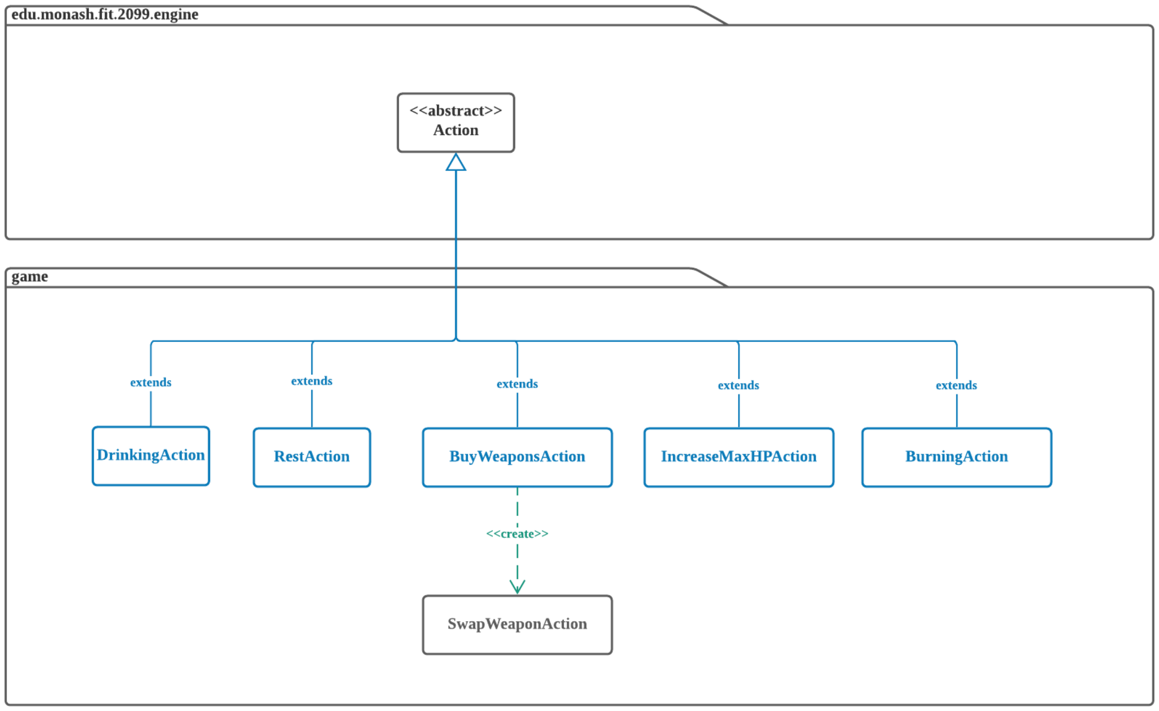
MA_Lab2Team4
Assignment1 UML Design Rationale

Name & Student ID :

1. Leong Wei Xuan (31435416)
2. Low Chin Boon (31993389)
3. Rebecca Nga (30720591)

Class diagram on Action (Requirement 1, Requirement 2, Requirement 7, Requirement 8):

UML Diagram on Action



Action class is an abstract class. It represents the things that the player can do. In other words, it represents the action that the user can invoke in the console. It has a menuDescription() method which can return a string describing a specific action in the console. The Player can then select the hotkey that represents the action in the console to invoke that specific action. It also has an execute() method which returns a string describing the condition after the action invoked is performed.

All the Action instances will be stored in the actions array list attribute in the Action class in the edu.monash.fit.2099.engine package (engine package). The execute() method of the Player selected action instance will be executed in the processActorTurn() method in the World class in the engine package when the application runs.

According to the diagram, I created the DrinkingAction class, RestAction class, BuyWeaponsAction class, IncreaseMaxHPAction class and BurningAction class in the game package. All these classes extend the Action class in the engine package.
By doing that, all these Action instances can be included in the actions array list attribute in the Action class in the engine package which will be displayed in the console. The inheritance

relationship allows all the child action classes to inherit all the attributes and methods from the Action parent abstract class. This prevents us from repeating similar codes in the application, which follows the Don't Repeat Yourself (DRY) principle. For example, the methods that are always inherited and implemented in the child classes are the execute() and menuDescription() method.

Although we will have created more classes in the game package, the overall application now is easier to maintain, extend and test.

1) **DrinkingAction class**

In our game, the Player (Unkindled) can choose to drink the Estus Flask which is a health potion to heal themselves. We are required to display the number of charges of Estus Flask in the console whenever the Player wants to drink it. In terms of the implementation, we can first override the inherited menuDescription() in DrinkingAction class to display the selection for this action in the console. By doing that, the player can select this action to drink the Estus Flask when they want. We can also override the inherited execute() method to display the number of charges of Estus Flask left after the drinking action.

2) **RestAction class**

Next, in our game, the Player can choose to rest when they are in the Bonfire. In terms of the implementation, we can first override the inherited menuDescription() in RestAction class to display the selection for this action in the console. We can also override the inherited execute() method to return a message describing the condition after the rest action.

3) **BuyWeaponsAction and IncreaseMaxHP action**

Apart from that, in our game, the Player can choose to trade souls with the Vendor, also known as the FireKeeper to buy new weapons or increase his own maximum hit points. This means that the Player can choose either of these options in the console. In terms of the implementation, we can first override the inherited menuDescription() in BuyWeaponsAction and IncreaseMaxHP action class to display the selection for these actions in the console. We can also override the inherited execute() method in both classes respectively to return a message describing the condition after the execution of these actions.

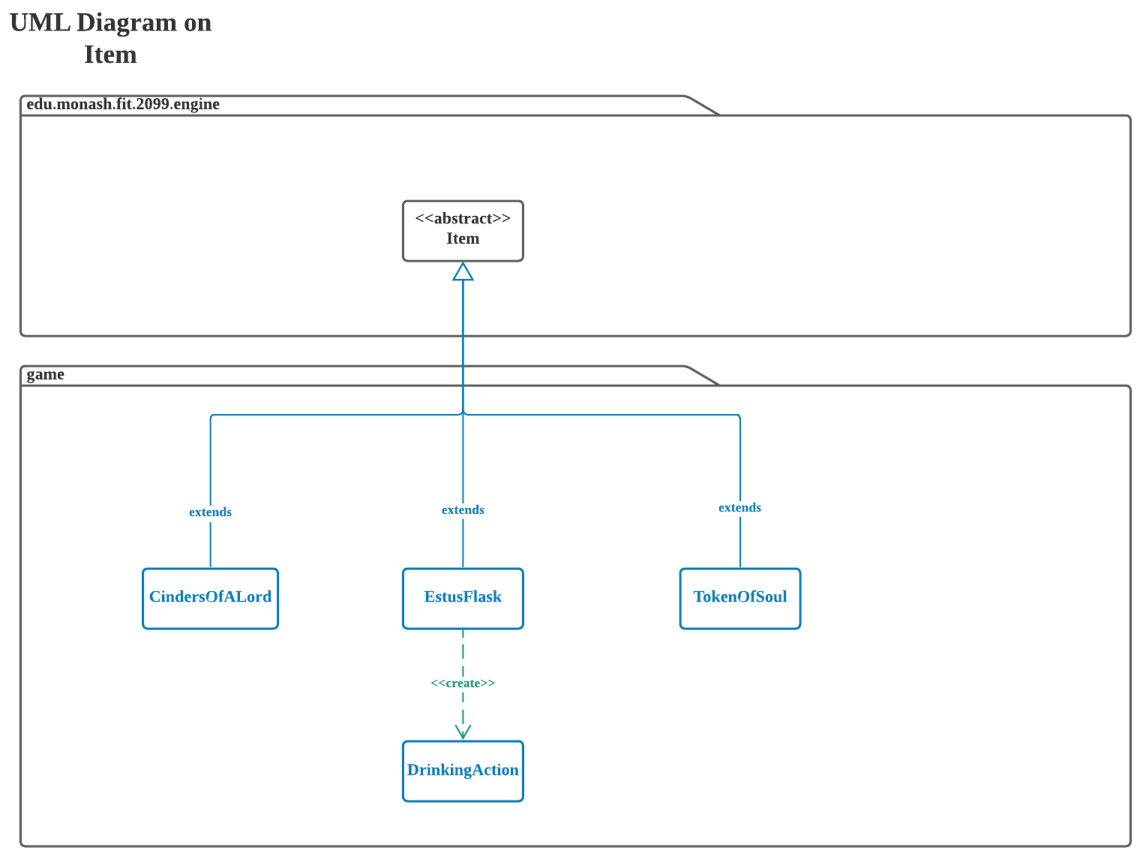
4) **BuyWeaponsAction class <<create>> SwapWeaponAction class**

