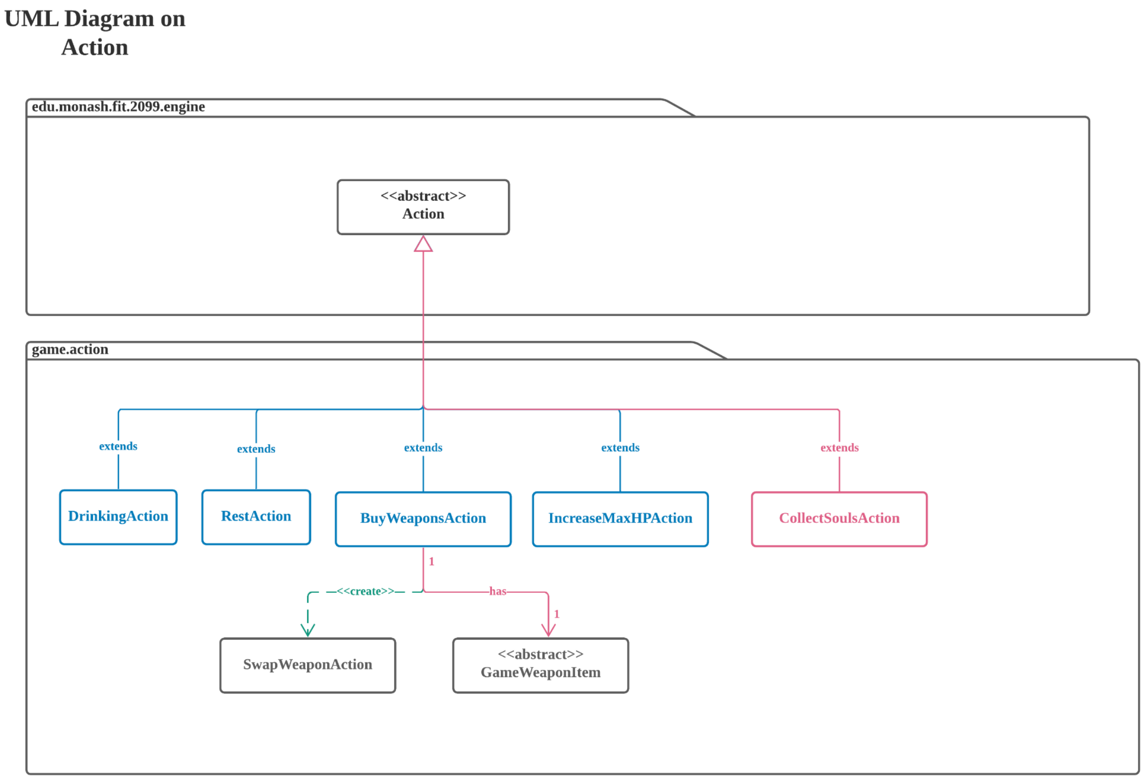
MA_Lab2Team4
Assignment 2 Amendments to Design Rationale

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)**:**



Inheritance relationship:

Action class is an abstract class. It is a base class which represents the things that the player can do. In other words, it represents the action that the player 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 conditions after the action invoked is performed.

We create the inheritance relationship as shown in the UML class diagram because it allows all the child action classes to inherit all the attributes and methods from the Action abstract parent class. This prevents us from repeating similar codes in the application, which follows the Don't Repeat Yourself (DRY) principle. For example, the DrinkingAction, RestAction, BuyWeaponsAction, IncreaseMaxHPAction and CollectSoulsAction classes will need to inherit and override the menuDescription() method from the Action class in order to display the selection for each of these actions in the console. They also need to inherit and override the execute() method to implement their own requirements and return the results of the action.

By doing that, we also obey the Open-Closed Principle, which states that software entities, such as classes, modules and functions should be open for extension and closed for modification. By using the above design, we do not need to modify codes in any of the existing classes when we create a new specific action class in the future. This shows that the system is easier to extend in the future.

Next, we apply the Single Responsibility Principle in our design as each of the DrinkingAction, RestAction, BuyWeaponsAction, CollectSoulsAction and IncreaseMaxHPAction classes only have one responsibility which is to perform their own specific task and action.

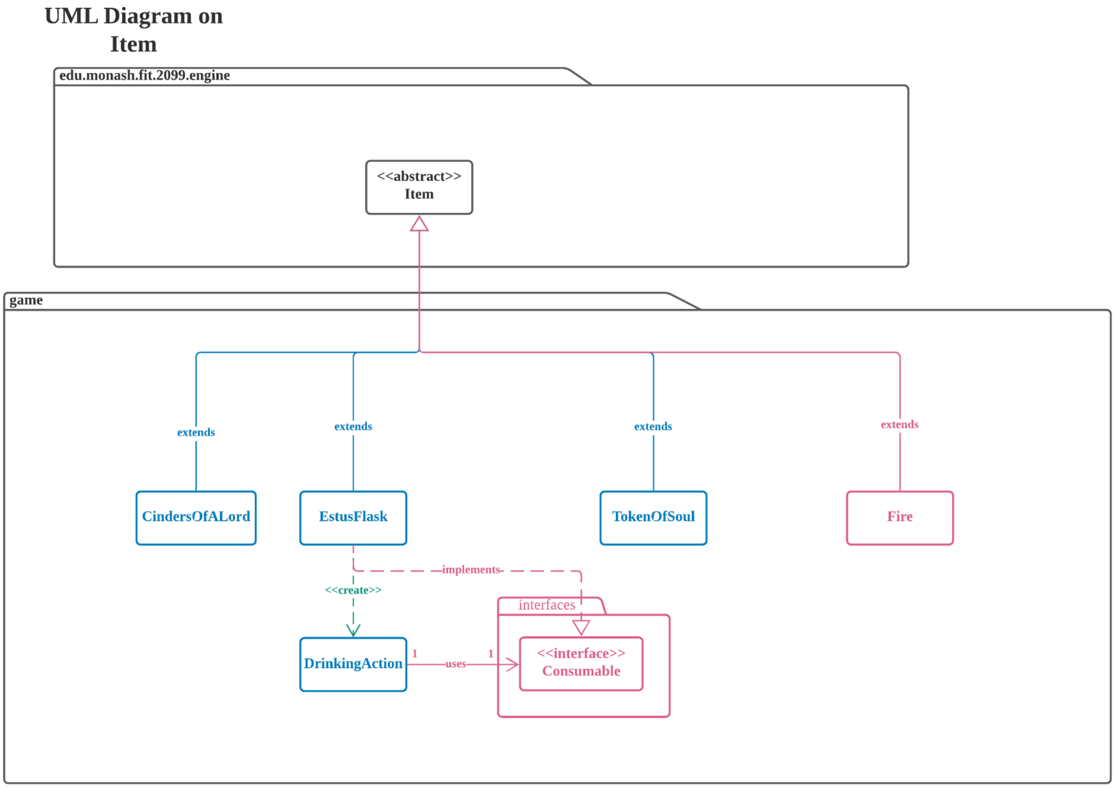
<u>BuyWeaponsAction class has GameWeaponItem class:</u>

