FIT 2099 - Assignment 2

Design Rationale

26 Sep 2020

Lab1Team3

*\*\* Red fonts represent new changes made in Assignment 2 \*\**

*\*\* Coloured blocks in UMLs indicates a newly created class\*\**

# 

# Table of Content

[**Table of Content**](#_z717zo965m74) **1**

[**Requirement 1 : Player & Estus Flask**](#_yk8991avp3lp) **3**

[Display player’s health point](#_6ky4rgmopch7) 3

[Estus flask](#_5ffo14mll46u) 3

[Class diagram](#_5p1czji9cciv) 4

[**Requirement 2 : Bonfire**](#_qz8m0grinlfo) **5**

[Floor](#_phbx9tl9cuj8) 5

[Bonfire](#_yjdnlo836th1) 5

[Rest at Bonfire](#_xu8ow5hsnktl) 5

[**Requirement 3 : Souls**](#_21lt9m1oswc6) **6**

[Storing Souls](#_wth5yxuoevdz) 6

[Display player’s souls](#_n3vuyre7gfnr) 6

[Functions Related to Souls](#_mm7wctm3ngj4) 6

[**Requirement 4 : Enemies**](#_p7xle9m72vtd) **8**

[Undead](#_h9z7e3gchb87) 8

[Skeleton](#_bichyuggjit1) 9

[Lord of Cinder](#_pakym3qpfyn9) 10

[Enemies](#_pmjechf4wzx7) 11

[**Requirement 5 : Terrains**](#_fsyh4ykr9adn) **12**

[Valley](#_v78mk9vfd1gs) 12

[Cemetery](#_d982cx6grfad) 12

[Fire](#_6q8l9pdj3pf5) 14

[Displaying hazard deaths](#_xaj7l42dv6gz) 14

[UML](#_er6fhj3zp6i6) 16

[**Requirement 6 : Soft reset/Dying in the game**](#_bapn9qytx85k) **17**

[Reset features](#_hcefwd27una9) 17

[Die vs Rest](#_4ua9v37673vc) 18

[Optional Feature](#_9yrbnkkeiqt) 19

[Token of Souls](#_rtuj5yhvt5bz) 19

[Class Diagrams](#_jj4bjpyrw9w1) 21

[**Requirement 7 : Weapon**](#_cty3w2cx6lbk) **22**

[Display Equipped Weapon](#_lu0ymams7bjd) 22

[Types of Weapons](#_whuld78530j) 22

[BroadSword](#_cuyr2i785etn) 22

[Giant Axe](#_jf951n4i63) 22

[Storm Ruler](#_w7citnt213lm) 23

[Yhorm’s Great Machete](#_477o5ky58j22) 23

[Skills](#_wsha5mpuxj86) 23

[Passive Skills](#_ckwx817jkj1d) 23

[Active Skills](#_9t2vfnpdbhbw) 24

[Spin Attack Action](#_ekdljt7wx9gi) 24

[Charge Action & Wind Slash Action](#_1dtf9375cm47) 24

[Burn Ground Action](#_o2cbd533n98o) 25

[**Requirement 8 : Vendor**](#_svesaa2krov7) **26**

[Display in the map](#_mc555rimauxe) 26

[Trading](#_l0xiznbnxd8l) 26

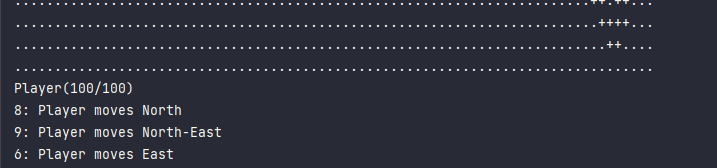
[Class Diagram](#_814w22xisyxt) 27

# Requirement 1 : Player & Estus Flask

## Display player’s health point

The status of Player’s HP can be displayed by calling the *display.println* function in the Player’s *playTurn* method.

Display:



## Estus flask

**Estus flask** should be implemented as a **subclass** of **item** class in order to acquire the behaviour of item class. This decision is made based on the observation that the player (Unkindled) must carry an Estus flask during the start of the game, which should be stored inside the player’s inventory. Additionally, the player is not allowed to drop the estus flask, indicating that the estus flask is not a “portable item”. Two private integer-type attributes, *maxCharge* and *charge*, should be implemented in the Estus flask class, representing the maximum charge and remaining charges available. Setting the attributes as private allows the class to be encapsulated; this prevents unauthorised parties from accessing the object’s data directly.

**Update 20th September 2021**

An abstract class named **Potion** which **extends** the **Item** class is created. This class represents a potion item which could be consumed by the player in the game. This allows the game to be more flexible in a sense where other types of potions can be easily added into the game in the future. **Estus Flask** will then **extend** the **Potion** class, allowing the **ConsumeAction**() to happen within the potion. These changes adhere to the **Liskov Substitution Principle**, as well as making good use of **polymorphism**.

**Assumption** : only Player objects can carry potion.

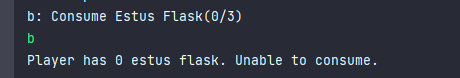
The player class should instantiate an estus flask while constructing a new player instance; the estus flask is then added into the player’s inventory by calling the *addItemToInventory* method.

The player can consume/use the estus flask through **ConsumeAction**, a **subclass** of the **Action** class. The override methods are to be customised to perform appropriate instruction (execute and menuDescription).

