MA_Lab2Team4
Assignment 3 Amendments to Design Rationale

Name & Student ID :

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

Notes:

1) UML Class Diagrams
    - All the diagrams in **black** represent the UML class diagrams that are provided in the base code.
    - All the diagrams in **blue** and green represent the UML class diagrams for Assignment 1 and Assignment 2.
    - All the diagrams in **purple** represent the UML class diagrams for Assignment 3. They also include amendments to the UML class diagrams for Assignment 1 and Assignment 2.

2) Design rationale:

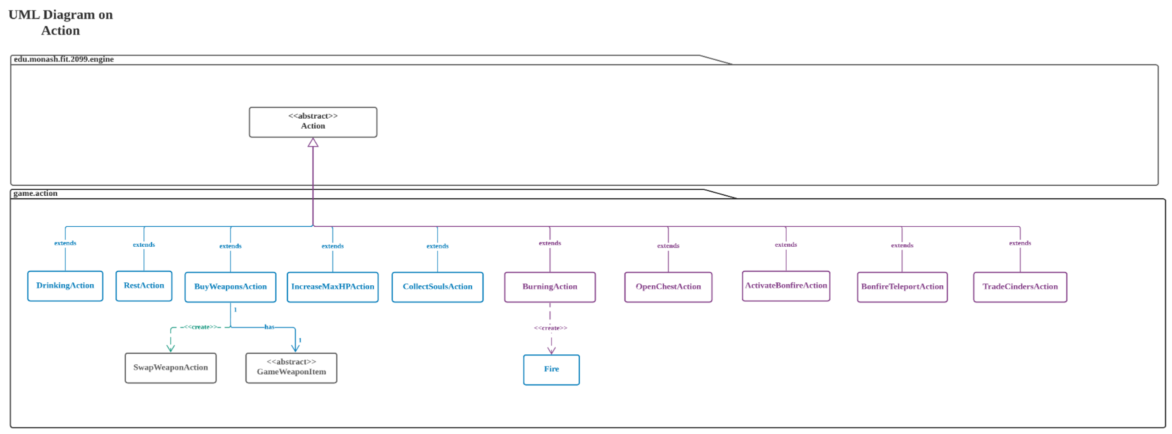
- The words in **black** represent the design rationales for the Assignment 1 and Assignment 2.
- The words in **blue** represent the design rationale for Assignment 3. They also include the amendments to design rationales for the Assignment 1 and Assignment 2.

3) UML Sequence diagram:

    Assignment 1 and Assignment 2:
    - BuyWeaponsAction.execute() method
    - WindSlashAction.execute() method

    Assignment 3 Requirement 4 (Mimic / Chest):
    - OpenChestAction.execute() method

Class diagram on Action (Requirement 1, Requirement 2, Requirement 7, Requirement 8, Assignment 3: Requirement 2, Requirement 3, Requirement 4, Requirement 5)**:**



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.

For Assignment 3, we create new classes such as BurningAction class, OpenChestAction class, ActivateBonfireAction, BonfireTeleportAction and TradeCindersAction class that extend the Action abstract class. This is because, by doing that, these classes can 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 can also inherit and override the execute() method to implement their own requirements and return the results of the action. This prevents us from repeating similar codes in the application, which follows the Don't Repeat Yourself (DRY) principle. As mentioned previously, they also obey the Open-Closed Principle and Single Responsibility Principle.

BuyWeaponsAction class has GameWeaponItem class:

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.

BuyWeaponsAction class <<create>> SwapWeaponAction class:

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.

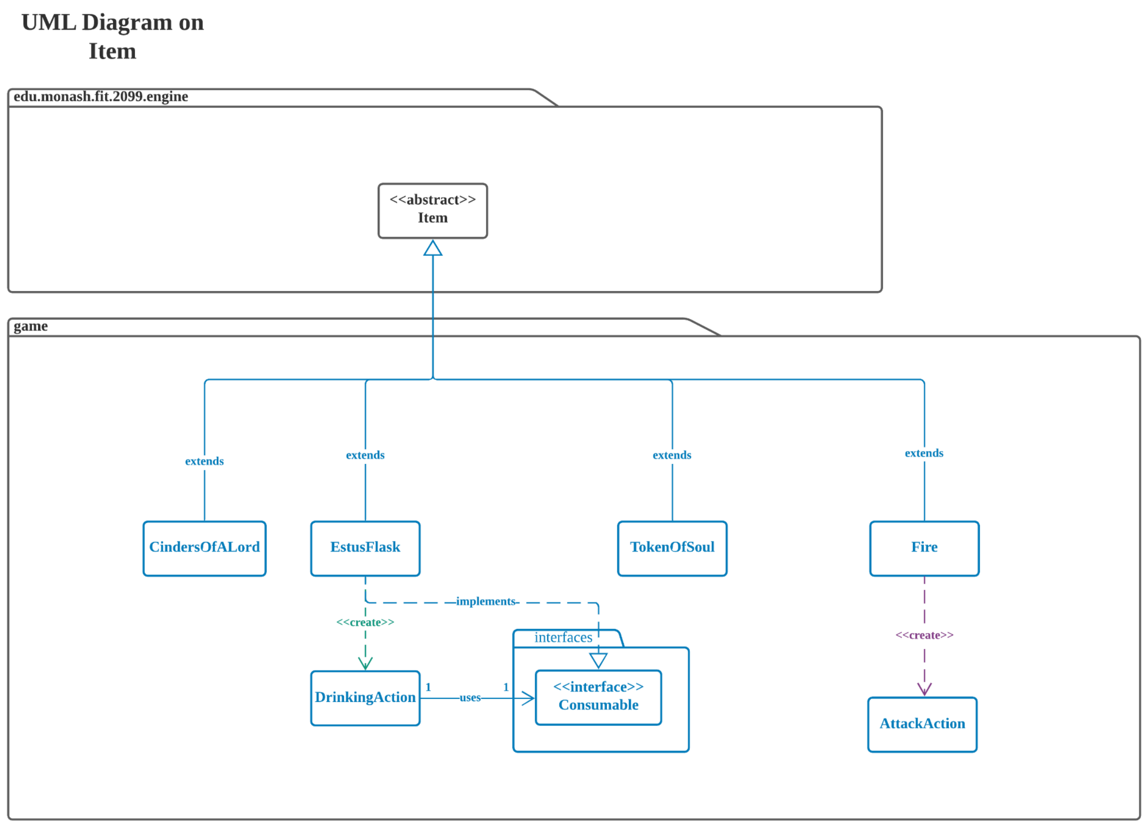
BurningAction class <<create>> Fire class:

