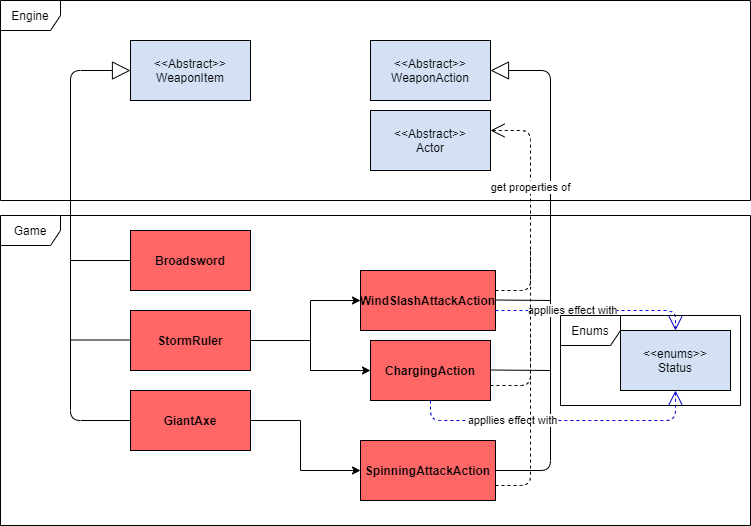
**Requirement 5**



**Requirement 6**

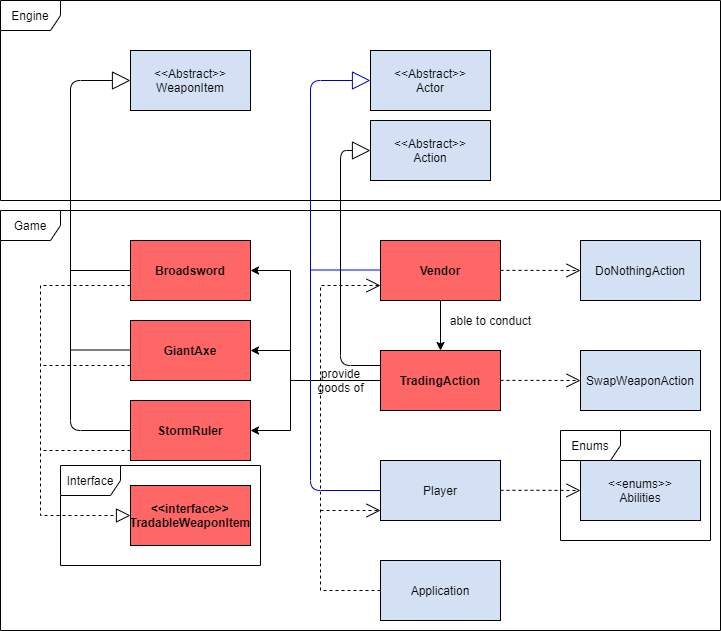


**Requirement 7**

****

\*Note: Red classes are new classes whereas blue classes are existing classes (i.e. Legacy codes).

**Requirement 8**



\*Note: Red classes are new classes whereas blue classes are existing classes (i.e. Legacy codes).

**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();***). Being implemented, encapsulation will be fulfilled on the player classes (i.e. Bundle the data with and its methods that retrieve it together into one single unit).

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. Therefore, we may reuse the existing code and not create unnecessary clutter of codes, making the overall representation of codes hard to understand. This is done in the *Application* class where it is dependent on a **Broadsword class** (which explains the dependency arrow pointing from the Application class to Broadsword 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 **UnportableItem** class (which is an extension to Item class that sets items’ portability to not true as a player can’t drop an EstusFlask). With UnportableItem being created, the Single Responsibility Principle is also enhanced as there’ll be no conflict between **PortableItem** and **UnportableItem** class.

So, not only the parent class (i.e. UnportableItem 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. Therefore and again, increase the reusability of the legacy code (i.e. Item class), which in turn gives more extensibility as the legacy codes are not interfered.

The EstusFlask item will also be set to private in the Player class as only the player may use the Estus flasks. It was decided to be designed that way to increase understandability of this class and the codes, but most importantly it would avoid other **Actor inherited classes** from accidentally using this Item class. Moreover, it will implement the **Resettable** class to refill all the charges back to maximum (i.e. **estusFlaskChargesLeft = 3**) so that it is closed for further modifications (i.e. Resetting the charges when the player dies or rests at a Bonfire) but open up for further extension, in other words, strictly following the **Open-closed principle**.

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.).

The methods in EstusFlaskItem includes **refillCharges()**, which is used by the Resettable class to refill back the charges back to full. Moreover, *DrinkingAction* class may also use methods from the *EstusFlaskItem* class to reduce one of its charges after healing the player.