In summary, **Player---<<create>>---> EstusFlask** and **EstusFlask ---<<create>>---> ConsumeAction**.

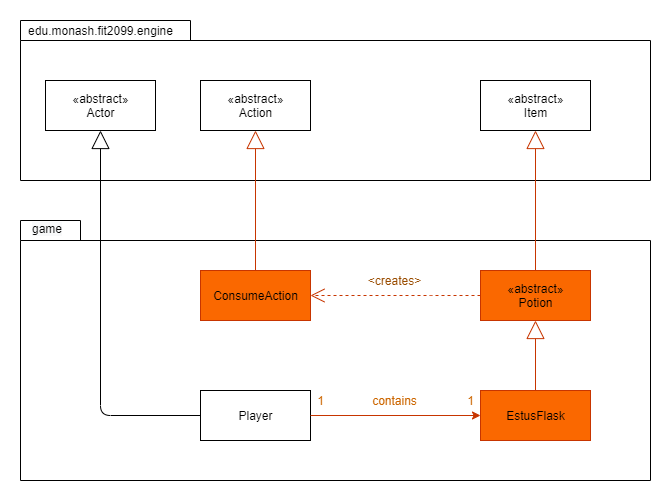
Estus flask’s charges are to be subtracted while executing the action. While the charge reaches 0, the program will display a sentence on the console upon selecting the “consume estus flask” selection, notifying the user that the operation has failed due to empty charges.

Sample:



## Class diagram

**Update 16th September 2021**

****

# Requirement 2 : Bonfire

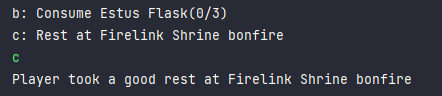
## Floor

Only player is allowed to pass through the floor, hence the *canActorEnter* method in Floor class should always return true whenever the actor is of player type.

## Bonfire

**Bonfire** should be implemented as a **subclass** of **Ground**, it inherits the behaviour of the Ground class. Whenever the player stands around it, the player is given the option to run “Rest at Firelink Shrine bonfire”. This is implemented by creating a **ResetAction** in Bonfire’s *allowableActions* method.

Sample:



## Rest at Bonfire

Upon selecting “Rest at Firelink Shrine bonfire”, the **ResetAction** is triggered. ResetAction **extends** the **Action** class, it inherits the behavior of Action class. While executing the ResetAction, the “run” method in the ResetManager object is executed. The details of ResetManager will be discussed later in [Requirement 6: Soft reset/Dying](#_fdeoktnarf0o).

In summary, **Bonfire ---<<create>>---> ResetAction.**

# Requirement 3 : Souls

## Storing Souls

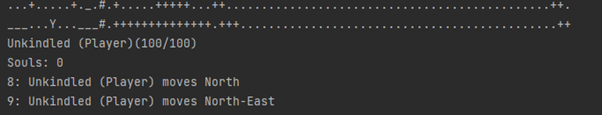
An attribute of integer type name soul will need to be included in the class Player and enemies, to store the number of souls that the actor is holding. Storing souls for each player and enemies will be needed as the value of the souls will be used when transferring the soul from enemies to the players, trading with the vendor etc.

There is also a class called TokenOfSouls, which will get all the souls of the player and drop it as an item in the map when the player dies. For more information, look at [TokenOfSouls](#_rtuj5yhvt5bz) under [Soft reset/Dying in the game](#_bapn9qytx85k).

## Display player’s souls

Since the souls are the currency of the game, the number of souls that the player holds could be displayed during each turn of the game. The display method can be implemented in player.playTurn so that it can display the amount of souls that the player holds each turn of the game, after the health points are displayed.

Sample display:



## Functions Related to Souls

There are three functions related to souls which will be used in different areas of the application.

TransferSouls, which transfers an actor’s number of souls to another actor, will be used when the player kills an enemy.

AddSouls, which increases the amount of souls of an actor. This method is used when overriding the TransferSouls method. During the TransferSoul method, it will increase the amount of souls of an actor by the amount of souls held by another actor.

SubtractSouls, which decreases the amount of souls of an actor. This method is used in the TradeAction, when the player trades with the vendor. Everytime the player performs a trade action, this method will be called to reduce the amount of souls of the player by the price of the trade.

# Requirement 4 : Enemies

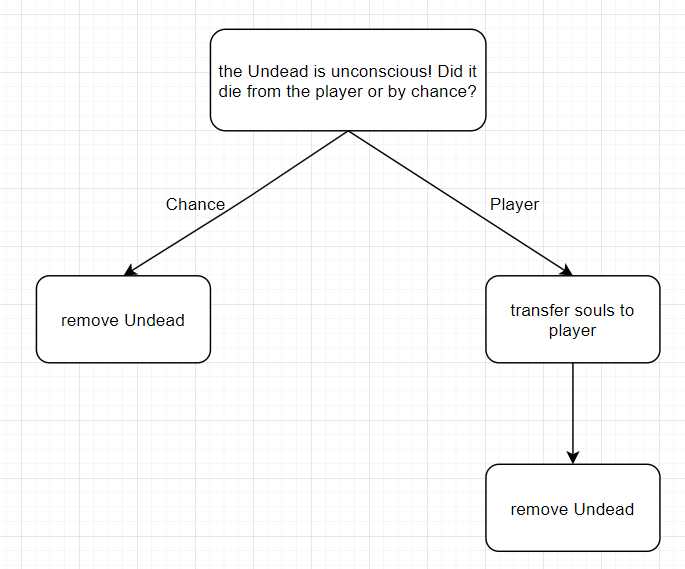
## Undead

The Undead is a basic enemy type that carries no weapons and spawns from the [cemetery](#_d982cx6grfad). It should have three behaviours, one which is the wandering behaviour which is their default state, and the other one is the following behaviour, and the last is the attacking behaviour, for when the Undead actually hits the player. Right now, the Undead only has the wandering behaviour up and running, while the following behaviour is already made but not implemented, and the attacking behaviour doesn’t exist yet.