The BuysWeaponsAction class has an attribute named weapon which is of the GameWeaponItem class type. This attribute references the weapon that is bought by the actor (Player). The BuyWeaponsAction holds a class level reference to the GameWeaponItem class. Thus, there is an association between these two classes.

<u>BuyWeaponsAction class <<create>> SwapWeaponAction class:</u>

In our game, when the Player buys a new weapon from the Vendor, the old weapon in the current inventory will be automatically replaced with it. Thus, inside the execute() 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 has a reference to the SwapWeaponAction class as part of its execute() method. Thus, the BuyWeaponsAction class depends on the SwapWeaponAction class to replace the old weapon with a new weapon. Without the SwapWeaponAction class, the BuyWeaponsAction class cannot fulfill the requirements completely. Thus, BuyWeaponsAction class <<create>> SwapWeaponAction class.

Class diagram on Item (Requirement 1, Requirement 2, Requirement 4, Requirement 8):



1) **Why remove the PortableItem class in the game package**

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 base 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) **Inheritance relationship**

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.

We decide to create the inheritance relationship between Fire class and Item class because Fire is an item type that can be placed on the ground in the game during burning when the YhormTheGiant activates its Ember Form behavior. Any other actors that are at the location where the Fire is placed will be hurt with 25 hit points damage. The damage can be implemented inside the tick() method in the Fire class that is inherited from the parent Item abstract class.

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) **Estus Flask class <<create>> DrinkingAction class**

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 class, EstusFlask class cannot fulfill its requirements completely. Thus, the EstusFlask class has a reference to the DrinkingAction class as part of its constructor. Thus, EstusFlask class <<create>> DrinkingAction class.

We apply the Single Responsibility Principle in our design as each of the CindersOfALord, EstusFlask, TokenOfSoul and Fire classes all have functionalities or methods that they need in their own class. Each of them has only a single responsibility.

Why we chose to implement the burning action using the Fire class (New design) instead of adhering to using the BurningGround (Original design)

We decided to **implement the burning action using the Fire item class instead of using the BurningGround class because implementing this way is easier**. If we use the BurningGround class, inside the tick() method of the BurningGround class, we have to set the current ground type to Dirt ground after burning. This may cause the BurningGround class to have a dependency with the Dirt ground type. Alternatively, if we implement the burning action using the Fire class, inside its tick() method, we can just place the Fire item on the ground during burning and remove itself from the ground after burning. It involves fewer dependencies.

Consumable Interface

There are 2 approaches to implement the DrinkingAction requirement. The first approach is **creating an association relationship between the Estus Flask and DrinkingAction class**. By doing that, the DrinkingAction class can have the EstusFlask instance as its attribute and use the EstusFlask class methods to update the charge of the EstusFlask instance. We believe that this approach is not a good design because the DrinkingAction class depends directly on the EstusFlask class to fulfill the requirements.

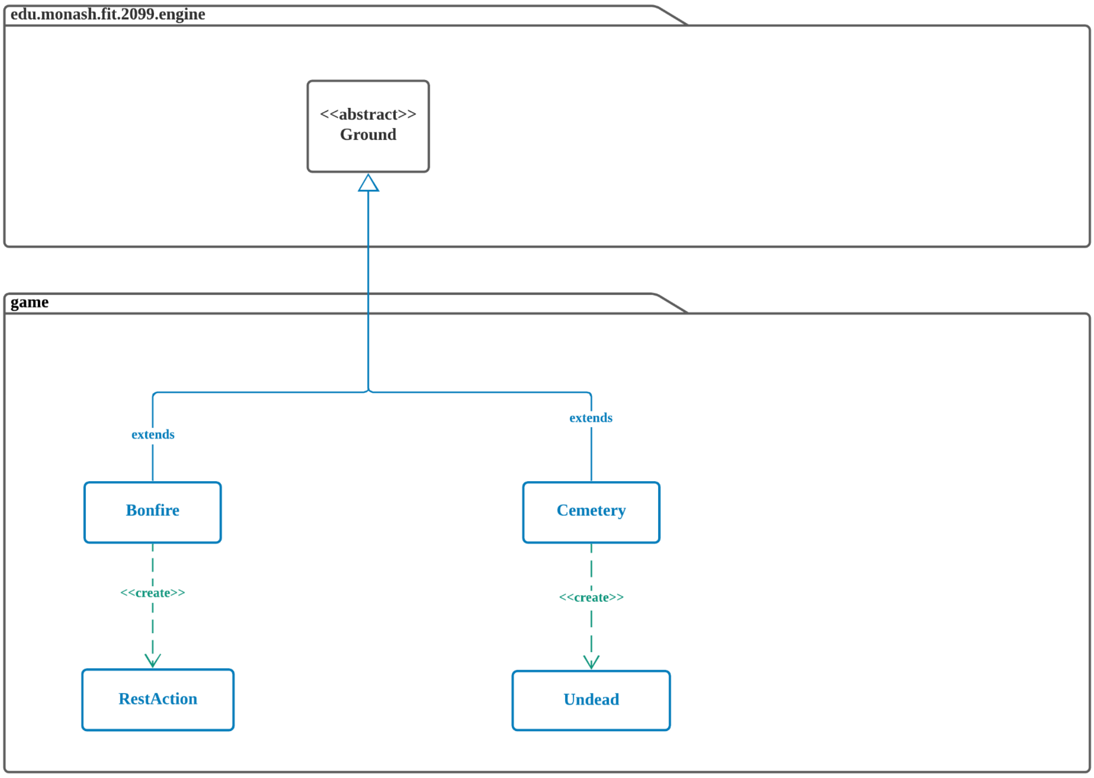
Thus, we decided to use the second approach, which is applying the dependency injection concept and dependency inversion principle to implement the drinking action requirement in the game. **We let EstusFlask implement the Consumable interface and we create an association relationship between the DrinkingAction and Consumable interface.** By doing that, we can reduce the association relationship between the EstusFlask class and DrinkingAction class. This allows the DrinkingAction class to be independent from the services (EstusFlask) it is relying on. Both classes now do not depend on low-level modules but depend on abstractions.