Similarly, the DrinkingAction class will also be set to private in the Player class as only the player may use the Estus flasks, thus only the EstusFlask class may be initialised with the DrinkingAction class with self.addAction() method. Then, adding the action into the allowable actions list (i.e. self.allowableActions) if there’s any charges left.

**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,* reusing the legacy codes*.*

However, an approach where the user can trigger the rest action has not been implemented and ***RestingAction*** precisely will be another class to provide execution of the resetting all instances in the ***ResetManager*** class (i.e. **ResetManager.getInstance().run()**), in the ***self.execution()*** method. This class is also extended from the existing legacy code (i.e. ***Action* abstract class**) to implement all the abstract methods, which also explains the extension arrow pointing from ***RestingAction*** class to ***Action*** class. Thus, obeying the **open-closed principle** by closing all the possible modifications can be done for *Action* class.

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.). Being said, a status indicating he or she is resting will be added to the player’s capabilities when he or she rests at a Bonfire, that is done in the **self.execute() method** in ***RestingAction*** class (i.e. **actor.addCapability(Status.RESTING);**). After that, a resetting instance for the Player class (i.e. **self.resetInstance()**), which implements that particular function in ***Resettable interface class*** is implemented such that it would check if the actor has the mentioned status and reset accordingly.

**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*** (i.e. Empty arrow head pointing from all the enemy classes to the *Soul* interface class). Being implemented this way, potential coupling is loosened by considerably an amount compared to creating an implemented *Soul* class which is depended on by the actors. On the other hand, a vendor can not be attacked, or can he/she attack, and can only stay in one place. Therefore, it is suitable to have a *Vendor* class extending the *Actor* class while not being dependent on the *Action* class.

The ***AttackAction*** class is also implemented such a way that it checks if the Actor subclasses (e.g. Undead, Player etc) has died (i.e. **self.isConscious() = false**) and checks if the subclass is an enemy of an Actor. After that, the enemy’s souls are transferred to the one who defeated it (i.e. **target.asSoul().transferSouls(actor.asSoul());**), which in this case is always the player (i.e. *Player* class).

**Requirement 4**

After implementing the code for the enemies,it can be said that several classes were made in order to suit the requirements as well as satisfy most principles.For starters,the Skeletons are manually added to the game through the application class according to the requirements as well as manually creating the weapons and adding to the skeletons inventory.This can be done for the **Undead** as well.However,Requirement 5 wants us to spawn the Undead from the cemetery so that will be changed for that requirement.Upon making the enemies,they are given certain behaviours to a list,which we can loop through to check as well as remove certain behaviours if we want to.We do this instead of hardcoding it to make it more easier and flexible to change the behaviours we want our enemies to have.

As for the weapons,these will again be created in the application and given to the enemies.We also want them to create an **AttackAction** separately rather than make one within as that will require the weapon classes to know which actor to interact with and in the end **make unnecessary dependencies which we want to reduce** as much in our design.

For our boss enemy **Yhorm The Giant**,we now have it as an extension from the new abstract class **Lord of Cinder** which in the end allows us to make bosses under Lord Of Cinder and provides reusability of code.Firstly,we have our boss enemy given the ember ability only when it reaches less than half of its health.When it does,it will change increase its own weapon hit rate within the class.Not only that,it will return (or create in this case) a **BurnGroundAction** on its playturn and call that action.Within the action class,it will be given the location of the boss and on every possible route,create a fire ground for 3 turns.From the uml,the fire is a separate class extending from the ground class.This reduces the dependency between **BurnGroundAction** and **Fire** as well as allow us to reuse Fire class for other situations.Other than that,the boss and its weapon will be made again from the Application class and also implementing **Soul** to transfer its soul amount to the player when it dies.

**Requirement 5**

For requirement 5,we need to implement the **Cemetery** and the **Valley**.Starting with the Cemetery,it goes through a **tick()** function each time to let it know that a turn has passed.On every turn,it will go through the tick function and check if the made random int function returns 25% for where it will create an **Undead object**.The difference here is that instead of making the **Undead** from **Application**,we make it within the class by constructing from the Undead class.A random weapon is also given to it and then added to the game map on the cemetery location.In order to this,a method handling the undead making was made which also satisfies the **single responsibility principle**.

For the valley,not much is needed aside from checkallowableactions method where it would see if the player is on it’s location or not.Rather than making an extra action class on killing the player,we can reuse the actor method **hurt()** instead to give it as much damage showing that we are making use of the classes by reusing the already added methods.From there,the player on its turn will check if it still has health or not for it will call the resetManager (not part of this requirement) to soft reset the player.