We create a relationship between the BuyWeaponsAction class and the SwapWeaponAction class such that the BuyWeaponsAction class <<create>> SwapWeaponAction class. In our game, when the Player buys a new weapon from the Vendor, the weapon in the current inventory will be automatically replaced with it. Thus, inside the getAllowableAction() method in BuyWeaponsAction class, it will need to create a SwapWeaponAction instance and invoke the execute() method on the SwapWeaponAction instance to swap the old weapon with the new weapon. Thus, the BuyWeaponsAction class depends on the SwapWeaponAction class to replace the old weapon with a new weapon. Without the SwapWeaponAction class, the BuyWeaponAction class cannot fulfill the requirements completely. Thus, BuyWeaponsAction class <<create>> SwapWeaponAction.

5) <u>BurningAction</u>

In our game, when the Ember Form active skill executes, it will burn the surrounding/adjacent squares. Thus, we can override the inherited execute() method in the BurningAction class to hurt everyone that stands on the burning ground except the holder by 25 hit points and also returns a string describing the effect of the burning action. This fulfills the requirements of the burning action used in the BurningAction class.

<u>Class diagram on Item</u> <u>(Requirement 1, Requirement 2, Requirement 4, Requirement 8):</u>



In this UML diagram, there are four classes in the game package, such as CindersOfALord, EstusFlask, TokenOfSoul and DrinkingAction class. There is also an Item abstract class in the edu.monash.fit.2099.engine package (engine package). We let the EstusFlask, TokenOfSoul and Drinking Action classes extend the Item abstract class. The EstusFlask class in the game package also has a dependency relationship with the DrinkingAction class.

1) **<u>Why remove the PortableItem class in the game package</u>**

Firstly, we remove the existing PortableItem class in the game package which extends the Item abstract class in the engine package. The PortableItem class acts as a class for any item that can be picked up and dropped. In this case, if we create the PortableItem class, we need to create a NonPortableItem class to act as a class for any item that cannot be picked up and dropped.

Instead of creating new classes, we decided to utilize the boolean Portable attribute in the Item class which can differentiate the portable and non-portable item types. For example, if the Portable attribute of an Item instance is true, it means that the item can be picked up and dropped. Otherwise, if the Portable attribute of an Item instance is false, it means that the item cannot be picked up and dropped.

For example, in our game, Estus Flask is a unique health potion which the Player holds at the start of the game and the Player cannot drop it. Thus, we can set the portable attribute inherited in the EstusFlask class as false as it is not portable. Next, the Cinders Of a Lord is an item that can be dropped by the Lord of Cinder when it is killed and can be picked up by the Player. Thus, we can set the Portable attribute inherited in the EstusFlask class as true as it is portable.

2) <u>Inheritance relationship</u>

Next, we created the CindersOfALord class, EstusFlask class and TokenOfSoul class in the game package which extends the Item abstract class in the engine package. It is an inheritance relationship. By doing that, all these newly created classes can inherit the attributes and methods from the Item abstract class in order to fulfill their requirements.

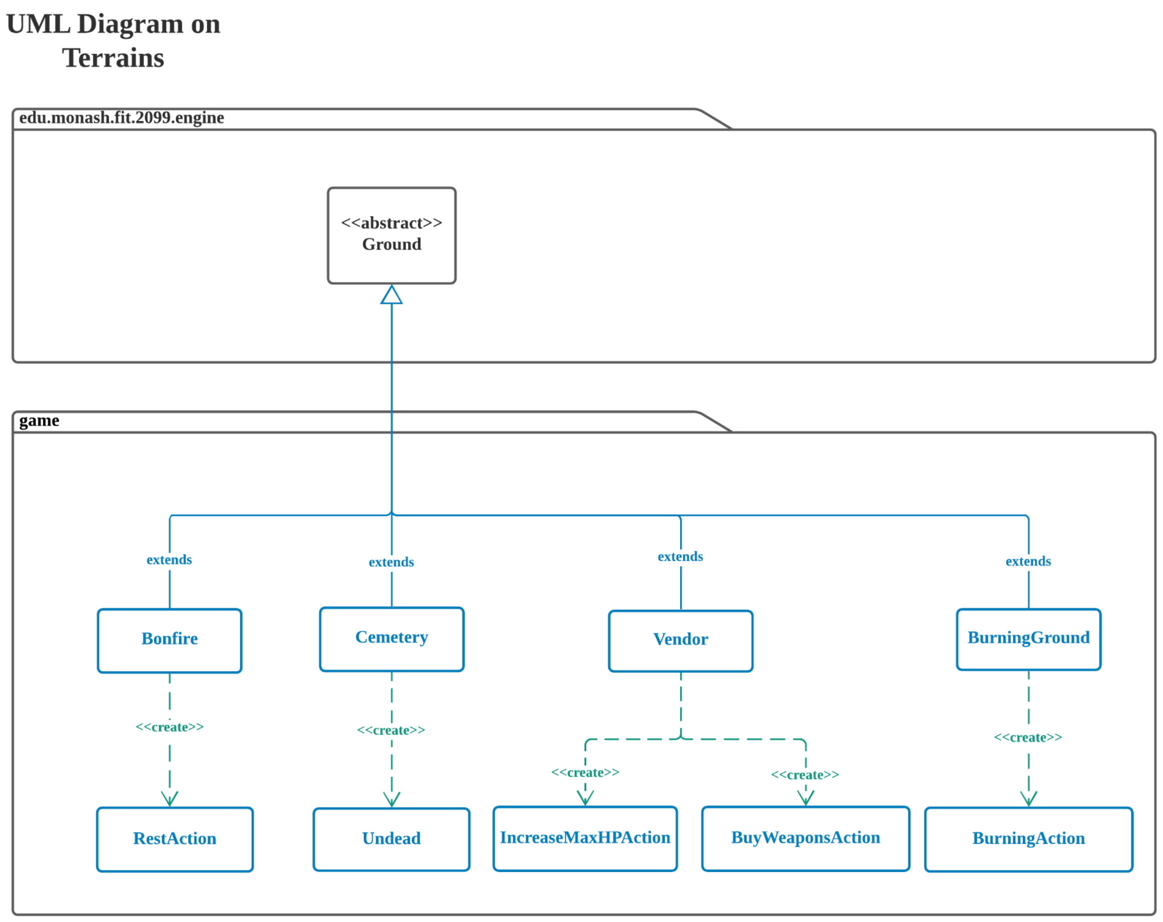
Alternatively, we can group all the items in the game into Portable and NonPortable classes which extend the Item abstract class. However, this is not a good design because each item has its own attribute and methods which differentiate them from others. For example, the EstusFlask has 3 charges, each charge can heal a Player with 40% of the maximum hit points. Thus, the EstusFlask class should have a charge attribute that is specific in its class and is not needed in other specific item classes.