In the future, for any classes that are consumable, we can let these classes implement the Consumable interface. The DrinkingAction class which is responsible for updating the charges of all the consumable classes will just have an association relationship with the Consumable interface, instead of having an association relationship with each of the concrete classes.

By doing that, we are adhering to the **dependency inversion principle** which suggests that high-level modules do not depend on low-level modules (concretion) but both depend on abstraction.

Class diagram on Terrains (Requirement 4, Requirement 5, Requirement 8)**:**

UML Diagram on Terrains



Inheritance relationship:

Bonfire and Cemetery are both ground types that are display with unique characters “#” and “C” respectively on the map. The inheritance relationship allows both terrains to inherit the attributes and methods from the Ground abstract class. For example, the Ground abstract class contains an allowableActions() method which can be overridden by the child classes to implement the different actions that can be performed by the Player on that ground. It returns an empty Actions instance which has an actions array list attribute that stores the different Action instances that can be performed by the Player on the corresponding ground.

Dependency relationship:

1) **Bonfire class <<create>> RestAction class**

Firstly, in the game, the Bonfire is an area in the middle of the map where the Player starts. It 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>> RestAction class.

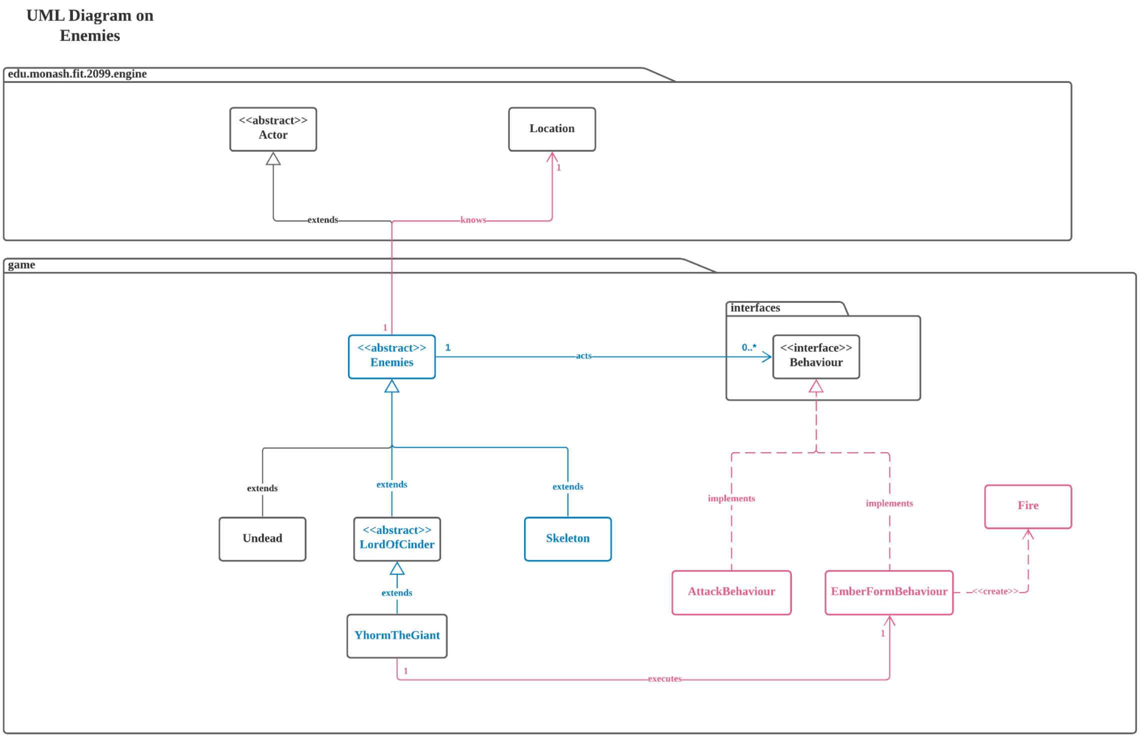
2) **Cemetery class <<create>> Undead class**

The 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(Location location) 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.

The Bonfire, Cemetery, Valley, Floor, Wall, Dirt, classes all obey the Single Responsibility Principle as they commonly depend on their abstract parent classes and each class has only a single responsibility to fulfill.

Class diagram on Enemies (Requirement 4)**:**



1) **Inheritance relationship between the Enemies class with Undead, LordOfCinder and Skeleton class**

Initially, we have only the Undead and LordOfCinder classes created in the game package which extend the Actor abstract class in the engine package.

We create an Enemies abstract class and a Skeleton class in the game package. The abstract Enemies class is created as there are various methods and attributes that are shared with each of the specific enemy classes. It extends the Actor abstract class in the engine package. We set the Enemies class as abstract instead of concrete class because it is a base class to create more specific enemies 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 Enemies class. Thus, we make it an abstract class.

Then, we let the Undead, LordOfCinder and Skeleton class extend the abstract Enemies 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 behavior and others. Thus, we can include all the common attributes and methods inside the abstract Enemies class. Then, all the child classes can inherit and override the attributes and methods from the Enemies abstract class. By doing that, we do not have to repeat similar codes in all the child classes and we have obeyed the Don't Repeat Yourself (DRY) design principle.

As mentioned previously, we create the Skeleton class that extends the Enemies class. This is because the Skeleton class can inherit all the common attributes and methods of an enemy from the Enemies 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.

By doing that, we also obey the **Open-Closed Principle**, which states that software entities, such as classes, modules and functions should be open for extension and closed for modification. By using the above design, we do not need to modify any existing code in the existing classes when we create a new specific enemy class in the future. We can just extend it to the Enemies abstract class. This shows that the system is easier to extend in the future.