**Requirement 6**

For this implementation,we would have the player class implement the Soul and Resettable interface.This allows us to be able to access the player’s soul since we can upcast it using **asSoul**.When a player is made,its instance is added to a ResetManager where the Manager adds the instance to its own list.We do this because,throughout the game we would want to soft-reset things in our game and rather than calling multiple classes to reset,we make a **ResetManager** to handle all of our classes and as a result simplifies things and the associations used.

Continuing on with the player,it overrides the resettable function on what to reset which are the player’s hitpoints and also create a new **Token of Souls** (extended from the PortableItem class) that will receive the Player’s soul amount and give it back through the PickUpSoulsAction (extended from PickUpAction) it creates that basically deals with transferring the souls and removing the token from the players inventory when picked up.The PickUpSoulsAction is also privated to encapsulate it and prevent any other access from other classes aside from just the Token class.Implementing Souls allows us to be able to transfer souls easily to whoever also implements Souls in their classes giving us **reusability**.

**Requirement 7**

First and foremost, *Broadsword*, *GiantAxe* and *StormRuler* are created as an extension of the Abstract class, *WeaponItem* (i.e. An empty arrow head pointing from those 3 classes to *WeaponItem* abstract class).Based on the appendix from the assignment previously, it was stated that each weapon in this game possesses an active skill as well as a passive skill. Being said, passive skills are invoked constantly and situationally by the game and throughout the game. Thus, a random number generator concept has been used in ***AttackAction*** to determine if the passive skills of each weapon should be executed (i.e. Twice the damage can be dealt with that particular weapon).

However, the active skills are executed manually and based on the intention of the player. Therefore, each active skills associated actions will be created (i.e. ***WindSlashAttackAction***, ***ChargingAction*** and ***SpinningAttackAction***). These actions are to be initialised in their respective ***WeaponItem***’s subclasses (e.g. *this.allowableActions.add(new SpinningAttackAction(this));)*. Needless to say, the active skills’ executable are all implemented in the *self.execute()* method, their respective classes. The Giant Axe’s active skills with be implemented such that it would check the actors’ surroundings (i.e. Instances of ***Exit*** classes) and hurt each of the enemies within its adjacent region (i.e. ***Actor.hurt()***). This is also the reason that this class is dependent on the *Actor* class (Dependency arrow pointing from *SpinningAttackAction* class to *Actor* class) to get the actor’s location, subsequently the *Exit* class instances. As for the active skills of a Storm Ruler, there are two of them, which are the *ChargingAction* and *WindSlashAttackAction*. The *StormRuler* class will be initialised with a private variable, namely ***chargedAmount***, which keeps track of the times the player has charged his/her Storm Ruler. It is also paired with an incrementer method (i.e. ***this.charge()***) as well as a method which reset the charges back to zero when unleashed a Wind Slash attack action (i.e. ***this.emptyCharge()***).

Coming back to the active skills associated with Action classes, the ***ChargingAction*** class would increment the *chargedAmount* with the *charge()* method mentioned previously and add a newly created status from the ***Status*** Enum class to the player wielding the Storm Ruler (i.e. *DISARMED*). To elaborate on this, this is so that in the *playTurn()* method in the player class can determine if player is able to conduct any attacking action (i.e. If the status is detected, the player is not allowed to do so). How it is implemented and designed is depicted in the diagram where a dependency arrow points from the *ChargingAction* class to *Status* Enum class. This class is dependent on the *Actor* class (Dependency arrow pointing from *ChargingAction* class to *Actor* class) to apply the mentioned status.

Similarly, *WindSlashAttackAction* class also depends on the *Status* Enum class to remove the *DISARMED* status from the player and add a new status (i.e. ***STUNNED***) to the Yhorm the Giant. This is due to the fact that this would allow the playTurn() method from LordOfCinder to determine if it can execute any actions or just *DoNothingAction()* when that status is detected. How it is implemented and designed is depicted in the diagram where a dependency arrow points from the *WindSlashAttackAction* class to *Status* Enum class. This class is dependent on the *Actor* class (Dependency arrow pointing from *ChargingAction* class to *Actor* class) to apply the *STUNNED* status and remove *DISARMED* status.