Each actor has their available behaviours kept in an arraylist of behaviours in the playTurn method, which is an overridden method from the Actor class. For the Undead, the attacking behaviour should be on top of the arraylist, constantly detecting if the player is in range, when the player isn’t in range, it will go to the next behaviour in the arraylist, which will be either wandering or following. As the name suggests, when the Undead is wandering, it will just simply go to another tile at random, but when it detects the player, it will start following them, swapping out the wandering behaviour for the following behaviour.

The Undead also has a 10% chance to die instantly, meaning for every turn, the Undead would have to roll a 10 sided die and die if it guesses the correct number, this means the Undead has two ways to die, one from the player and other from the 10% chance. These two causes of death has to be kept seperate, since if the Undead dies from random chance, the souls contained in the Undead should not be given to the player.

The Actor class has the method playTurn, which manages the actor's turn, this could be made where every time playTurn is called, it will roll the die and when it dies from the die it will simply remove the actor. The player has an AttackAction class, where in the class, detects if the actor it's attacking is dead or not, this could be set as the Undead dying from the player, which will transfer their souls to the player then remove the actor.

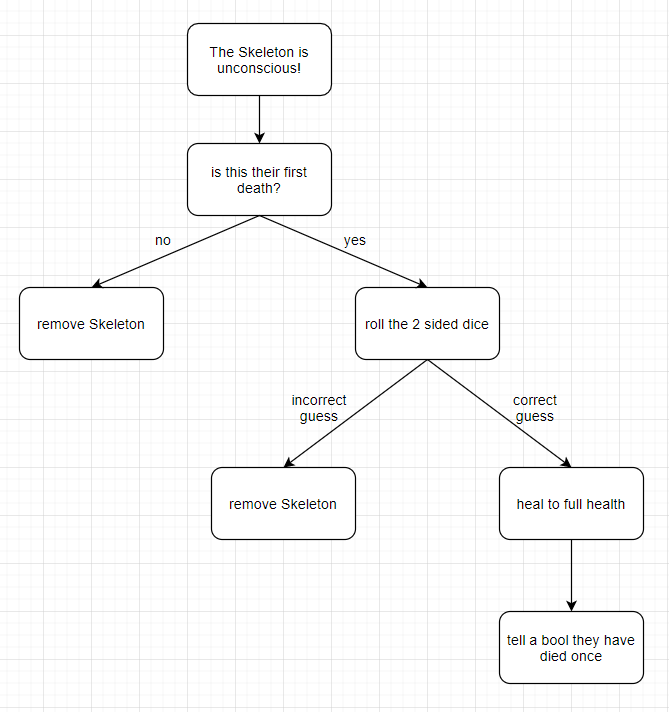


## Skeleton

The Skeletons is a more advanced version of the Undead, that should have the same behaviours but this time, it can hold a weapon and are also placed pre-determinately around the map instead of spawning out of the cemetery. Skeletons also don’t exist in the current version of the game, so a new class has to be made to add them in.

Even though the Skeleton has a weapon, it doesn’t mean they will have a new behaviour to make them use those weapons, [since special moves by the weapon are managed by the weapons themselves](#_cty3w2cx6lbk) instead of the actor’s behaviour.

The Skeleton too holds a dice, which will be rolled when its isConcious returns false, this dice will be 2 sided and if the correct number is guessed, it will be revived using the heal method. So when the Skeleton’s isConcious returns false, before removing the actor, it will first check if this their first death, if this isn’t their first death, then skip the dice roll and remove the actor, if it is their first death, then roll the dice, guessed the wrong number and it gets removed, guessed the correct number and it gets healed to full health and also tells a bool that it has died once.



**Update September 23rd 2021**

The skeleton will be revived through a **ReviveAction** so that the action isn't hard coded in the skeleton class and follows the Single Responsibility Principle. And the player's **AttackAction** will check if an enemy has the **Revive** ability, this is because actors that are attacked by the player and then become unconscious are removed immediately. So the check makes sure it gives the skeleton, or any enemy with the revive ability, to revive before being removed.

## Lord of Cinder

Lord of Cinder would have a unique set of behaviours, where for their movement, it would only consist of standing still and chasing the player. When fighting the player, if it’s HP goes down by 50% it would advance to the second phase, which would not change their behaviour but instead [activate an ability in their weapon](#_cty3w2cx6lbk). This ability consists of burning all dirt tiles that are adjacent to the Lord of Cinder.

This fire would be a new ground type that is added by the setGround method in the Location class and also only lasts for 3 turns. When the player steps on this ground for a turn, it will inflict damage onto the player. This damage can be given to the player through the Fire class itself.