In the requirement 5 of Assignment 3, the Player is able to hold the Yhorm's Great Machete after trading the Cinder of Yhorm with the Vendor. The Player can then activate the burning action of the Yhorm's Great Machete. The burning action will burn the ground that can be burnt such as Dirt ground in adjacent squares of the Player. Thus, inside the execute() method of the BurningAction class, we create and place a Fire instance on the ground in adjacent squares of the Player if the ground has the Abilities.CAN_BE_BURNT capability. This shows that the BurningAction class depends on the Fire class to implement burning requirements.Without the

Fire class, the BurningAction class cannot fulfill the requirements completely. Thus, BurningAction class <<create>> Fire class.

By doing that, we are reusing the Fire class we created in Assignment 2.

Class diagram on Item (Requirement 1, Requirement 2, Requirement 4, Requirement 8, Assignment 3: Requirement 5**):**



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, TokenOfSoul class and Fire 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.

In Assignment 2, 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 attributes 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 constructor of the EstusFlask class, we create and add a DrinkingAction instance into the allowableActions list attribute of the EstusFlask. This allows the Player to select the drinking action in the console. 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, the 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.

4) 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.

5) 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.

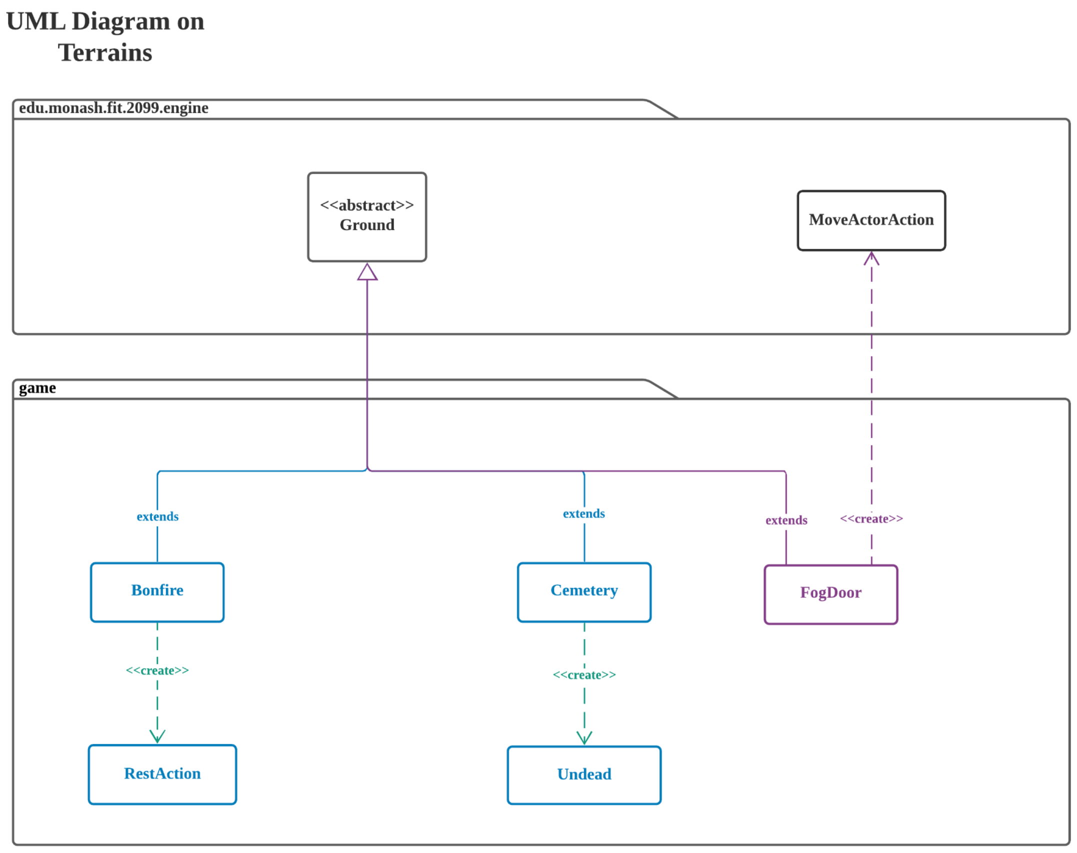
In the future, for any classes that are consumable, we can let these classes implement the Consumable interface. The DrinkingAction class which is responsible for updating the charges of all the consumable classes will just have an association relationship with the Consumable interface, instead of having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

6) Fire class <<create>> AttackAction class

In Assignment 3, when the Yhorm The Giant holding the Yhorm's Great Machete executes the EmberForm behaviour or when the Player is holding the Yhorm's Great Machete, the weapon can add Fire instance on the ground in its adjacent squares which has the can be burnt ability. The Fire can hurt the actor in its surroundings by 25 hit points HP damage. In order to implement the attack and reset the target who is unconscious after the burning, we create an AttackAction instance inside the tick() method of the Fire class. By doing that, we can use the methods we implemented inside the AttackAction class during Assignment 2 to fulfil the requirements. By doing that, we can also see that Fire class depends on AttackAction class to fulfil its requirements. There is a dependency relationship between these 2 classes as shown in the UML diagram above.