3) <u>Dependency relationship between Estus Flask class and DrinkingAction class</u>

Next, We also create a dependency relationship between the EstusFlask class and the DrinkingAction class where EstusFlask <<create>> DrinkingAction. In our game, the Player (Unkindled) can choose to drink the Estus Flask which is a health potion to heal themselves. Thus, inside the getAllowableAction() method of the EstusFlask class, it will need to create an Actions instance and a DrinkingAction instance. Then, we can add the DrinkingAction instance as a parameter into the constructor of the Actions instance. By doing that, the DrinkingAction instance will be added into the actions array list attribute of the Actions instance that will be returned by the method.

When Player selects this action, the execute() method of this instance will then be invoked in the processActorTurn method in the World class in the engine package which will increase the hit points of the Player appropriately to fulfill the requirements. Without the DrinkingAction, EstusFlask cannot fulfill its requirements completely. Thus, EstusFlask class <<create>> DrinkingAction class.

<u>Class diagram on Terrains</u> <u>(Requirement 4, Requirement 5, Requirement 8)</u>:



In this UML diagram, we created Bonfire class, Cemetery class, Vendor class and BurningGround class in the game package. We let all these classes extend the Ground abstract class in the engine package. It is an inheritance relationship. This is because the attributes and methods in the Ground abstract class can be reused or overridden by these concrete child classes to meet their requirements. For example, the Ground abstract class contains an allowableActions() method which returns an empty Actions instance. As mentioned previously, the Actions instance has an actions array list attribute which stores the different action instances that can be performed by the Player on different grounds.

1) <u>Bonfire Class</u>

Firstly, in the game, the Bonfire is an area in the middle of the map where the Player starts. It is displayed by the character “#” on the map. By inheriting the attributes and methods from the Ground abstract class, we can set the displayChar attribute of the Bonfire class as “#”. The Bonfire has only one action known as the rest action and only players can interact with the Bonfire in order to rest. When the player rests, some of the reset features will be executed.

Thus, in the Bonfire class, we can then override the allowableActions() method that is inherited from the Ground abstract class by creating an Actions instance and the RestAction instance inside the method. Then, we add the RestAction instance as a parameter into the constructor of the Actions instance. By doing that, the RestAction instance will be added to the actions array list attribute of the returned Actions instance. By doing that, Bonfire ground can allow this action to be performed by the Player when he interacts with the Bonfire.

In this case, we are creating a RestAction instance inside the allowableActions() method in the Bonfire class. In other words, we are also creating a dependency relationship between the Bonfire class and RestAction class. Without the RestAction class, Bonfire cannot fulfill its requirements completely. Thus, Bonfire class <<create>> DrinkingAction class.

2) **Cemetery class**

In the game, the cemetery is displayed as character "C" on the map. By inheriting the attributes and methods from the Ground abstract class, we can set the displayChar attribute of the Cemetery class to "C". Next, each cemetery has a 25% success rate to spawn the Undead. Thus, we create a dependency relationship between the Cemetery class and the Undead class. In terms of the implementation, we can override the inherited tick(Location location) method in Cemetery class which keeps track of each turn of the game.

In this method, we can set an if-else statement to validate if the success rate is larger than or equal to 25% to create an Undead instance. Then, we can invoke the addActor() method on the location instance to add the created Undead instance to replace that particular cemetery located in the map. Since we are creating the Undead instance inside the tick() method in the Cemetery class, it shows that the Cemetery class depends on the Undead class in order to fulfill its requirements. Without the Undead class, the Cemetery class cannot fulfill the requirements completely. Thus, Cemetery class <<create>> Undead class.

3) **Vendor Class**

Next, in the game, the Player can choose to trade souls with the Vendor, also known as the FireKeeper to buy new weapons or increase his own maximum hit points. The Vendor is displayed as character "V" on the map. By inheriting the attributes and methods from the Ground abstract class, we can set the displayChar attribute of the Vendor class to "V". Previously, we have created the IncreaseMaxHPAction class and the BuyWeaponsAction class which enable the player to select these actions in the console to buy new weapons or increase his own maximum hit points.

Thus, in the Vendor class, we can then override the allowableActions() method that is inherited from the Ground abstract class by creating an Actions instance, IncreaseMaxHPAction instance and the BuyWeaponsAction instance inside the method. Then, we can add the IncreaseMaxHPAction instance and the BuyWeaponAction instance as parameters into the constructor of the Actions instance. By doing that, the IncreaseMaxHPAction instance and the BuyWeaponsAction instance will be added into the actions array list attribute of the returned Actions instance. By doing that, Vendor can allow these actions to be performed by the Player when he interacts with the Vendor.

Since we are creating both the BuyWeaponsAction and IncreaseMaxHPAction instance in the allowableActions() method in the Vendor class, shows that the Vendor class needs and depends on both BuyWeaponsAction class and IncreaseMaxHPAction class in order to fulfill its requirements. Thus, we create a dependency relationship between Vendor class with both BuyWeaponAction class and IncreaseMaxHPAction class where Vendor class <<create>> BuyWeaponAction class and IncreaseMaxHPAction class.

Alternatively, we can also let the Vendor class extend the abstract Actor class because the abstract Actor class has a getAllowableActions() method which can also fulfill the requirements. However, we do not choose this approach because the Vendor instance does not need most of the attributes and methods in the Actor abstract class. For example, the Actor class has attributes such as the inventory list, maxHitPoints, hitPoints and others. It has methods such as addItemToInventory, removeItemFromInventory and others. However, the Vendor instance does not need all these attributes and methods. Thus, it is more appropriate if we extend the Vendor class to the Ground abstract class.

4) **BurningGround**

Next, in the game, when The Yhorm the Giant activates the Ember Form skill, it will burn the surrounding dirt grounds. This burning action is implemented in the BurningAction class which is explained in the UML Action diagram.