2) **Changing the LordOfCinder from concrete class to abstract class**

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.

By using such a design, we obey the **Open-Closed principle as** we are able to add specific bosses with new features into our game without changing any existing code in the existing classes in the future.

3) **Association between the Enemies Class to Behaviour Interface**

Furthermore, we create an association between the Enemies abstract class and the Behaviour interface. 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 behavior is implemented in the FollowBehaviour class. Next, by now, only the Skeleton and Undead can walk around on the map. This behavior 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 Enemies class, we can create an array list known as behaviors, 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 Enemies parent abstract class, all the Enemies child classes can have the behaviors Java array. By doing that, we can store the behaviors 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 behaviors Java array 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 Enemies class as the parent class and create only the association between the Enemies class and the Behaviour interface.

4) **AttackBehaviour and EmberFormBehaviour class implements Behaviour interface**

The AttackBehaviour class and EmberFormBehaviour class implements the Behaviour interface to use the getAction() method and the getPriority() method for their implementations. The reason why the child behavior classes are designed in this way is because it will be easier for the future extension of the application when we add more specific child behavior classes into the application.

5) **YhormTheGiant class executes EmberFormBehaviour class / EmberFormBehaviour class <<create>> Fire class**

According to the requirements, Yhorm The Giant which holds Yhorm Great Machete executes the ember form behavior under certain conditions. Thus, we create EmberFormBehaviour instance attribute in YhormTheGiant class so that YhormTheGiant class can use the attributes and the actions of the EmberFormBehaviour instance for its implementation. Thus, we create the one-to-one association relationship between the YhormTheGiant class and the EmberFormBehaviour class.

Next, when the Ember Form behavior is executed, the Yhorm The Giant which holds Yhorm Great Machete will burn the Dirt ground in its adjacent squares. Thus, inside the EmberFormBehaviour class, it will create a Fire instance in its getAction() method. By doing that, EmberFormBehaviour class depends on the Fire class to implement its requirement, Thus, EmberFormBehaviour <<create>> fire.

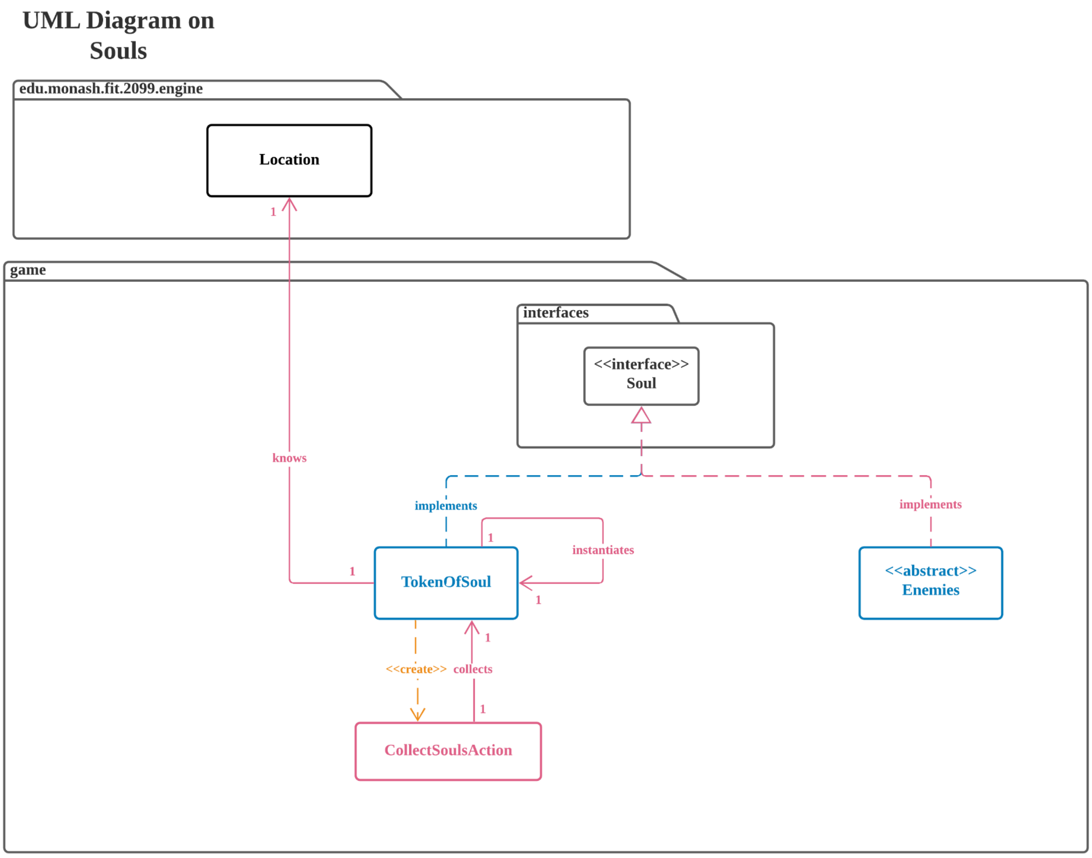
While designing the Enemies classes, we obey the Liskov-Substitution Principle, which states that every subclass or derived class should be responsible for their base or parent class. This is because all the Skeleton, Undead, YhormTheGiant classes can inherit and fulfill the requirements of their parent Enemies class. Whatever methods and functionalities that are implemented in the parent class can also be used by the child enemies classes.

As an example, these classes reference the same playTurn() method as seen in the abstract Enemies class. We can also add the specific attributes and methods in the child enemies classes in future.

6) <u>Enemies class knows Location class</u>

The Enemies class has an attribute named initialLocation of Location type. The attribute is used to store the initial location of the enemy that is spawned or created in the game map. By doing that, the enemies can be reset to their initial positions when the reset action is executed. Since the Enemies class has an instance variable of Location type, the Enemies class has an association relationship with the Location class.

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



1) **<u>TokenOfSoul class and Enemies class implement the Soul interface</u>**

By doing that, TokenOfSoul and Enemies class can implement various methods such as the transferSouls, addSouls, subtractSouls and getSouls methods. For the TokenOfSouls, these methods enable the actors in the game to have interaction with the token of soul and allow the actors to fulfill their needs to transferSouls, increase or decrease their current souls.