<u>Class diagram on Terrains</u> <u>(Requirement 4, Requirement 5, Requirement 8, Assignment 3: Requirement 1):</u>



<u>Inheritance relationship:</u>

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.

In the requirement 1 of Assignment 3, we created a FogDoor class which extends the ground type. The FogDoor is represented by the unique character “=”. It is a sort of portal that can transport the Player from the first game map (Profane Capital) to the second game map (Anor Londo) and vice versa. It overrides the allowableActions() method to implement the action of moving the Player to another map when Player steps onto it.

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.

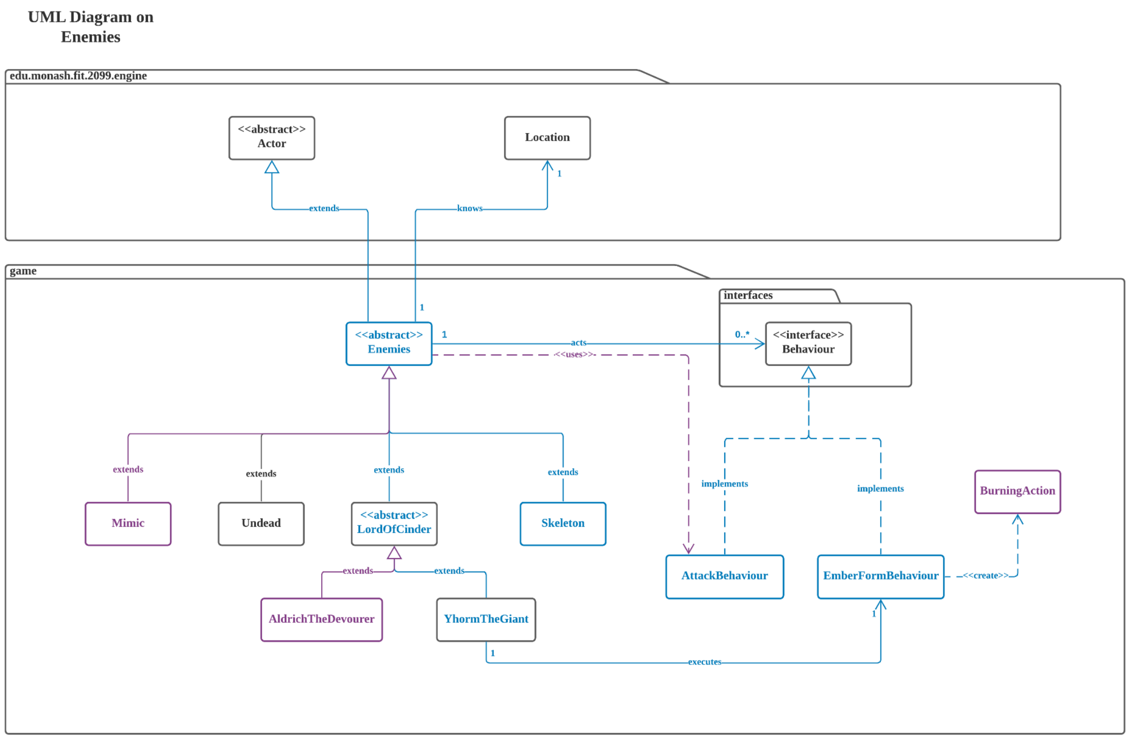
The Bonfire, Cemetery, Valley, Floor, Wall, Dirt and FogDoor classes all obey the Single Responsibility Principle as each class has only a single responsibility to fulfill. Each of these classes only implement a specific requirement. For example, FogDoor is only responsible for transporting the Player between maps.

3) **FogDoor class <<create>> MoveActorAction class:**

As mentioned previously, FogDoor is a sort of portal that can transport only the Player from the first game map (Profane Capital) to the second game map (Anor Londo) and vice versa. Inside the execute() method of the FogDoor class, we will check if the actor standing on the FogDoor is a Player. If yes, we will create and add a MoveActorAction instance into the arraylist if Actions type that will be returned by the method. Thus, the FogDoor class depends on the MoveActorAction class instance to implement its requirement. Therefore, there is a dependency relationship between the FogDoor class and MoveActorAction class such that FogDoor class <<create>> MoveActorAction class.

By doing that, we are reusing the given MoveActorAction class instead of creating a TeleportAction class to implement this requirement in Assignment 3.

Class diagram on Enemies (Requirement 4, Assignment 3: Requirement 3, Requirement 4)**:**



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

While designing the Enemies classes, we also 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.

In the requirement 4 of Assignment 3, the Chest in the map has a 50% chance to be transformed into a Mimic, which is an ambiguous enemy. Thus, we create a Mimic class that extends the abstract Enemies class so that it can inherit all the attributes and methods from the abstract Enemies class. We are reusing the codes in Enemies abstract class that are created in Assignment 2. This design also follows the **Open-Closed Principle and Liskov-Substitution Principle** as explained previously.

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.

In the requirement 3 of Assignment 3, we will have a new Boss, **Aldrich The Devourer**, which is a type of Lord Of Cinder. Thus, we created a new AldrichTheDevourer class which extends the LordOfCinder abstract class so that the new Boss 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 doing that, we obey the Don't Repeat Yourself and Open-Closed Principle as mentioned previously.