## Enemies

As discussed previously, all enemies actually have a very similar structure, it has 3 behaviours, wandering (or do nothing for Lord of Cinder), following, and attacking, and wandering and following gets switched out depending if the player is detected. This means, to avoid repetition, an enemies class can be made which these three actors can extend to which will give them their behaviours.

The only main difference between these 3 enemies would be how they do their playTurn, which is a simple problem to fix since we can do polymorphism on playTurn, where most of what is in that method will be kept the same, but we also add some parts to it for the specific enemy.

**Update 22nd September 2021**

An extra thing added to the enemies is making them implement the **souls** and **resettable** interface, since all enemies should implement those two. This will allow them to be able to send their souls to the player when defeated and resetted when the player dies or use the bonfire. All enemies have their own soul count and not all reset the same way, but managing the different resets can be done with the [**resetManager,** which is explained later](#_bapn9qytx85k). Different soul amounts are managed by simply having a souls variable in each of their own classes.

**Update 23rd September 2021**

The LordOfCinder class has now become an abstract class. In this assignment, our first Lord of Cinder, **Yhorm the Giant**, will have their own class that **extends** to the **Lord of Cinder** class. Inside the Yhorm the Giant class will include a overridden playTurn function, where it checks if their health it below 50 percent, when it is, it will return a new action called the **emberAction** which is an action that simply prints out a message that Yhorm the Giant has entered its second phase.

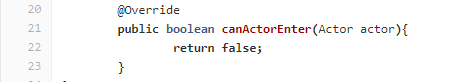
This implementation adheres to the **Dependency Inversion Principle**, where the abstraction layer allows us to limit the amount of effort to modify the parent class and child classes any further.

## 

# Requirement 5 : Terrains

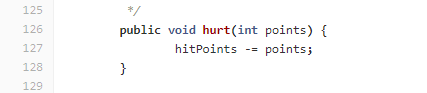
## Valley

Valley is a class that is extended from the Ground class, so it holds the method canActorEnter, this method simply accepts an actor as a parameter and returns either true or false, if it is false then the actor in the parameter cannot enter the specific ground.



The valley right now simply set this method to return false, so no actor can go into it. This should be changed so that it returns true, but not simply true for every actor, true only for players and when players step on it, give the player a ridiculous amount of damage for a guaranteed instant death.

Each enemy and player have their own class which is extended from the Actor class in the engine module and the Actor class has the method hurt, which from what the method name implies, hurts the actor for a certain amount of hit points, so the instant death will be sent to that method.



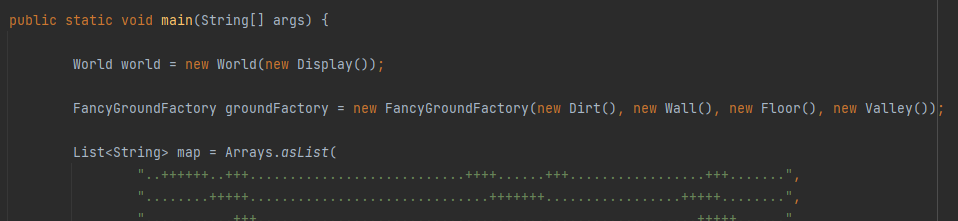
**Update 22nd September 2021**

Doing this approach of killing the player would not actually kill the player, unless the player has to constantly check their health every turn, which is what was changed. Now in the player’s playTurn, it will check if it's actually alive or not and reset the world if it is not.

This enhances **information hiding** in Player class, where the **attributes** of players are **never accessed outside of the class**. In addition, the program can always access that private/protected information through a method (if applicable).

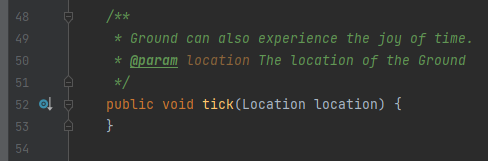
## Cemetery

The Cemetery right now does not exist in the current version of the game, so a class extended from the Ground class has to be made.



The application class makes a new instance of every ground type, and then places them in the arraylist below.

The World class in the engine runs on a clock called tick, when the Application class runs the world, each turn of the world will end by calling a tick class, which signifies a turn has ended. The World class is, to put it simply, quite inaccessible to the game module, so the next best thing is to use the tick in the Ground class, which is a class that Cemetery extends to so it can be used quite freely.

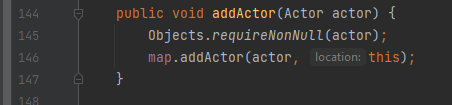


From here it looks like it doesn’t do much, but it can be Overridden.

*Sample of code from one of the demo games*



With this, tick can be Overridden to make it pick a number between 1 to 4, roll a 4 sided die, and spawn an Undead at its location if it rolled the correct number. The spawning could be done by using the Location object that was given in the parameter, since the Location class has the method addActor.



**Update 16th September 2021**

An extra check has to be made in the cemetery before it spawns an undead is to check if there is already an undead on the cemetery tile, this check was simply done by calling the **containsAnActor** method, if it returns true, it will not spawn an undead. Not doing this will throw an error.

## Fire

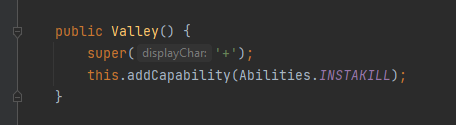
Fire will be a new ground type that is made by Lord of Cinder’s ability in their weapon. What fire does is pretty simple, it will just count 3 turns and turn itself back to dirt, and in those 3 turns if the player is standing on it, it will inflict 25 damage to the player.