Soul Interface is also implemented by the Enemies class because the Enemies class would need to implement Soul in order to implement the specified methods in the interface to store and transfer Souls to and from the player using the transferSouls() method.

2) **TokenOfSoul instantiates itself**

TokenOfSoul is a singleton class. The reason why we implemented TokenOfSoul as a singleton class is because there is only one instance of TokenOfSoul needed in the game. Whenever the player dies, it will be removed and spawn at the new spawn location. Therefore, we can just reuse the same instance without creating multiple TokenOfSoul instances. Soul interface is implemented in the TokenOfSoul class 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 a Soul interface to be able to store and transfer Souls to and from the player using the transferSouls() method.

3) Association and Dependency between TokenOfSoul and CollectSoulsAction

The TokenOfSoul class is a singleton class and has a dependency and association relationship with the CollectSoulsAction class. The reason why we designed the CollectSoulsAction class is to have an association relationship with the TokenOfSoul class so the actor is able to collect the TokenOfSoul item through the CollectSoulsAction class.

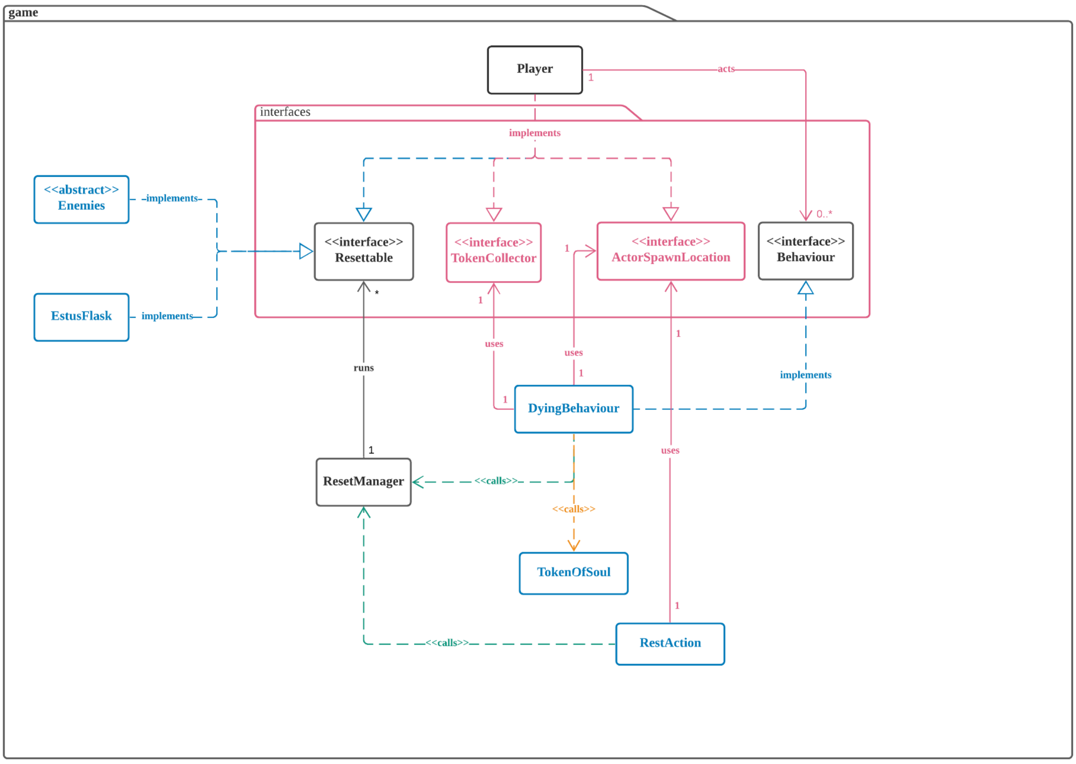
The TokenOfSoul class obeys the Single Responsibility Principle as it has all its required functionality in its one class.

4) TokenOfSoul class knows Location class

The TokenOfSoul class has an attribute named tokenSpawnLocation of Location type. The attribute is used to store the location of the token that should be spawned when the player dies. Since the TokenOfSoul class has an instance variable of Location type, the TokenOfSoul class has an association relationship with the Location class.

Class Diagram on Dying in this game/Soft Reset (Requirement 1, Requirement 2, Requirement 4, Requirement 6):

UML Diagram on
Dying in this
game/Soft Reset



The Player class implements the new TokenCollector interface and the new ActorSpawnLocation interface, these classes are created to help the game locate the latest token and actor spawn location. The DyingBehaviour class also has an association relationship with both of these classes as it is required for the tokenOfSoul and the actors to be spawned back at the latest spawned location, in other words, its initial location. The Player also acts on Behaviours so that the Player can use the different behaviours functionalities. Currently, it only acts on DyingBehaviour.

1) **Enemies Class implements Resettable interface**

The reason why the Enemies class implements the Resettable interface is that all of the Enemies child classes such as the Undead, Skeleton and Lord of Cinders have to be resettable. Since they all need to be reset, 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 Enemies instances and Player instances.

3) **<u>Player implements the Resettable interface, TokenCollector interface, ActorSpawnLocation interface.</u>**

The Player implements the Resettable interface because Player needs to be reset. In the game, when the player rests or dies, the player would need to perform the RESET implementation. Therefore, we can perform the RESET feature when resting at the Bonfire and also when performing a soft reset. The Player also implements the new TokenCollector and ActorSpawnLocation interfaces as we need to validate the spawned token location and the actor spawned location when performing the soft reset.

We are obeying the Interface Segregation Principle which states that no client should be forced to implement a method that they do not use by creating 2 different new interfaces which handle different locations spawning actions. The TokenCollector interface contains methods that spawn the token whereas the ActorSpawnLocation interface contains methods that spawn the actor. Each of these interfaces are small and have their required functions and purposeful functionalities while representing quality that the implementing code should have.

We do not put all the methods inside one interface, considering that some classes may not need to spawn both the token and actor locations at the same time. For example, the RestAction class only needs to reset or spawn the actor to its initial location, it requires and uses only the methods in the ActorSpawnLocation interface. Thus, it is unnecessary for it to access the methods about spawning the token.