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

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.

6) EmberFormBehaviour class <<create>> BurningAction class

In assignment 2, we know that when the Yhorm The Giant which holds the Yhorm's Great Machete activates the EmberForm behaviour, it will burn its surrounding grounds which have the CAN_BE_BURNT capabilities, such as the Dirt ground. Since only Yhorm The Giant has the burning ability, we create and add the Fire instances into surroundings of Yhorm The Giant inside the getAction() method in the EmberForm behaviour class in Assignment 2. Thus, the EmberFormBehaviour <<create>> fire.

However, in assignment 3, the Player can also hold Yhorm's Great Machete weapon after trading the Cinder of Yhorm with Vendor and he can invoke the burning ability anytime in the game. Thus, we decided to move the codes that add fire instances into surroundings from EmberFormBehaviour class to a newly created BurningAction class. Then, the EmberFormBehaviour class can create and return the BurningAction instance in order to implement the burning action. This shows the EmberformBehaviour class depends on BurningAction class to implement its requirement. There is a dependency relation between these 2 classes.

By doing that, we are preventing the redundancy of codes that implement the burning action for Player and Enemies. This adheres to the Don't Repeat Yourself (DRY) principle.

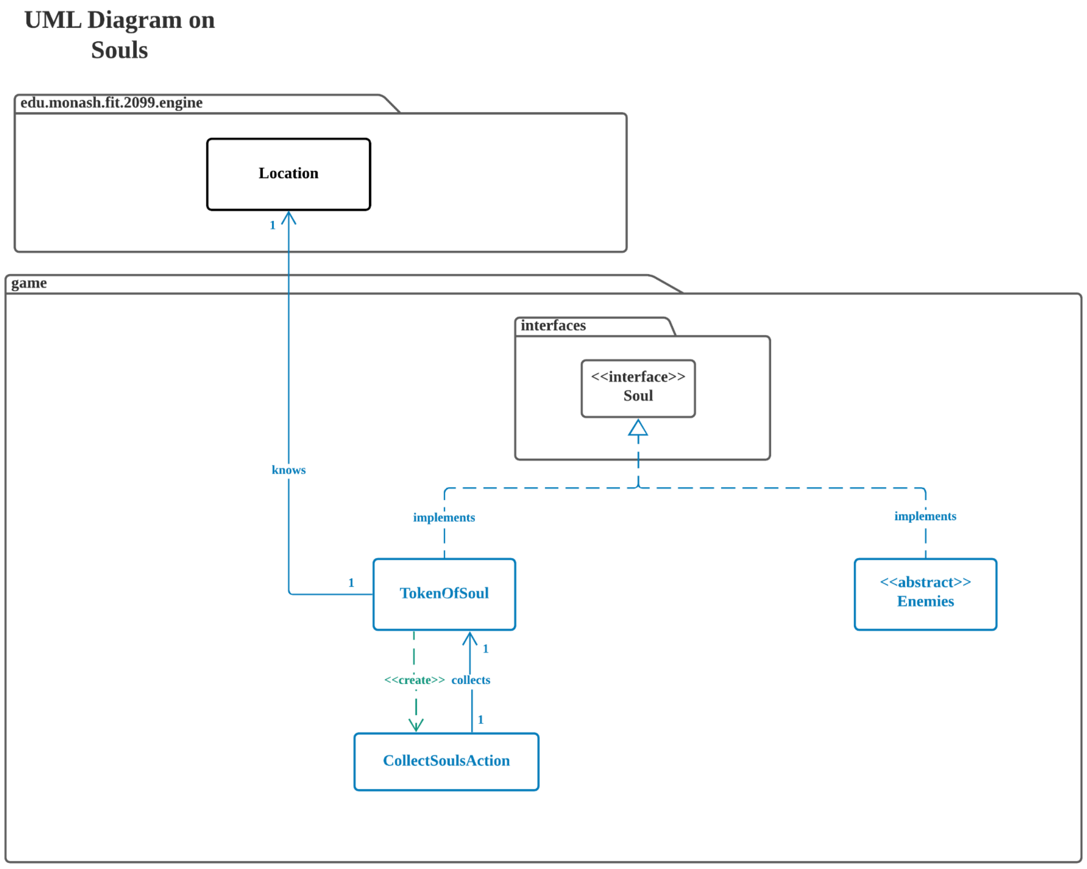
7) **Enemies class knows Location class.**

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.

8) **Enemies class <<use>> AttackBehaviour class.**

In the game, we know that the Enemies can attack the Player when the Player is within its adjacent squares. Thus, we implement the attack action inside the AttackBehaviour class. Inside the Enemies class, we create an AttackBehaviour instance and add it into the behaviours list attribute of the Enemies class. By doing that, all the child enemies that extend the Enemies abstract class can also implement the AttackBehaviour class. From this, we know that the Enemies depend on the AttackBehaviour class to implement the attack Player requirements. Thus, there is a dependency relationship between the Enemies class and AttackBehaviour class such that Enemies class <<use>> AttackBehaviour class.

Class Diagram on Souls (Requirement 1, Requirement 3, Requirement 4, Requirement 8)**:**



1) **TokenOfSoul class and Enemies class implement the Soul interface**

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) **Association and Dependency between TokenOfSoul and CollectSoulsAction**

The TokenOfSoul class 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.