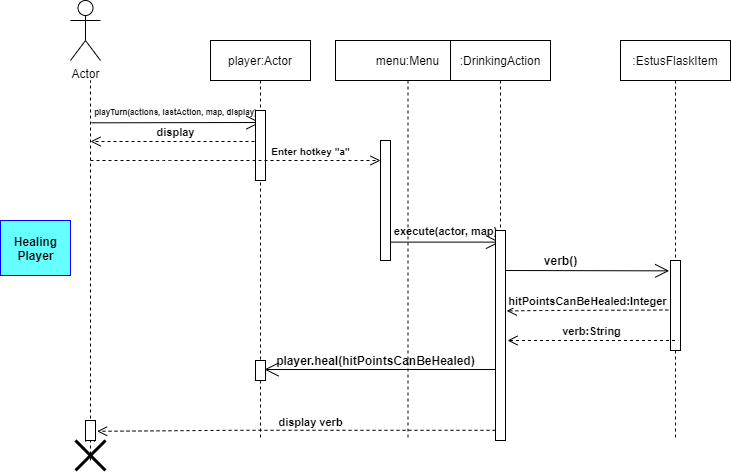
It is also important to note that since each actions are only initialised in their respective *WeaponItem* subclasses, the active skills associated action classes are all kept private in their respective *WeaponItem* subclasses to enhance encapsulation of the classes designed, also providing limited accessibility to classes other than their outer classes (Less prone to error).

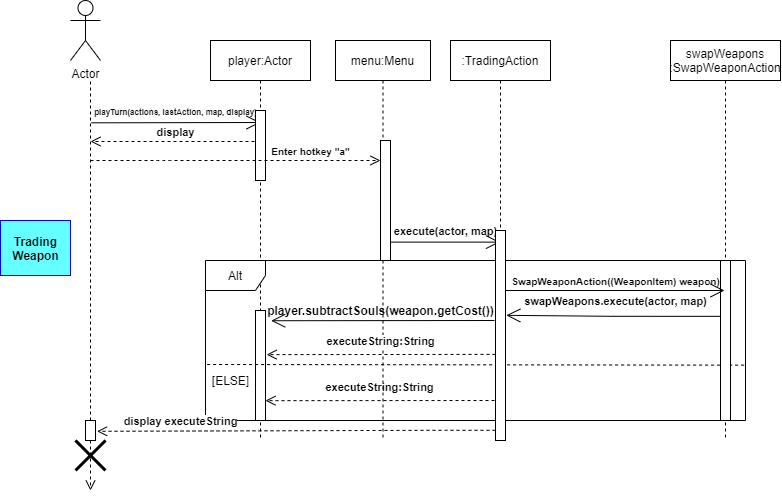
**Requirement 8**

Based on the *WeaponItem* extension classes mentioned in Requirement 7 (i.e. *Broadsword*, *GiantAxe* and *StormRuler*), it was stated that these classes are tradable by the Fire Keeper(i.e. *Vendor* class). Therefore, an interface to accommodate the tradable properties for some WeaponItems is created (i.e. ***TradableWeaponItem***). This design decision is based upon the **Liskov Substitution Principle** to **enhance subtyping of classes**, subsequently, making it simpler to place instances of subtype (which in this case is *TradableWeaponItem*) of some class in placeholders which is defined for their base class (which in this case is *Weapon*), and vice versa. The mentioned interface contains unimplemented methods that gets the name and cost of the tradable weapon item (i.e. *getName()* and *getCost()* respectively).The UML diagram also depicted a dashed empty arrow head pointing from the three *WeaponItem* subclasses to the *TradableWeaponItem* interface class, stating it’s an implementation of the interface mentioned.

An action to allow trading is also created (i.e. ***TradingAction***) which also will be used in the *Vendor* class and initialize it into its *this.allowableActions* (It also explains how the UML diagram depicted a dependency between these two classes, where a dependency arrow points from the *Vendor* class to the *TradingAction* class). Being said, the method which appears in every *Action* subclasses and would also appear in the *TradingAction* class would be implemented in a way that it would determine if anything can be done, by checking if the player has enough souls for the *TradableWeaponItem* instances’ cost (i.e. ***if (player.getSouls() >= (this.weapon.getCost()))***). Assuming that the player has enough souls, a ***SwapWeaponAction*** instance will be initialised with the data of the *Player* and *TradableWeaponItem* instance, that the player wishes to trade for. This is also depicted in the UML diagram where a dependency arrow is pointed from the *TradingAction* class to *SwapWeaponAction* class, showing how the event of trading takes place.

The *Player* class would also be added with an newly created ability, ***TRADING***, to trade (i.e.  *this.addCapability(Abilites.TRADING)*). This is so that when the Player is at the adjacent of a Vendor (i.e. The Fire Keeper), the Fire Keeper would give the allowable actions (i.e. *TradingAction*) if the adjacent entity has the mentioned ability. This is also depicted in the UML diagram where the *Player* class has a dependency arrow pointed to the *Abilities* Enum class .

**Sequence Diagrams**

****