It will count for 3 turns using the **tick** method and turn itself into dirt by using the **setGround** method. It hurts the player the same way valley does it, but instead of an instakill, it just does 25 damage.

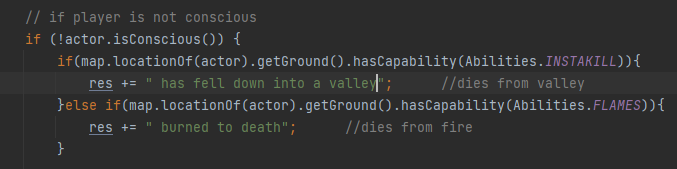
## Displaying hazard deaths

**Updated 22nd of September 2021**

Being killed by an enemy, burning to death, and falling to death are all different methods for the player to die, and different messages must be shown to indicate that. To do this, grounds that have hazards like valleys and fire will be given their own abilities.



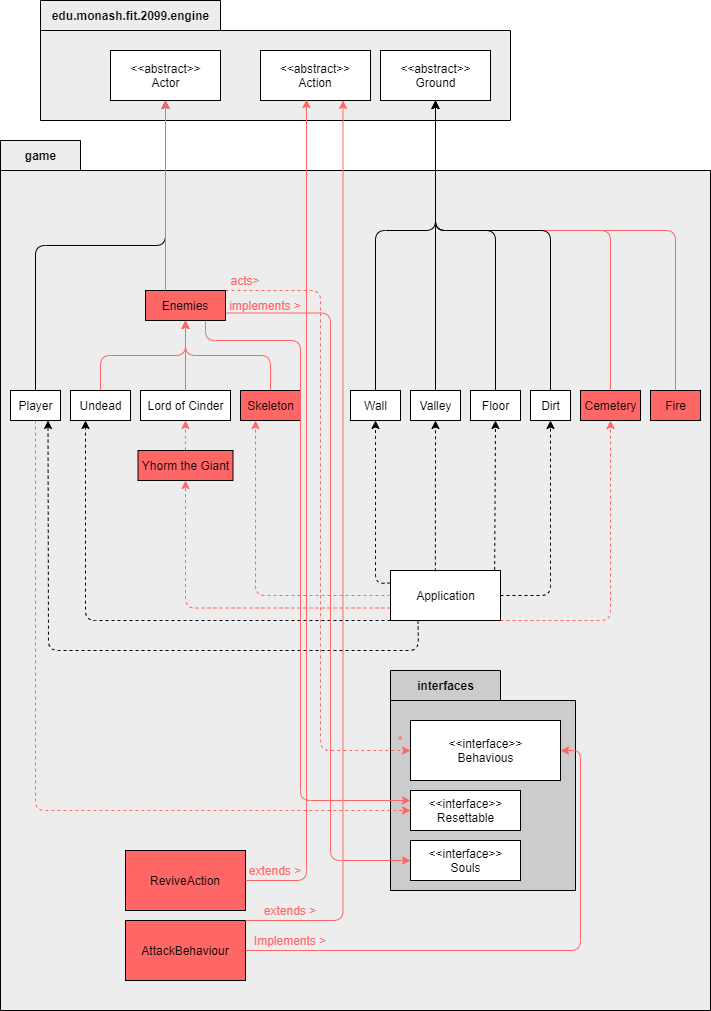
These new abilities are added to the **abilities enum** and when the player dies, their **resetAction** will check if the ground they died on has the special abilities, which then will return the appropriate messages.



This implementation adheres to the **Single Responsibility Principle**, where the ResetAction should handle the “death+reset” of the player. Particularly, the PlayTurn() method in player should not handle the “death+reset”, instead, it should render the ResetAction to handle this.

## UML

**Updated 26th September 2021**



***\*Note: Connections and classes in pink signifies a new connection and classes***

So by the end, the UML for both the 4th and 5th requirement would look like this. For the enemies requirements, there are now 3 new classes, **Skeleton**, **Enemies**, and **AttackBehaviour**. The skeleton class would function more or less the same as the other enemies, extending to the enemies class, which the enemies class now implements the behaviour interface, the souls interface, and the resettable interface, since all enemies have those and the enemies class also extends from the actor class. The attack behaviour class will now extend from the Action class and also implement the behaviour interface. There are also now the **fire** and **cemetery** classes. There is also the new **ReviveAction** class for enemies that can revive themselves, like the skeleton.

# Requirement 6 : Soft reset/Dying in the game

## Reset features

Whenever the player dies/gets killed (HP <= 0), the reset features will be executed through **ResetAction**. Referring to the ResetAction discussed earlier in [Requirement 2: Bonfire](#_qz8m0grinlfo), it extends the Action class and inherits all the behaviour of the parent class. The ResetAction is responsible for calling the *run* method of the **ResetManager** object while executing.

**Update 22nd September 2021**

The **run()** method in **ResetManager** will loop through the **ResettableList** and **trigger** their **resetInstance**() method accordingly.

This **reduces the dependencies** between **ResetAction** and **ResetManager**, as calling the run() method now would be enough to trigger all reset features.

The actors/items which are to be “reset” should **implement** the **Resettable** class in order to “reset” through running the *run*  method in ResetManager.

The actors can be registered as an instance in ResetManager through calling the *registerInstance* method in their constructor. By doing so, the instances will be added into the *resettableList* of the ResetManager instance, representing that they will be “reset” once the *run* method is executed.