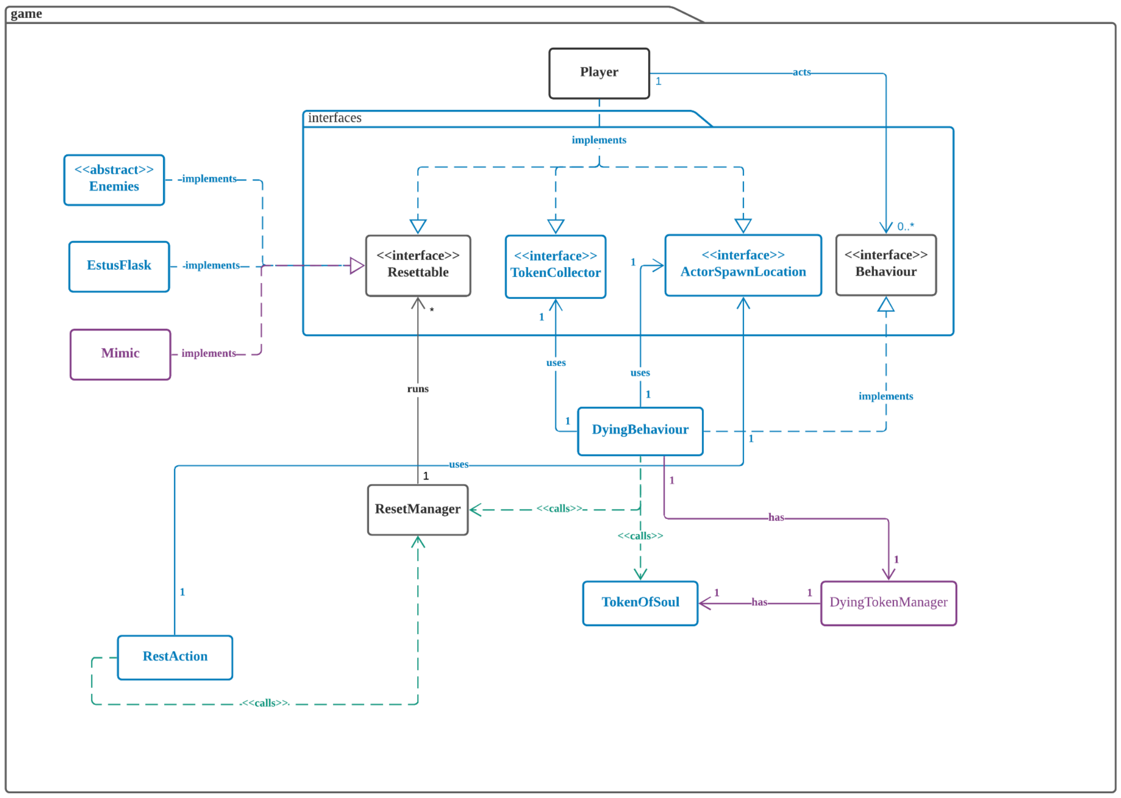
The TokenOfSoul class obeys the Single Responsibility Principle as it has all its required functionality in its one class.

3) <u>TokenOfSoul class knows Location class</u>

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.

<u>Class Diagram on Dying in this game/Soft Reset</u> <u>(Requirement 1, Requirement 2, Requirement 4, Requirement 6, Assignment 3: Requirement 4):</u>

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) <u>Enemies Class implements Resettable interface</u>

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 instance 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) <u>Changes to the ResetManager class</u>

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.

8) <u>Mimic class implements the Resettable interface</u>

The Mimic class implements the Resettable interface as Mimic needs to be reset in the game. For example, when the Mimic is still conscious after fighting with Player during the reset, it can be transformed into the Chest instance and the Chest will be placed in the initial location where Mimic is spawned from it. Thus, It needs to implement the methods in the Resettable interface, such as the resetInstance() method to implement the reset requirements of Mimic.

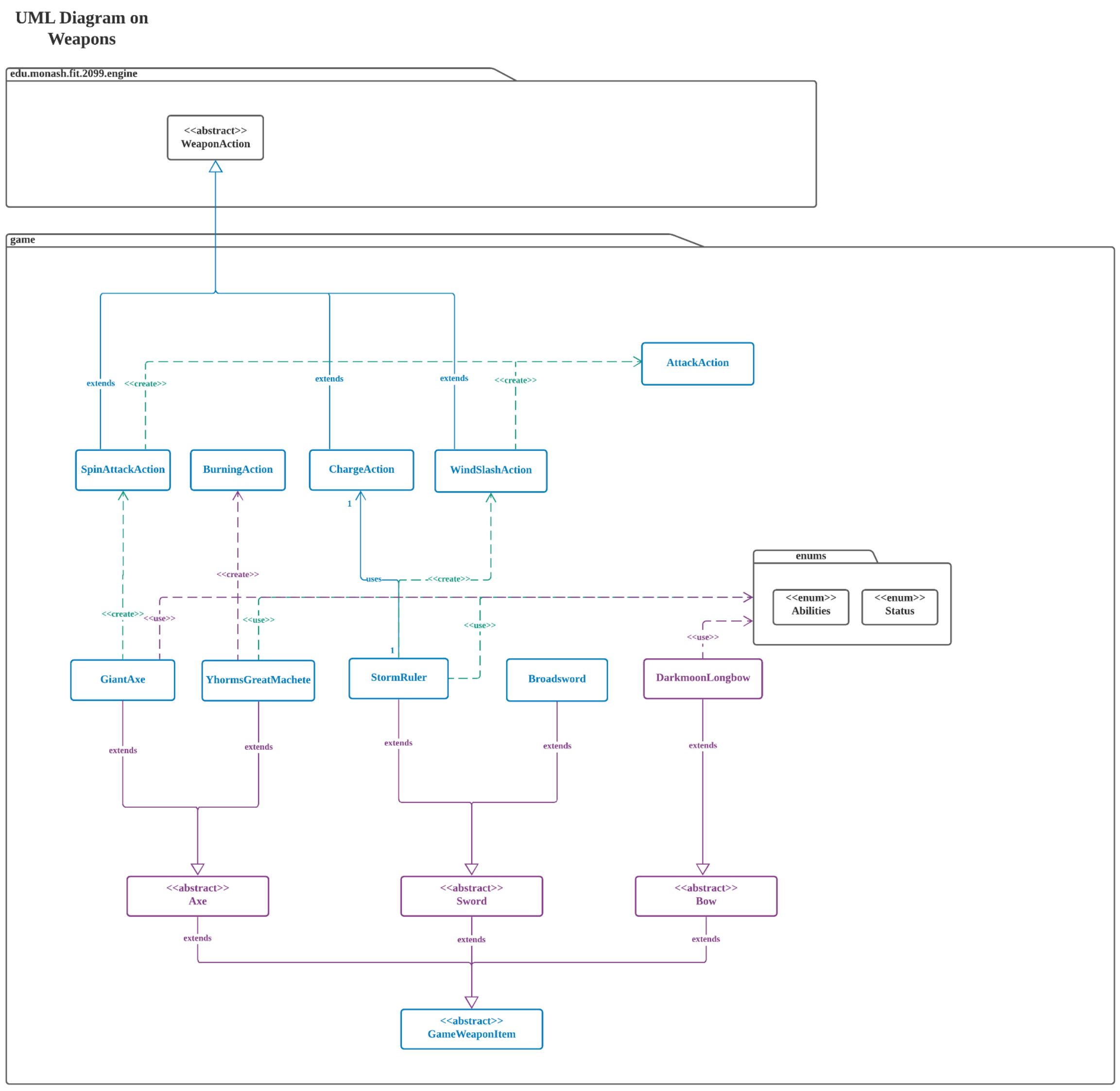
9) <u>Dying Behaviour class has DyingTokenManager</u>