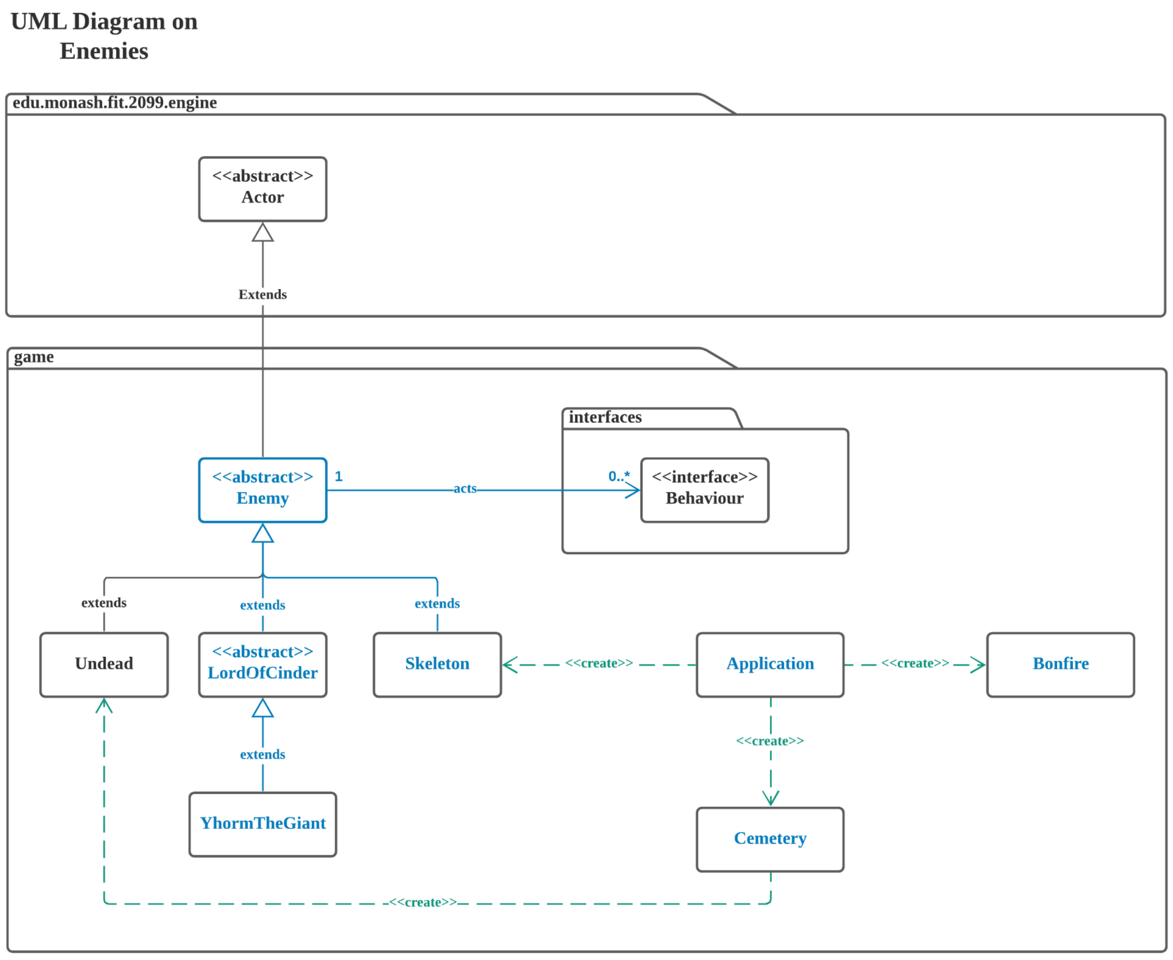
The burning area is displayed as the character “V” on the map. By inheriting the attributes and methods from the Ground abstract class, we can set the displayChar attribute of the BurningGround class to “V”. Next, The burning area “V” will stay on the map for the next 3 turns and will hurt anyone that stands on it by 25 hitpoints, except the holder that stands on it. Thus, in the BurningGround class, we can override the allowableActions() method that is inherited from the Ground abstract class to fulfill the requirements. Inside this method, we can first invoke the containsAnActor() method on the location instance parameter to check if there is an Actor on the ground. If yes, we can create and add a BurningAction instance into the actions list attribute of the returned Actions instance.

Since we are creating the BurningAction instance in the allowableActions() method in the BurningGround class, it shows that the BurningGround class needs and depends on the

BurningAction class in order to fulfill its requirements. Thus, we create a dependency relationship between the BurningGround Class and BurningAction class where BurningGround class <<create>> BurningAction.

Alternatively, instead of creating a BurningGround class to represent the burning area and return the BurningAction instance, we can also reuse one of the existing Ground child classes, such as Floor to fulfill this requirement. However, we do not choose this approach because the Floor may have more different functionalities than the BurningGround when we extend the project in the future. Next, this will be easier for us to add or update the functionalities of the burning area in the future. By creating a BurningGround class, we can differentiate different grounds so that each Ground child class will each have their own functionalities. Therefore, we believe that each Ground child class having an independent functionality would be a better design.

<u>Class diagram on Enemies</u> <u>(Requirement 4):</u>



1) **<u>Inheritance relationship between the Enemy class with Undead, LordOfCinder and Skeleton class</u>**

Initially, we have only the Undead and LordOfCinder classes created in the game package. These classes extend the Actor abstract class in the edu.monash.fit.2099 engine package (engine package). It is an inheritance relationship.

We decided to create an Enemy abstract class and a Skeleton class in the game package. The abstract Enemy class is created as there are various methods and attributes that are shared with each of the enemy classes. We extend the abstract Enemy class to the Actor abstract class in the engine package. We set the Enemy class as abstract instead of concrete class because it is a base class to create more specific enemy classes. The property of an abstract class is that it cannot be instantiated. In this case, we have created the specific enemy child classes and we will just instantiate those specific enemy instances. We do not want to instantiate the base Enemy class. Thus, we make it an abstract class.

Then, we extend the Undead, LordOfCinder and Skeleton class to the abstract Enemy class. Firstly, the enemies in the game all share some similar attributes and methods. For example, all enemies cannot enter the Bonfire and cannot attack each other, all enemies have the following behaviour and others. Thus, we can include all the common attributes and methods inside the abstract Enemy class. Then, all the specific enemy classes which extend to the Enemy class can inherit and override the attributes and methods from the Enemy abstract class. By doing that, we do not have to repeat similar codes in all the specific enemy classes. We have obeyed the Don't Repeat Yourself (DRY) design principle.

As mentioned previously, we create the Skeleton class that extends the Enemy class. This is because the Skeleton class can inherit all the common attributes and methods of an enemy from the Enemy class. By creating a Skeleton class, we can add some attributes and methods in this class that are only specific to Skeleton. For example, Skeleton starts from different maximum hit points than others. Furthermore, it can resurrect itself with a 50% success rate for the first death. Thus, we can create some new attributes such as isFirstDeath, or methods such as resurrect() method in the newly created Skeleton class to fulfill all its requirements.

2) <u>Changing the LordOfCinder from concrete class to abstract class</u>

Initially, the LordOfCinder class is a concrete class in the game package. We decided to change the LordOfCinder class from concrete to abstract class. This is because according to the expectations for the game expansion, we will have more unique Lords of Cinder(bosses) in the future. Since we will be creating the specific boss classes in the future, the LordOfCinder class is now a base class to create those specific boss child classes. We do not want to instantiate the base LordOfCinder class. Thus, we make it an abstract class.

By now, we have only one Lord of Cinder(boss) which is the Yhorm The Giant. Thus, we decide to create YhormTheGiant Class which extends the LordOfCinder abstract class. By doing that, YhormTheGiant can inherit all the attributes or methods that will be shared by all bosses while maintaining its unique attributes and methods which differentiate it from other bosses.

3) <u>Dependency relationship between the Application class with Skeleton, Cemetery and Bonfire</u>

In the Application class, we will create instances of the newly created Skeleton, Cemetery and Bonfire class in the main method of the class. By doing that, these instances will be added into the game map. Thus, we created the relationship between the Application with all these classes as Application <<create>> these classes to fulfill the requirements. Next, the dependency relationship between the Cemetery and Undead is explained previously in the UML diagram for terrains.