4) **<u>Soft Reset/Dying Implementation</u>**

To start off, when a resettable instance is created such as all the subclasses of the Enemies 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 turns in the playTurn method. The DyingBehaviour will be responsible to check when the player is conscious by using the isConscious method at every turn. If the player is unconscious, it will invoke the run() method in ResetManager.

5) DyingBehaviour

The reason why we implemented a DyingBehaviour is because we can check if the player is conscious or not 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 cleaning up the resettableList using the cleanUp() method and 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.

The **DyingBehaviour class has an association relationship with the TokenCollector and ActorSpawnLocation interfaces**, as when the soft reset functionality occurs, we need to locate the new token spawn's location as well as the actor's latest spawn location so that the actor and TokenOfSouls are able to return to the latest spawn location.

The DyingBehaviour class also has a dependency relationship with the TokenOfSouls class as it calls the token of soul to be spawned when the soft reset occurs.

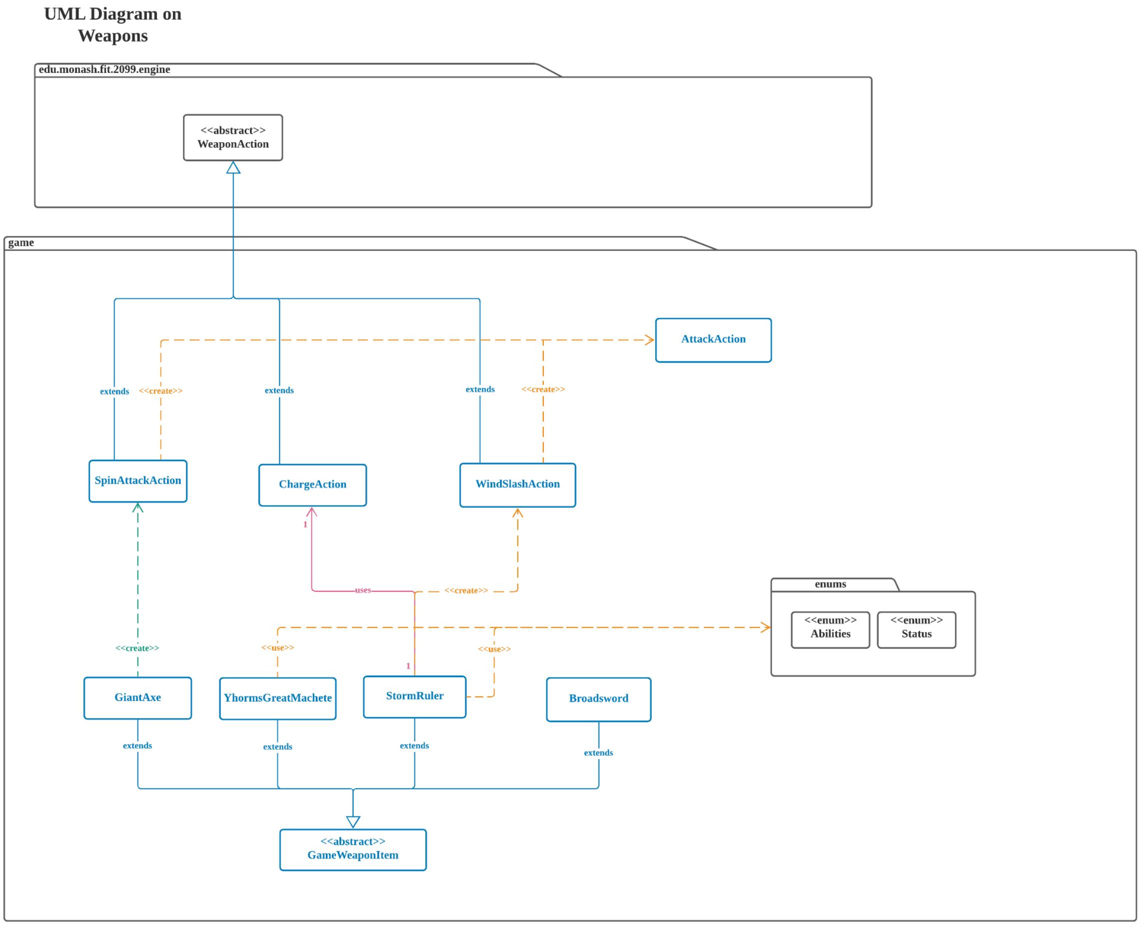
6) **Rest Action**

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. The RestAction class uses the ActorSpawnLocation interface to locate the Location of the actor instance to be spawned and sets the Player back to the rightful location in the game when soft reset occurs which is 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 reuses the current created class to satisfy overlapping features.

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



1) **Design rationale of the UML Diagram on Weapon Action**

We create a class for each specific active skill action, which includes the SpinAttackAction, 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, which can be inherited by all its child 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. 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 the 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.

By doing that, we obey the **Open-closed design principle,** where the classes are open for extension but closed for modification. We can easily implement new active skills in the game without modifying the existing code in the existing classes.

2) Design rationale of the UML Diagram on GameWeapon Item

In this game, we 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 specific weapon concrete 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. By inheriting the abstract GameWeaponItem class, each specific weapon item class can inherit the common attributes, such as damage, hitRate, etc. and methods, such as damage(), verb() 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. By doing that, we are obeying the Open-closed principle.

3) YhormsGreatMachete and StormRuler use Abilities and Status Enum

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 decide to use the Enum constants in Abilities and Status class to implement the passive skills requirements. The Abilities enum class stores constants that represent the permanent condition or ability. The Status enum class stores constants that represent the temporary condition. For example, we will add the Status.RAGE_MODE Enum constant into the capabilities list of the YhormGreatMachete class so that it will pass the if-statement in the overridden chanceToHit method and increase its hitRate by 30%. Similarly, for StormRuler, we will use the Enum constants such as Status.CHARGING, Status.FULLY_CHARGED, Abilities.WEAK_TO_STORM_RULER, etc to update the status of the StormRuler and determine if it can be charged and it can invoke the Wind Slash active skill.

4) **<u>GiantAxe <<create>> SpinAttackAction and StormRuler <<create>> WindSlashAction</u>**

In the game, the Giant Axe can use the spin attack skill and StormRuler can use the wind slash action. The getActiveSkill() action in these 2 classes can create and return the appropriate active skill action class instance that these weapons should have.