The DyingTokenManager is responsible for managing the TokenOfSoul that is spawned when the player dies. Thus, we create a DyingTokenManager instance attribute inside the DyingBehaviour class to remove the token in the map when the Player is killed in the game before reaching that token. It will then respawn the token with new value in the map.

10) <u>DyingTokenManager has TokenOfSoul</u>

The DyingTokenManager class has a TokenOfSoul injected through setter injection so that it can access the TokenOfSoul instance.

Class diagram on Weapons (Requirement 1, Requirement 4, Requirement 7, Assignment 3: Requirement 3):



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 Assignment 3, we create a DarkmoonLongbow class.

In Assignment 3, we also created an Axe, Sword and Bow abstract class which all extend the GameWeaponItem abstract class because they represent game weapons. We create them as base classes to create more specific weapon child classes. For example, GiantAxe and YhormsGreatMachete are of Axe type, StormRuler and BroadSword are of Sword type and DarkmoonLongbow is of Bow type. Thus, they will extend the appropriate abstract classes that are created as shown in the UML class diagram. By doing that, we can code all the common attributes and methods in the parent abstract class so that they can be inherited by the specific child weapon classes. This prevents the redundancy of codes, which adheres to the Don't Repeat Yourself (DRY) design principle.

Next, the design obeys the **Single responsibility principle** because each Axe, Sword and Bow class contains only the attributes and methods that are needed for their weapon type rather than containing all the attributes and weapons of general weapons. Each of them has a single responsibility. Next, the design also follows the **Open-closed principle,** because we can add new attributes and methods into the specific weapon class without modifying the existing code in other existing classes.

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, GiantAxe, StormRuler and DarkmoonLongbow 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.

For the GiantAxe class, we use the Enum constant such as Status.SPIN_ATTACK_EXECUTED in its damage() method to reduce the damage of the Giant Axe weapon by half when the spin attack action is executed. For the DarkmoonLongbow class, we use the Enum constant such as the Status.HOSTILE_TO_ENEMY and Abilities.IS_ENEMY in its tick() method to detect if the holder and target are player or enemies.

4) GiantAxe <<create>> SpinAttackAction and StormRuler <<create>> WindSlashAction

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) **StormRuler class uses ChargeAction class**

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.