4) **Association between the Enemy Class to Behaviour Class**

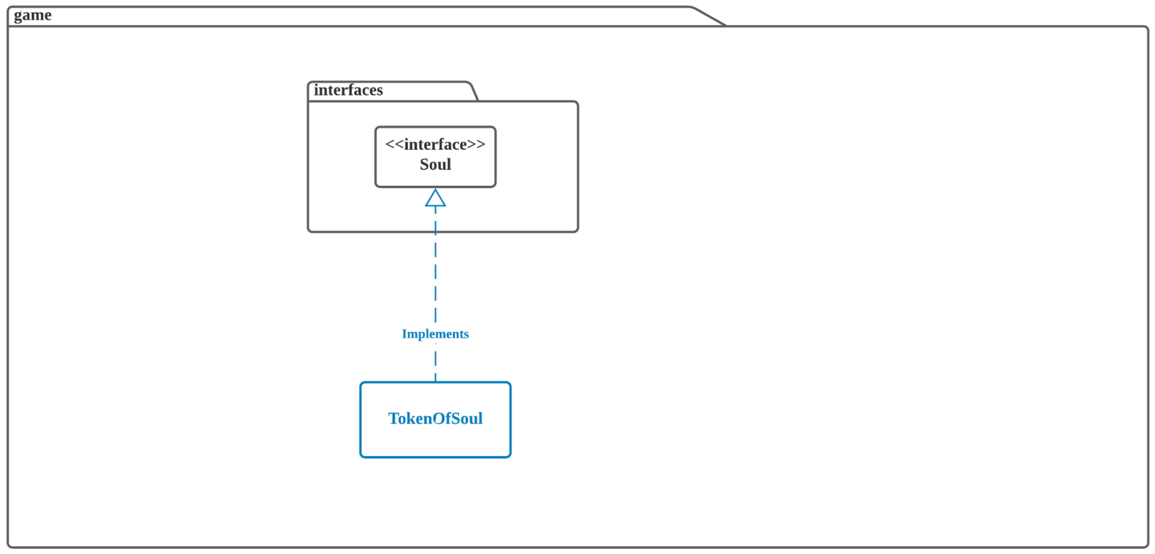
Furthermore, we create an association between the Enemy abstract class and Behaviour class. This is because, in the game, all the enemies will follow and start to attack the Player when it detects that the Player is in its surrounding (adjacent squares). This behaviour is implemented in the FollowBehaviour class. Next, by now, only the Skeleton can walk around on the map. This behaviour is implemented in the WanderBehaviour class. The FollowBehaviour and WanderBehaviour class implement the Behaviour interface. This means that the FollowBehaviour and WanderBehaviour Class can be declared as having the Behaviour data type due to the Polymorphism concept.

In the Enemy class, we can create an array list known as behaviours, which is declared as having a Behaviour data type that can be used to store all the Behaviour data type instances. Since all the specific enemies inherit the attributes in the Enemy class, all the Enemy child classes can have the behaviours array list. By doing that, we can store the behaviours instances that are only needed by certain enemies in its class. For example, we can add the WanderBehaviour instance which is of Behaviour data type into the behaviours array list attribute in the Skeleton class to implement its walk around requirements.

Alternatively, we can create an association relationship between each specific enemy class to the Behaviour interface. However, this creates unnecessary redundancy in codes which is not a good design. Thus, we did not use this approach. This is also one of the reasons we decided to create the Enemy class as the parent class and create only the association between the Enemy class and the Behaviour interface.

<u>Class Diagram on Souls</u> <u>(Requirement 1, Requirement 3, Requirement 4, Requirement 8):</u>

UML Diagram on Souls



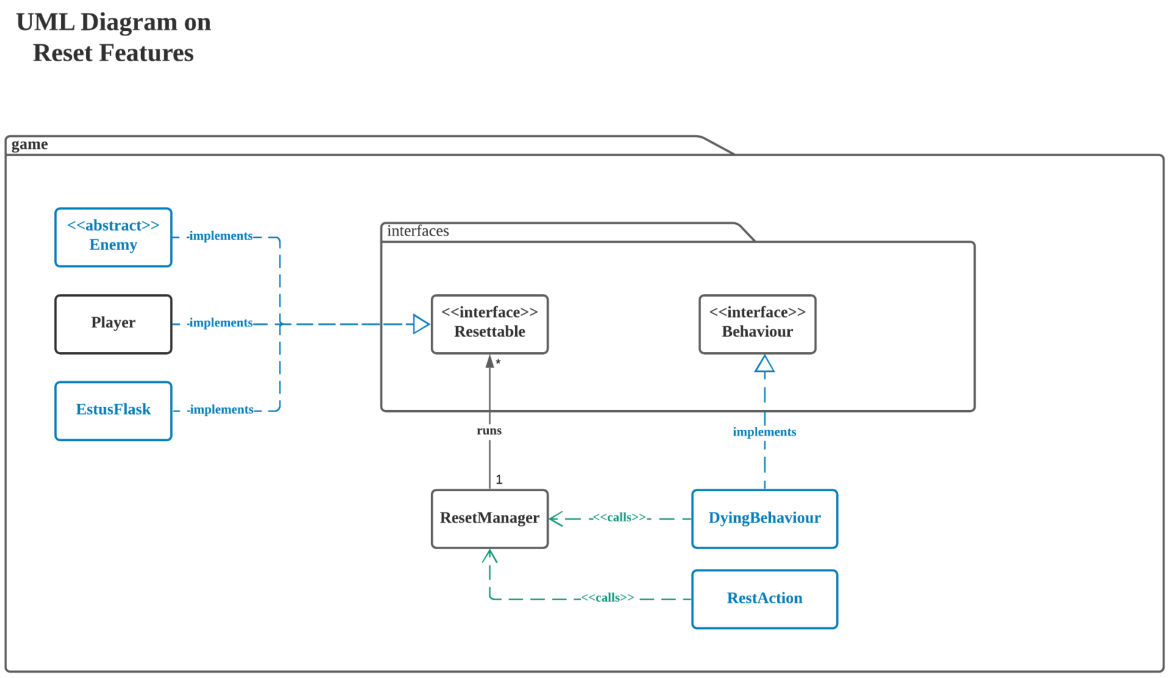
In the Souls UML diagram, we implemented the Soul interface in the newly created TokenOfSoul class.

1. **<u>TokenOfSoul</u>**

The reason why we implemented Soul Interface in the TokenOfSoul class is because it needs to store the number of Souls that is previously in the player when it dies. Therefore, the TokenOfSoul would need to implement Soul in order to be able to store and transfer Souls to and from the player using the transferSouls() method.

Alternatively, the GameWeaponItem and the Enemy class can also implement the Soul Interface as they involve the transfer of Souls. For example, when an enemy is killed by the Player, the number of Souls that are stored in the Enemy will be transferred to the Player. However, we think that this design is unnecessary because, in the current specification, the GameWeaponItem and Enemy each are worth a constant number of Souls. Therefore, we think that it would be a better design to just get the number of Souls of the weapon or enemy and just add the number of Souls of the Player using the addSoul() or subtractSoul() method in Player class after trading or killing an enemy.

<u>Class Diagram on Reset Features</u> <u>(Requirement 1, Requirement 2, Requirement 4, Requirement 6):</u>



In this game, we let the existing Player class and the 2 newly created Enemy classes and EstusFlask class extend the Resettable interface. Then, we create a DyingBehaviour class and let it implement the Behaviour interface. We also create a dependency relationship between the DyingBehaviour class and the RestAction class to the ResetManager.

1. **<u>Enemy Class implements Resettable interface</u>**

The reason why the Enemy class implements the Resettable interface is that all of the Enemy child classes such as the Undead, Skeleton and Lord of Cinders need to be resettable. Since they all need to be resettable, it would be a better design to inherit all of the required methods in the Resettable interface as they all share the same methods. In other words, these child classes can use this interface to reset their attributes and abilities. However, the methods inherited from Resettable will have different implementations. This will be further explained below.