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.

The same concept applies to the StormRuler class where we create a WindSlashAction instance inside the getActiveSkill() method in the StormRuler class. StormRuler depends on the WindSlashAction class to fulfill its requirement such that it <<create>> WindSlashAction class.

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.

5) <u>StormRuler class uses ChargeAction class</u>

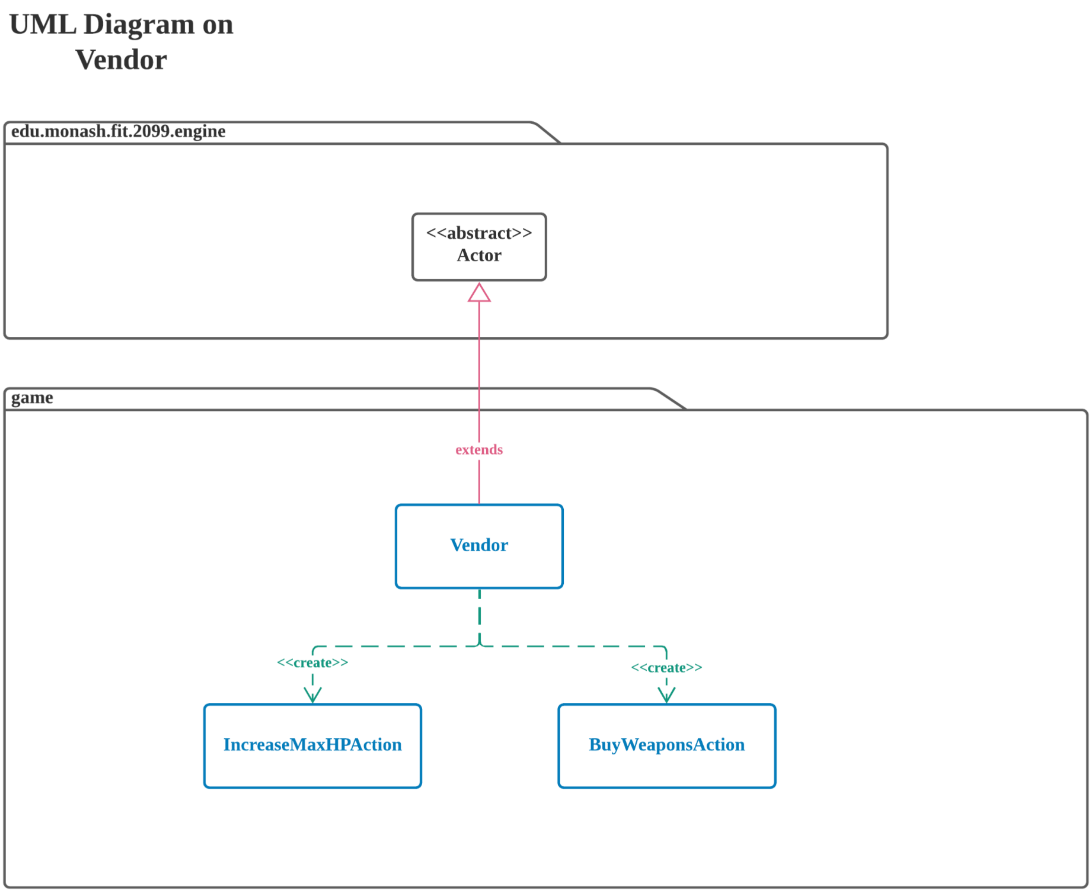
According to the requirements, StormRuler has a charge action. Both of these active skills can only be invoked under certain conditions. For the charge action, it will automatically charge itself for 3 turns once the user invokes this action in the console. Thus, in the StormRuler class, we first create a ChargeAction instance attribute and it is by default referencing the null.

Inside the StormRuler class, the tick() method will only be executed when the Player is holding the StormRuler. Thus, inside the tick() method, we will create a ChargeAction instance and call the execute() method of the instance to execute the charging action. Thus, we can see that the StormRuler depends on the ChargeAction instance to complete the charging of itself. When we remove the StormRuler instance, the ChargeAction instance will also be removed. This shows an association relationship.

6) SpinAttackAction class and WindSlashAction class <<create>> AttackAction class

The main reason for creating dependency relationships between these 2 active skill actions classes with the AttackAction class is to reduce code redundancy. This is because the codes inside the attack() method in the AttackAction class which perform the weapon damage and conditions checking after the damage are also needed in the SpinAttackAction class and WindSlashAction class. Thus, in order to follow the Don't Repeat Yourself (DRY) principle, we decided to create an AttackAction instance in the SpinAttackAction class and WindSlashAction class. Then, we can invoke the attack() method of these created AttackAction instances to implement the attack requirements. The cons of this approach is that we will be creating dependencies between both SpinAttackAction class and WindSlashAction class with AttackAction class.

<u>Class diagram on Vendor UML</u> (<u>Requirement 8)</u>:



<u>Vendor class extends Actor class</u>

Previously, we let the Vendor class extend the Ground abstract class. However, we now create a new UML diagram for the Vendor class where the Vendor class extends the Actor abstract class. This is because the Vendor and the Actor are of similar type. The Vendor contains many attributes and methods that can be reused or overridden by the Vendor class to fulfill the requirements of the Vendor. By using the inheritance relationship, we can reuse and override the codes that are contained in the Actor class to implement the Vendor class functionalities. By doing that, we reduce the code redundancy and achieve the Don't Repeat Yourself (DRY) design principle.

However, the cons of the design is that 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.