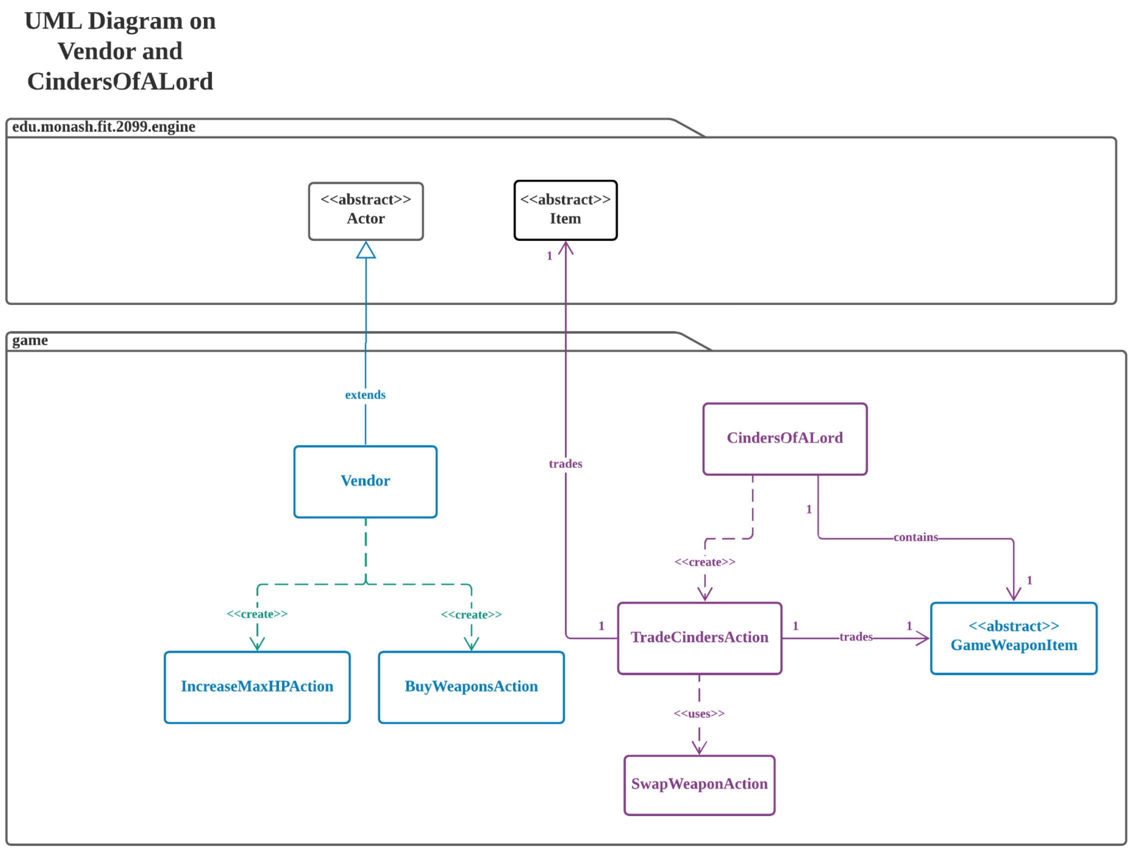
6) **SpinAttackAction class and WindSlashAction class <<create>> AttackAction class**

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.

7) **YhormsGreatMachete class <<create>> BurningAction class:**

YhormsGreatMachete has the burning ability that will burn the surrounding grounds which have the CAN_BE_BURNT capability . Thus, we will create and add the BurningAction instance into the allowableActions attribute in the YhormsGreatMachete class constructor. By doing that, when the Player is equipped with this weapon, they can invoke the burning action. Since we are creating the BurningAction instance inside the YhormsGreatMachete class, we can see that YhormsGreatMachete class depends on the BurningAction class to implement its requirement. There is a dependency between these two classes where YhormsGreatMachete class <<create>> BurningAction class.

Class diagram on Vendor and CindersOfALord UML (Requirement 8, Assignment 3: Requirement 5):



1) **Vendor class extends Actor class**

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.

2) Vendor <<create>> IncreaseMaxHPAction, BuyWeaponsAction and TradeCindersAction

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 BuyWeaponsAction class which enable the player to select these actions in the console to buy new weapons and 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 inside the method. Then, we can create and add , IncreaseMaxHPAction instance and the BuyWeaponsAction as parameters into the constructor of the Actions instance. By doing that, the IncreaseMaxHPAction instance and BuyWeaponsAction instance 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 the BuyWeaponsAction and IncreaseMaxHPAction instance in the allowableActions() method in the Vendor class, it shows that the Vendor class needs and depends on these classes in order to fulfill its requirements. Thus, we create a dependency relationship between Vendor class with both BuyWeaponAction class, IncreaseMaxHPAction and TradeCinderAction class where Vendor class <<create>> BuyWeaponAction class and IncreaseMaxHPAction class.

The BuyWeaponAction and IncreaseMaxHPAction all follow the Single Responsibility Principle as each of them has only one responsibility. The BuyWeaponAction class allows Players to only buy weapons by trading his souls. The IncreaseMaxHPAction only allows the actor to increase maximum HP by trading his souls.

3) CindersOfALord class <<create>> TradeCindersAction class

In the requirement 5 of Assignment 3, the Player can trade the CindersOfALord item with the corresponding weapon of the enemy. Thus, inside the tick() method of the CindersOfALord class, we will check if the Vendor is at the adjacent squares of the actor who holds the CindersOfALord, such as the Player. If yes, we will create and add a TradeCindersAction instance into its allowableActions list attribute. By doing that, the Player can trade the cinder with the corresponding weapon of the enemy. The CindersOfALord class depends on the

TradeCindersAction class to implement its requirements. Thus, there is a dependency relationship between CindersOfALord class and TradeCindersAction class such as CindersOfALord class <<create>> TradeCindersAction class.

4) **CindersOfALord class contains GameWeaponItem class:**

In the requirement 5 of Assignment 3, the Player can trade the CindersOfALord item with the corresponding weapon of the enemy. Thus, the CindersOfALord class should store a GameWeaponItem instance attribute that references the specific weapon that is held by the Lord. One CindersOfALord can be traded with only one GameWeaponItem because each Lord or Boss is now holding only one weapon. Thus, there is a one-to-one association relationship between the CindersOfALord class and GameWeaponItem class.