2. **<u>Estus Flask implements Resettable interface</u>**

The reason the Estus Flask class implements the Resettable interface is because Estus Flask is resettable. Whenever the player performs a soft reset/dying in the game, Estus Flask will be refilled. Therefore, we think it would be a good design to perform the methods inherited from Resettable to perform the soft rest altogether with all the Enemy instances and Player instances.

3. **Player implements the Resettable interface**

The reason the Player implements the Resettable interface is because Player needs to be resettable. In the game, when the player rests or dies, the player would need to perform the RESET implementation. Therefore, we think it would be a good design to have the player implementing the Resettable interface so that we can perform the RESET feature when resting at the Bonfire and also when performing a soft reset.

4. **Soft Reset/Dying Implementation**

To start off, when a resettable instance is created such as all the subclasses of the Enemy class, Player and the Estus Flask, the instance will be registered to the ResetManager's resettableList attribute. This is done through the registerInstance() method implemented from the Resettable interface together with resetInstance() method and isExist() method in each subclass. Since the ResetManger is a singleton class, there would only be one instance of ResetManager and all instances will be registered to that same ResetManager instance.

To know when to perform a soft reset, a DyingBehaviour behaviour will be run at each of the Player's play turn in the playTurn method. The DyingBehaviour will be responsible to check when the player is consciously using the isConscious method at every turn. If the player is unconscious, it will invoke the run() method in ResetManager.

5. **Explanation - DyingBehaviour**

The reason why we implemented a DyingBehaviour is because we can check if the player is on a lethal terrain or object at each turn/tick so that in the future extensions of the game, we would only need to change the implementation of the DyingBehaviour to check if the Player would die instead of changing the run() method in ResetManager as the run() method is only responsible for running all the resetInstance() method for each registered Resettable instance in ResetManager.

Then, the run() method of the ResetManager instance will run the resetInstance() for each of the registered Resettable instances. Each instances' resetInstance() will perform all the actions required to reset each particular instance. For example, the resetInstance() in the Skeleton instance will increase the health point of itself to max health point and move itself to its initial position as per the requirements but the resetInstance() in the Undead instance will remove itself from the map instead.

Finally, when all of the resettable criteria are already being reset, the DyingBehaviour will then return a MoveActorAction instance that will move the Player back on top of the latest rested bonfire location as per the requirements.

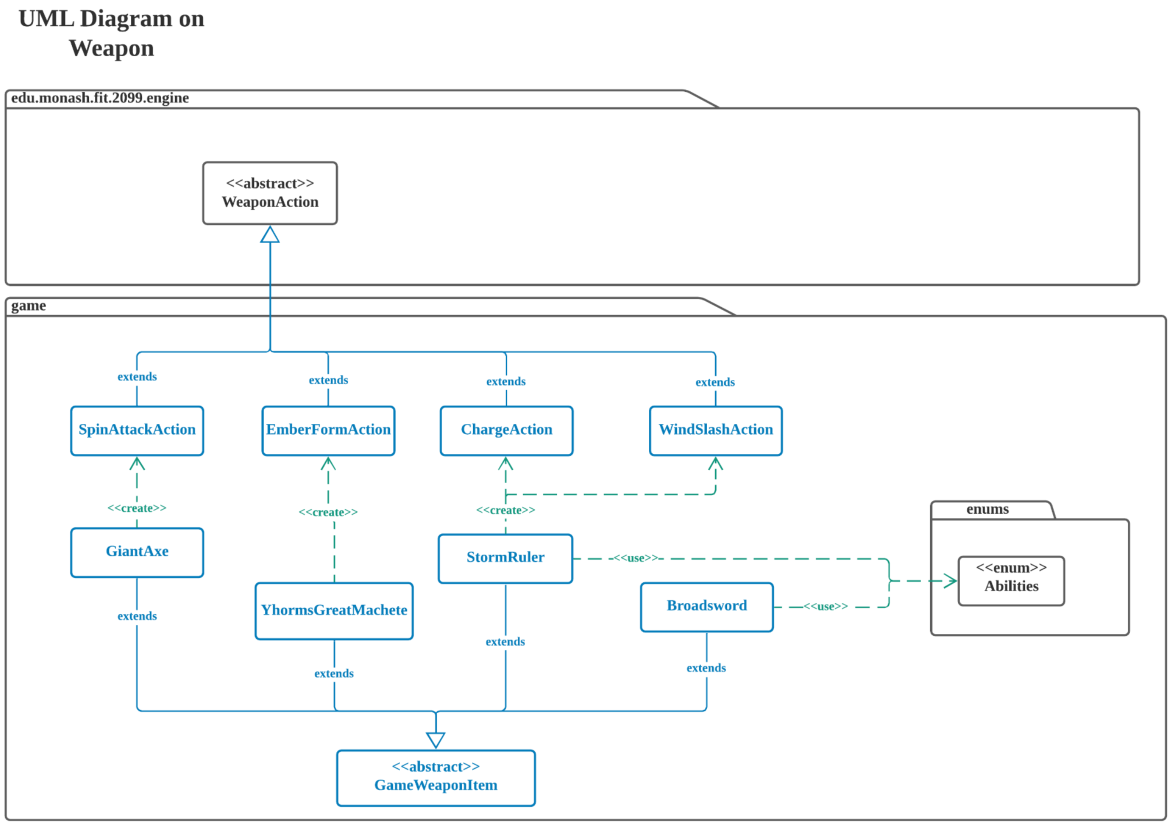
6. **Rest Action**

The reason why we implemented a RestAction class that calls the ResetManager in its class is to perform the RESET features when the player selects this action to rest at the bonfire.

7. **Changes to the ResetManager class**

In the ResetManager class, we decided it would be a better design to create a new method to perform the RESET feature when resting at the bonfire and leave the existing run() method to perform the soft rest/dying feature. In this case, we can use the ResetManager to perform both RESET and soft reset/dying features as they both involve resetting all of the resettable instances. This reduces the number of classes and reuse the current created class to satisfy overlapping features.

Class diagram on Weapon (Requirement 1, Requirement 4, Requirement 7):



1) Design rationale of the UML Diagram on Weapon Action

In this game, there are 4 active skills, which are the Spin Attack action which can be performed by the Giant Axe, the Charge and Wind Slash action which can be performed by the Storm Ruler and Ember Form action which can be performed by the Yhorm's Great Machete.

I create a class for each specific active skill action, which includes the SpinAttackAction, EmberFormAction, ChargeAction and WindSlashAction in the game package. All these classes will extend the WeaponAction class in the engine package which in turn extends the Action abstract class in the engine package. It is an inheritance relationship.

This is because all the active skill actions share similar attributes and methods that exist in the WeaponAction and Action parent class. Inheritance relationship enables code reusability, meaning all active skill action classes can inherit the common attributes or methods from the WeaponAction class and Action parent class without us repeating the same codes in each specific active skill action class.

For example, each active skill action is associated with a weapon object. The WeaponAction parent class has an attribute named weapon which is of WeaponItem data type representing a weapon that can use that particular action. By extending all the active skill action classes to the WeaponAction class, I do not have to code this attribute in each of the active skill action classes. This follows the Don't Repeat Yourself (DRY) design principle.

Next, active skills are skills that require action or input from the Player in order to trigger its effect. In this game, in order to invoke the specific active skills on the specific weapons, the player has to input the hotkey that represents the active skill action in the console to perform the skills. The WeaponAction class is an abstract class and it has abstract methods such as the execute() method and menuDescription() method. By making all the active skill action classes concrete and extend them to the parent WeaponAction class, it is compulsory for the active skill actions to implement those methods such as the menuDescription() method provide the description of that action to be selected by the player in the console or execute() method to provide the written text after the interaction with player. This ensures all the active skill action classes fulfill their requirements.