The reset features of each actor/item are to be implemented in their respective class, using the method *resetInstance*.

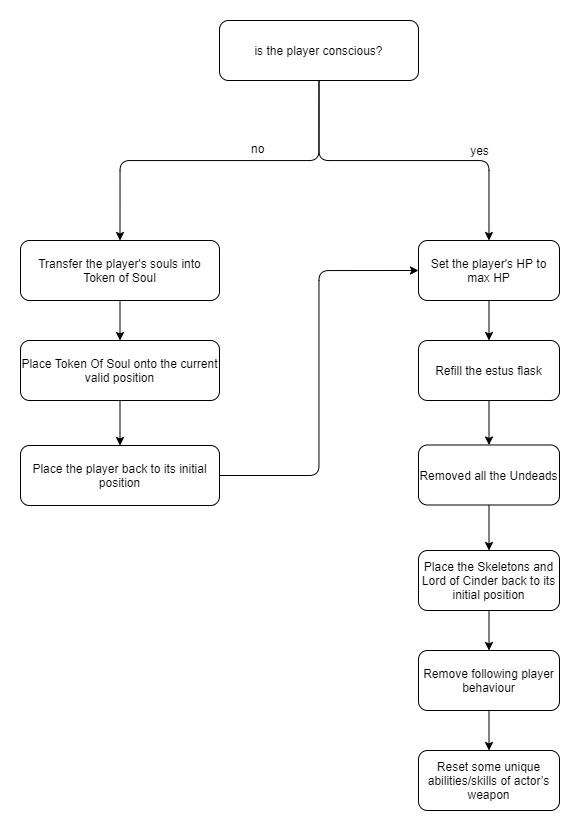
Eg: set player HP to max HP, reset player location to initial location, refill charges of estus flask

In summary, **ResetAction---<<create/call>>---> ResetManager** and **ResetManager---<<use>>---> Resettable**.

## 

## Die vs Rest

To differentiate between rest and die, the *run* method will check if the player is currently conscious. The activity diagram below shows how the method will differentiate between die and rest:



## 

## Optional Feature

**Update 22nd September 2021**

Our team decided to implement the optional features, which allows the enemy to experience the reset features.

Two new classes were created, which are **DieAction** and **RepositionAction**. Both of them **extend** the **Action** class.

We could send an actor to death by returning DieAction in its playturn. Notice that the “remove actor from map” action will then be done in DieAction instead of playTurn. This adheres to the **Single Responsibility Principle**, where the DieAction should only have **one responsibility**, which is to remove the actor from the game while returning a string. The same principle applies to RepositionAction, where the actor is repositioned onto its initial position.

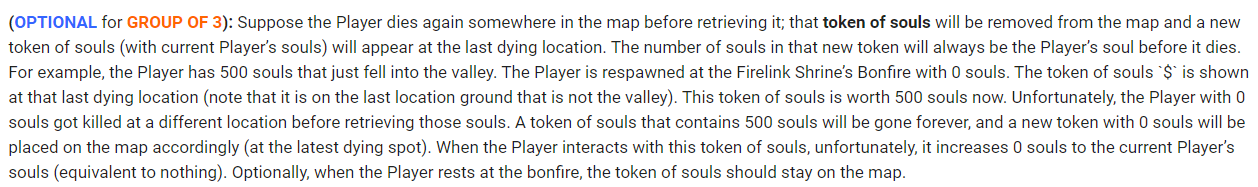
## Token of Souls

**Token of souls** will be implemented as a **portable item** named **TokenOfSouls** which **extends** the **Item** class. By doing so, the TokenOfSoul **inherits** all the properties of the Item class. According to the activity diagram shown above, if the player is unconscious (HP <= 0), a TokenOfSouls object will be instantiated and the player’s souls will be transferred into this object. The TokenOfSoul will then be dropped on the player’s dying location through the *addItem* method.

The TokenOfSouls will remain on the map until the player steps onto it. This will trigger the **PickUpTokenAction**, which **extends** the **PickUpItemAction** class. The *execute* and *menuDescription* methods are to be **overridden** as the TokenOfSouls are not required to be stored into the player’s inventory. This action allows the player to pick up the TokenOfSouls and regain the lost souls.

**Update 17th September 2021**

In this project, our team decided to implement the feature below.



A new **TokenManager** class is created to manage the token of souls in the game. Similar to the ResetManager class, TokenManager class is implemented as a **singleton** class which can only be instantiated in the ResetAction class. The purpose of creating this class is to manage the token of souls in the game, in particular, this class is responsible for adding a new token of souls onto the map while deleting the previous one.

This design **reduces the dependency between ResetAction class and TokenOfSouls**, as the ResetAction cannot directly access the TokenOfSouls class.

**What if the player dies again before regaining the Token of Souls?**

The TokenOfSouls will be implemented as a **singleton** class so that the program can only have an instance of the TokenOfSoul at a time. This is to ensure that the game map will not have two TokenOfSouls at the same time.

Given that the player dies before regaining the TokenOfSouls, the “reset features” will execute. As mentioned above, the TokenOfSouls will be instantiated while “resetting” the program. The “reset features” is expected to instantiate the TokenOfSouls through a **static factory method** (similar to the ResetManager class). If the TokenOfSouls already exist in the program, it will be removed through *removeItem* method. A new TokenOfSouls will be created and placed on the player’s last dying location.