Vendor <<create>> IncreaseMaxHPAction and BuyWeaponsAction

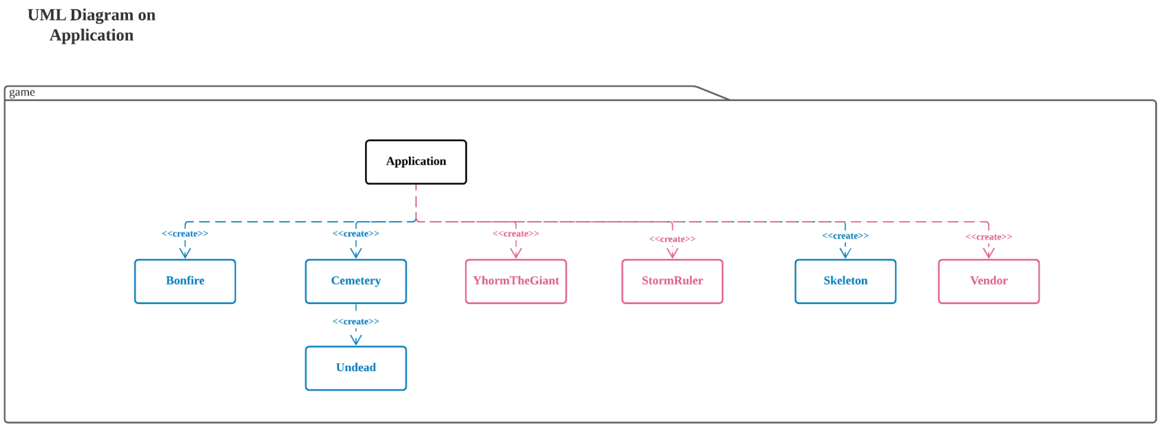
The Player can choose to trade souls with the Vendor, also known as the FireKeeper to buy new weapons or increase his own maximum hit points. The Vendor is displayed as character “F” on the map. By inheriting the attributes and methods from the Actor abstract class, we can set the displayChar attribute of the Vendor class to “F”. 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 Actor 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, it 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.

The BuyWeaponAction and IncreaseMaxHPAction both uses the Single Responsibility Principle as it only has one responsibility which for BuyWeaponAction class is to allow Players to buy weapons and in exchange for souls, whereas IncreaseMaxHPAction allows the actor to increase maximum HP when needed or required.

Class Diagram on Application:



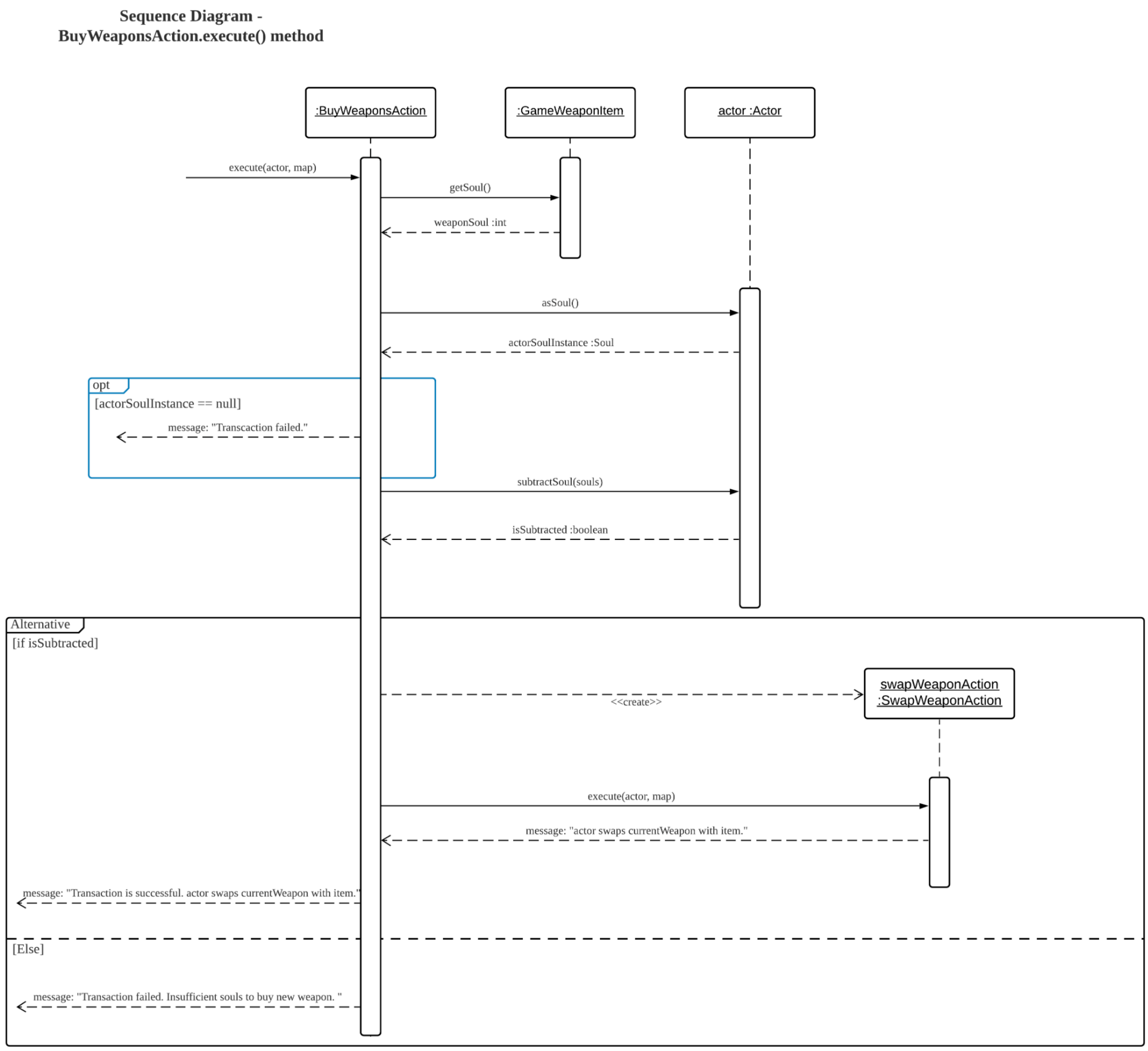
Dependency relationship:

The application class creates the Bonfire, Cemetery, YhormTheGiant, StormRuler, Skeleton, Vendor classes and the Cemetery in its main method. By doing that, these instances will be added into the game map. Thus, it has a dependency relationship with all these classes in order to fulfill its requirements. Next, the dependency relationship between the Cemetery and Undead is explained previously in the UML diagram for terrains.

Optional features

Note: We have completed all the optional requirements given in the assignment.

Sequence Diagram : BuyWeaponAction.execute()



Sequence Diagram : WindSlashAction.execute()

Sequence Diagram on WindSlashAction.execute() method