I do not create a general active skill action class to represent all the different active skills in the game. This is because although the active skill actions all share some similarities, they have some specific and unique attributes and methods that distinguish them from others. For example, each active skill action performs different functions and causes different levels of damage to other actors in the game. Each action also prints out a different message in the console when they are used. Therefore, each active skill action class that we created can override the execute() method, messageDescription() method and other methods inherited from the parent class specifically and respectively. Thus, it will be more convenient and comprehensive if we create a specific class for each active skill action that extends the WeaponAction class.

2) <u>Design rationale of the UML Diagram on Weapon Item</u>

In this game, I create a specific concrete class for each of the weapons used in this game, which are the GiantAxe class, YhormsGreatMachete class, StormRuler class and BroadSword class in the game package.

Then, we changed the GameWeaponItem class from a concrete class to an abstract class. This is because the GameWeaponItem class is now a base class to create more specific weapon child classes. The property of an abstract class is that it cannot be instantiated. In this case, we have created the specific weapon classes and we will just instantiate those specific weapon instances. We do not want to instantiate the base GameWeaponItem class. Thus, we make it an abstract class.

All these classes extend the abstract GameWeaponItem class in the game package which in turn extends the WeaponItem abstract class in the engine package. It is an inheritance relationship.

Firstly, the abstract GameWeaponItem class inherits all the attributes and methods that exist in the WeaponItem abstract class and all the parent classes of the WeaponItem abstract class. The attributes include the name, damage, hit rate, verb, etc. which are the common attributes of a weapon item. The methods include the damage method, getActiveSkill method, etc. which are the common methods that can be performed by a weapon.

By inheriting the abstract GameWeaponItem class, each specific weapon item class can inherit the common attributes and methods from the WeaponAction class. This prevents us from repeating the same code in each of the newly created weapon item classes. By doing that, we have obeyed the Don't Repeat Yourself (DRY) design principle in software design and development.

Alternatively, we can use the general abstract GameWeaponItem class to create all the weapon objects that are used in the game. However, we do not use this approach because although each weapon shares some common attributes and methods, they still have some specific and unique attributes and methods that distinguish them from others. Furthermore, in the future, if we want to extend the game to include more different weapons, each with more unique attributes, using just the abstract GameWeaponItem class to create all the weapons will not be practical. Thus, we create a specific concrete class for each of the weapons used in this game.

3) <u>Explanation on how to handle the passive skills of the weapons</u>

Unlike the active skills, we do not create each passive skill as the child class of the WeaponAction class. This is because passive skills are skills that will upgrade players' overall states and they cannot be used or invoked by the player directly. However, all the child classes of the Action Class have methods such as the menuDescription(), getHotKey() and execute() method which enables these actions to be triggered by the player. These are not needed for passive skills. Thus, it will not be appropriate if we create the passive skills as the child classes of the Action class.

We choose to include all the passive skills in the Abilities class in the enum package. This is because the Abilities enum class stores constants that represent the permanent condition or ability. The critical strike passive skill will always have a 20% success rate to deal double damage with a normal attack. The Dullness passive skill will always decrease the effectiveness of Storm Ruler when Storm Ruler is used to attack the enemies that are not weak to it. Thus, we can include the constants that represent the passive skills such as the CRITICAL_STRIKE and DULLNESS in the Status enum class.

4) **<u>Explanation on the dependency relationship between some GameWeaponItem child classes and Ability enum class</u>**

Then, we create a dependency relationship between the StormRuler class and Broadsword class to the Ability enum class. The relationship is described as the StormRuler class and Broadsword class use the Ability class. This is because the StormRuler weapon is associated with critical strike and dullness passive skills whereas the Broadsword weapon is associated with critical strike passive skill in the game.

In terms of the implementation, we can add the enum constants in the Abilities class from the enum package which represent the passive skills such as CRITICAL_STRIKE and DULLNESS into the capabilities set attribute in StormRuler class and we can add the enum constants DULLNESS into the capabilities set attribute in Broadsword class. Since both classes have Abilities enum constant in their class, it shows that the StormRuler Class and Broadsword class depend on the Abilities enum class to fulfill their passive skills requirements.

Then, during an attack, the weapon will be used in the execute() method in AttackAction class. We can then check if the weapon has a certain enum constant that represents the passive skill which determines the effectiveness of the weapon and which alters the damaging effect caused by the weapon using the hasCapability() method. We can then implement the effect accordingly in the execute() method.

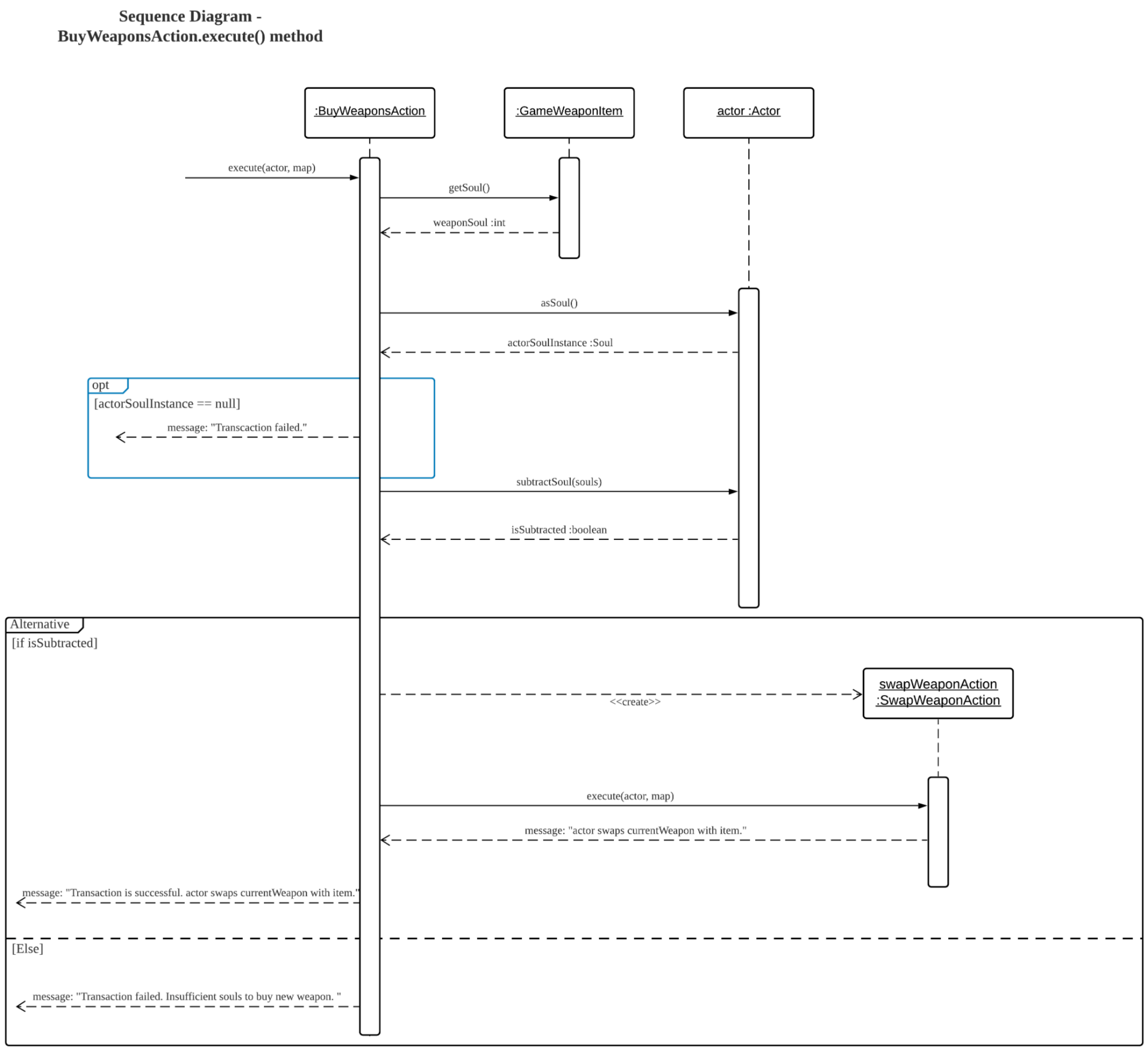
5) **<u>Dependency between each specific weapon class and active skill attack action class</u>**