**Update 17th September 2021**

The **TokenOfSouls** will no longer be implemented as a singleton class. The **getPickUpAction**() method in **TokenOfSouls** class will be **overridden** by returning a **PickUpTokenAction**, this action will allow the actor to retrieve the souls in the token instantly right after picking it up from the ground, without storing it into its inventory.

This design adheres to the **Dependency Inversion Principle**, where the **high level component** (in this case, it will be the PickUpItemAction) **should never depend on the low level** **component** (the PickUpTokenAction).

**Updated 22nd September 2021**

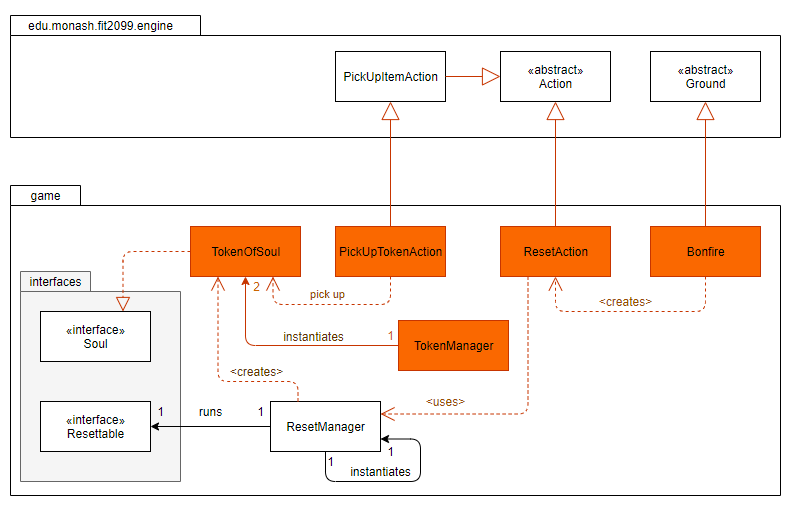
Our team has decided to implement the optional feature, which allows the token of soul to be dropped on the player's last location before falling into the valley. A new private attribute of Location type has been added into the player class to store the previous location of the actor, this allows the player to call the ResetAction along with a location parameter. This location now represents the dropped location of the token of souls.

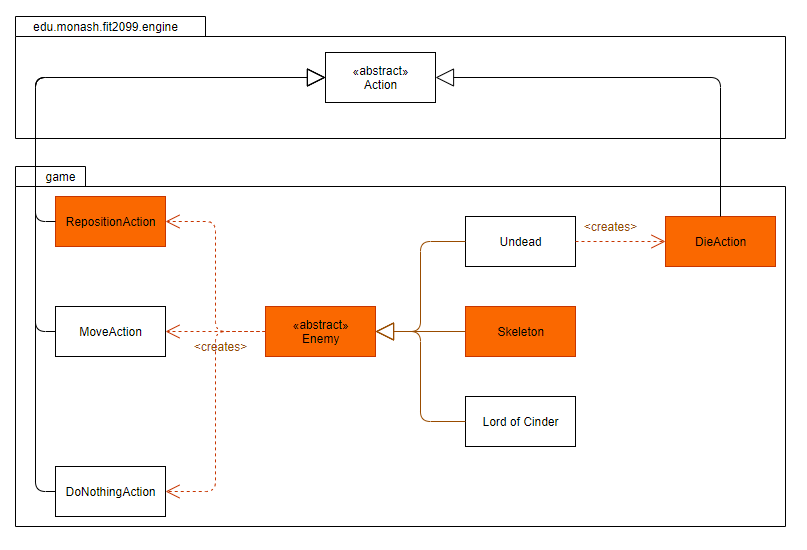
This design did not violate the **information hiding** and **encapsulation** of player class, as the location will not be accessible outside the class.

## 

## Class Diagrams

**Update 22nd September 2021**



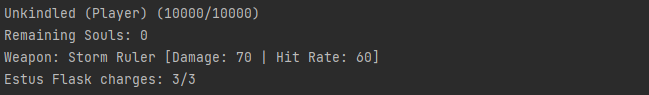


# Requirement 7 : Weapon

## Display Equipped Weapon

During each turn of the game, the name of the weapon equipped by the player, followed by the damage and hit rate will be displayed in the console. This feature is needed because it provides the player the information of the weapon so that the player can build strategies based on the weapon they use.

Sample display:



## Types of Weapons

**Updated 24th September 2021**

Different classes for each weapon will be implemented due to each weapon having their unique passive and active skills. This follows the practice of **Open/Closed Principle**, which allows each weapon to have their own methods in the future. A class called GameWeaponItem is also implemented to store the name of weapon, display character, damage, verb, hit rate and also its price for trading use. For more explanation on trading, look at [Trading](#_5eavt8b79iag) under [Vendor](#_h95en5c8qeyw).

### BroadSword

This class **extends GameWeaponItem** and it initializes all the required information on the weapon in the constructor by calling the parent constructor. Besides that, it also adds a capability to the weapon so that the [passive skill](#_ckwx817jkj1d), Critical Strike, will apply during the attack action of the holder.

### Giant Axe

This class also **extends GameWeaponItem** and it initializes all the required information on the weapon in the constructor by calling the parent constructor. This weapon has no [passive skill](#_ckwx817jkj1d) but it allows the actor to perform an [active skill](#_9t2vfnpdbhbw) called Spin Attack when the actor holds this weapon. Therefore, an allowable action, **SpinAttackAction**, is added to the weapon in the constructor.