In this case, I create a dependency relationship between each specific weapon class (GameWeaponItem child class) with the active skill attack action class (WeaponAction child class) as shown in the UML diagram on Weapon. Each specific weapon class has a getActiveSkill() method implemented from the Weapon interface. Inside this method, we will create the appropriate active skill action class instance that the weapon should have and the method will return that instance.

For example, in the game, the Giant Axe can use the spin attack skill. Thus, inside the getActiveSkill() method in the GiantAxe class, we will create a SpinAttackAction instance. Then, the method will return the SpinAttackAction instance. This also shows that the GiantAxe class depends on the SpinAttackAction class such that it <<create>> SpinAttackAction class. The same applies to the other specific weapons with their corresponding active skill attack action as shown in the UML diagram on Weapon.

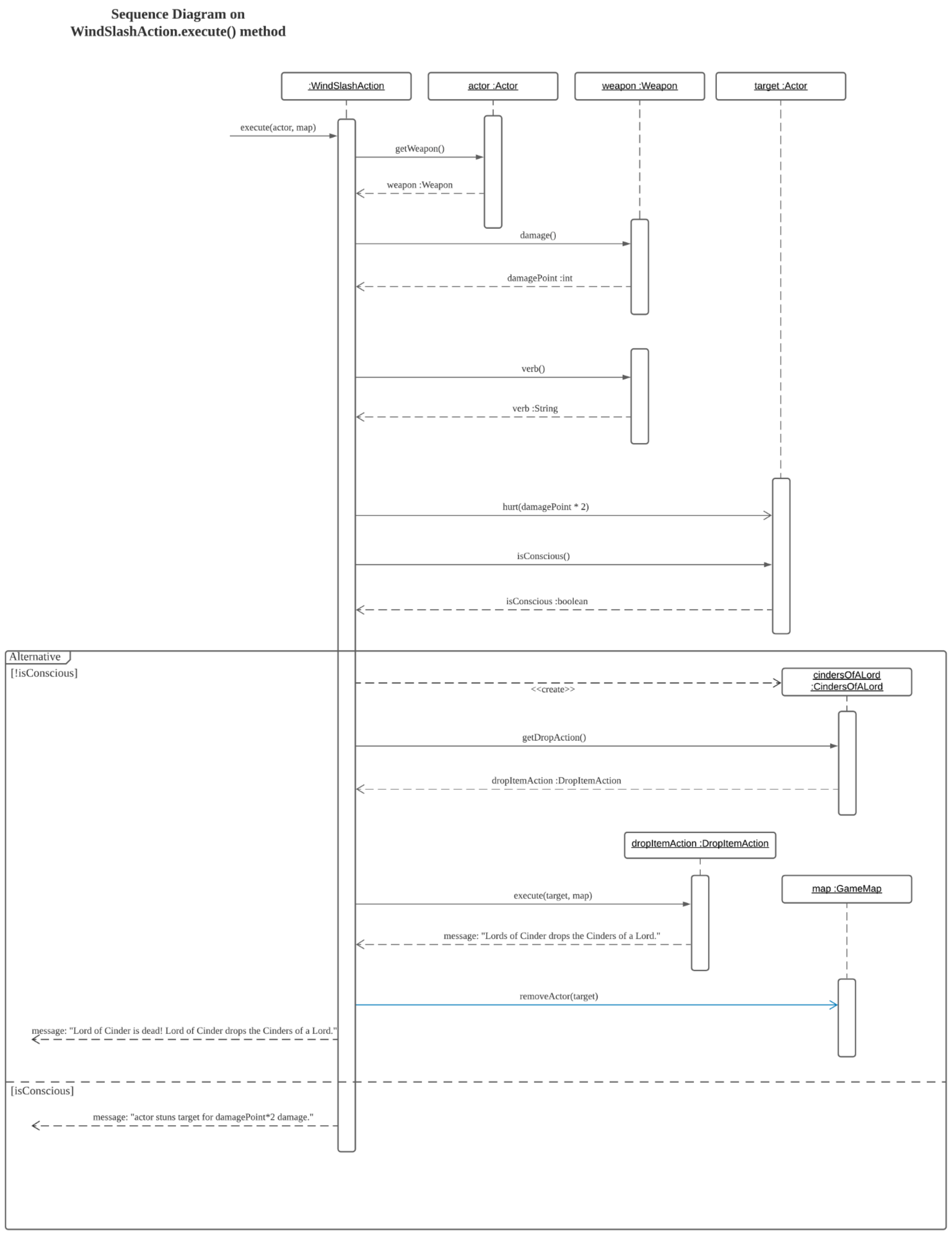
Alternatively, I can create the active skill action instance in the Actor class. For example, I can create the SpinAttackAction instance inside the Player class such that the Player can use the action. However, this will result in additional dependency between the game weapon item with the Player who is using the action. This is because we have to check if the weapon that uses this action is the appropriate weapon. This requires the use of if-else statements to check the class type of the weapon. This results in unnecessary dependency. Hence, we do not select this alternative. We decided to use the former approach which follows the Reduce Dependency Principle design principle.

Sequence diagrams - BuyWeaponAction.execute() method



Sequence Diagram - BuyWeaponsAction.execute() method
:BuyWeaponsAction
:GameWeaponItem
actor :Actor
execute(actor, map)
getSoul()
weaponSoul :int
asSoul()
actorSoulInstance :Soul
opt
[actorSoulInstance == null]
message: "Transcaction failed."
subtractSoul(souls)
isSubtracted :boolean
Alternative
[if isSubtracted]
<<create>>
swapWeaponAction :SwapWeaponAction
execute(actor, map)
message: "actor swaps currentWeapon with item."
message: "Transaction is successful. actor swaps currentWeapon with item."
[Else]
message: "Transaction failed. Insufficient souls to buy new weapon. "

Sequence diagrams - WindSlashAction.execute() method


Sequence Diagram on WindSlashAction.execute() method
:WindSlashAction
actor :Actor
weapon :Weapon
target :Actor
execute(actor, map)
getWeapon()
weapon :Weapon
damage()
damagePoint :int
verb()
verb :String
hurt(damagePoint * 2)
isConscious()
isConscious :boolean
Alternative
[!isConscious]
<<create>>
cindersOfALord :CindersOfALord
getDropAction()
dropItemAction :DropItemAction
dropItemAction :DropItemAction
execute(target, map)
map :GameMap
message: "Lords of Cinder drops the Cinders of a Lord."
removeActor(target)
message: "Lord of Cinder is dead! Lord of Cinder drops the Cinders of a Lord."
[isConscious]
message: "actor stuns target for damagePoint*2 damage."