### Storm Ruler

This class also **extends GameWeaponItem** and it initializes all the required information on the weapon in the constructor by calling the parent constructor. This class has an attribute called charge which stores the charge of the weapon that will be used when calling the [active skills](#_9t2vfnpdbhbw).

It also overrides getAllowableActions and also tick method to change the allowable actions of the user depending on the charges of the weapon.

### Yhorm’s Great Machete

This class also **extends GameWeaponItem** and it initializes all the required information on the weapon in the constructor by calling the parent constructor. This class only overrides the tick method to add capability called EMBER\_MODE to the weapon when the actor enters the Ember mode for the first time and removes it in the second turn so that the active skill, Burn Ground Action will not perform every turn when the actor is in ember mode.

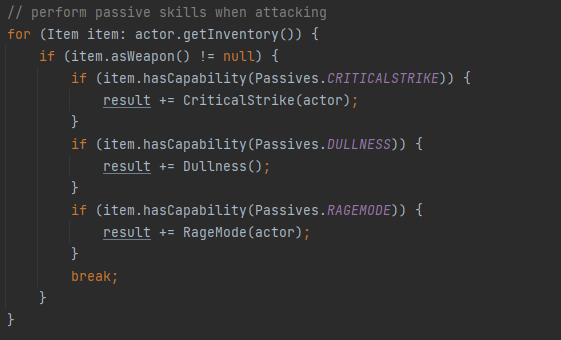
## Skills

**Updated 24th September 2021**

There are two types of skill for the weapons in the game, where passive skill will perform all the time but active skills allow the actor to perform it whenever they want.

### Passive Skills

There are three passive skills in the game, Critical Strike, Dullness and Ragemode. To perform the passive skills, we need to check for the type of weapons. To do this, we created enums so that we can add capabilities to the weapons. Then we check if the weapon has any capability of the passive skill, if it has the capability of the passive skills, the effects of the passive skills are performed.



### Active Skills

Active skills are actions that can be performed by the actors. Therefore, classes are implemented for each of the active skills. Active skills will be shown in the action menu during each playturn, depending on the weapon type and their requirements to perform the active skills.

Such design adheres to the **Single Responsibility Principle** and **Dependency Inversion Principle**, where each class should have one and only one responsibility and should not depend on low level components.

#### **Spin Attack Action**

This action can be performed anytime, even when there are no enemies around. When it is performed, it will check for all the actors around the location of the holder of the weapon by using the class Exit. If there are any enemies around the location of the holder of the weapon, it will deal half damage to all the enemies around it.

#### **Charge Action & Wind Slash Action**

The charge action can also be performed anytime. When it is performed, it will increase the number of charges of the storm ruler by 1. When the charge of the storm ruler is 3, the holder will not be able to perform this action, instead it will allow the player to perform wind slash action when the player is around a Lord of Cinder. The wind slash action deals double damage of the weapon to the target with the hit rate of 100%. The implementation includes enums to check if the weapon is charging or fully charged.

#### **Burn Ground Action**

This action will be performed only once when the weapon holder enters [Ember mode](#_pakym3qpfyn9). It is done in the [EmberAction](#_pmjechf4wzx7) class that was created to change the mode of the actor to Ember mode. Besides changing the actor to Ember mode, it also sets the surroundings to [Fire](#_6q8l9pdj3pf5) for 3 rounds, a new ground class that will deal 25 damage to the player when the player stands on it.

# 

# Requirement 8 : Vendor

## Display in the map

A new class named Vendor which extends the class, Actor, is implemented. In the Vendor class, an object of behaviour is created as the vendor has its own behaviour. The constructor should have a parameter of the vendor’s name. In the constructor, it calls the constructor of its superclass and its souls is set to 0;

To display the character in the map, do the same as how the boss is created. Call the method, gameMap.at(position).addActor(new Vendor(name));.



## 

## Trading

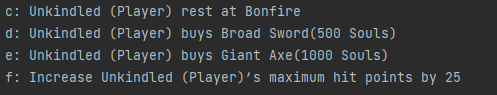
**Updated 24th September 2021**

To allow the player to interact with the vendor, a class named TradeAction is implemented, which has 2 constructor, 1 accepts GameWeaponItem as parameter and the other 1 accepts Actor as parameter. These two constructors are created so that the vendor can either sell the weapons to the player or upgrade the player stats.

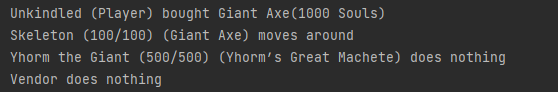
Besides that, an enum in Abilities, called TRADABLE is added as a capability to the player so that only player can trade with the vendor. When the player trades with the vendor, it will subtract the player’s soul based on the price of the trade by calling subtractSouls method in DesignOSoulsAddOn. If the player trades for a weapon, it will swap the new weapon with the old weapon by calling SwapWeaponAction.

In the Vendor class, an override method, getAllowableActions is created to add new TradeActions to the actions list so that it allows players to select their desired trades.

When the player interacts with the vendor, it displays all the available trades and their information in the console.



When the transaction is successful or fails, it also displays an appropriate message.



## Class Diagram

**Updated 25th September 2021**