5) **TradeCindersAction class trades GameWeaponItem class and Item class.**

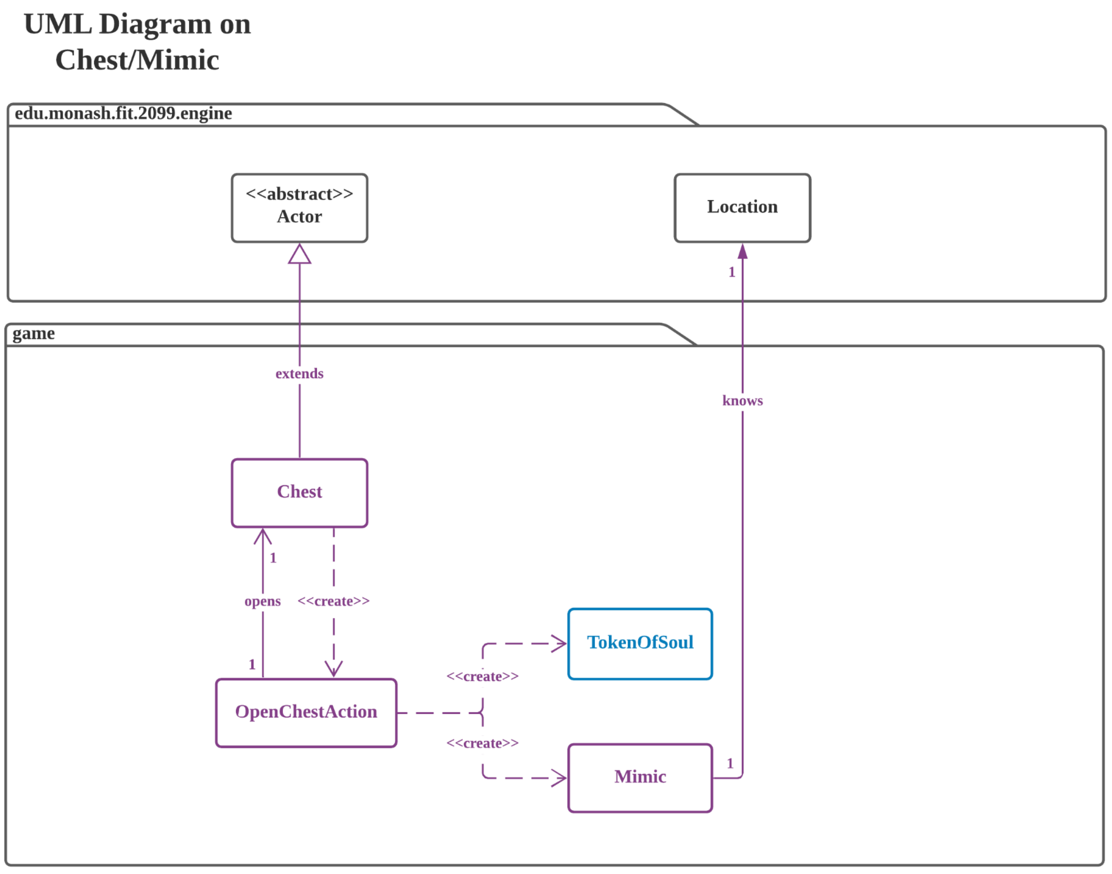
In the requirement 5 of Assignment 3, the Player can trade the CindersOfALord item with the corresponding weapon of the enemy. Thus, we create an Item instance attribute in the TradeCindersAction class which references the cinder that should be traded and a GameWeaponItem class which references the weapon that should be traded. Since we are creating the Item instance and GameWeaponItem instance inside the TradeCindersAction class, it shows an association one-to-one relationship between the TradeCindersAction class with each of the GameWeaponItem class and Item class.

6) **TradeCindersAction class <<create>> SwapWeaponAction class.**

In the requirement 5 of Assignment 3, the Player can trade the CindersOfALord item with the corresponding weapon of the enemy. When the trading action is done, the Player will receive a new weapon. However, in Assignment 1 and 2, we know that Player can only hold one weapon at a time. Thus, inside the execute method() of the TradeCindersAction class, we create a SwapWeaponAction instance. The method will then invoke the execute method of the SwapWeaponAction instance to replace the old weapon in Player's inventory list with the new traded weapon. The TradeCindersAction class depends on the SwapWeaponsAction class to implement its requirement. Thus, there is a dependency relationship between the TradeCindersAction class and SwapWeaponAction class such as TradeCindersAction class <<create>> SwapWeaponAction class.

To sum up, we do not create child classes for the CindersOfALord to represent Cinder for different bosses. This is because the implementation can be sufficiently done by the CindersOfALord which is created in Assignment 1 and 2.

Class Diagram on Chest/Mimic (Assignment 3: Requirement 4):



1) Why Chest class extends Actor class:

This is because we would like the Player to interact with or open the Chest when the Player is within its adjacent squares. If the Chest class extends the Item class, the Player can only interact with or open the Chest when he is exactly on top of the Chest Item. In order to improve the game design and experience, we choose to extend the Chest class to Actor class.

However, the cons of the design is that the design may be confusing because the Chest should be an Item and should extend the Item abstract class.

2) Chest class <<create>> OpenChestAction class:

In the requirement 4 of Assignment 3, the Player can open the Chest placed in the game map. Thus, inside the getAllowableActions method of the Chest class, we create an OpenChestAction instance and we add it into the Actions instance in the method. By doing that, the Player can invoke the open chest action when he is within the adjacent squares of the Chest. Thus, we can see that the Chest depends on the OpenChestAction to fulfill its requirements. Therefore, the Chest class has a dependency relationship with the OpenChestAction class such that Chest class <<create>> OpenChestAction.

3) **<u>OpenChestAction class opens Chest class:</u>**

In the requirement 4 of Assignment 3, the Player is able to open the chest placed in the map. There is an equal chance that the Chest will be replaced by a mimic or token of souls when it is opened by the Player. Thus, inside the execute method() of the OpenChestAction class, we will remove the Chest instance and replace it with a newly created Mimic or TokenOfSoul instance. In order for us to remove the Chest, we have to create an attribute of Chest instance which will reference the Chest instance to be removed from the map. Thus, there is an association between the OpenChestAction class and Chest class such that the OpenChestAction class opens Chest class.

4) **<u>OpenChestAction class <<create>> Mimic class and TokenOfSoul class:</u>**

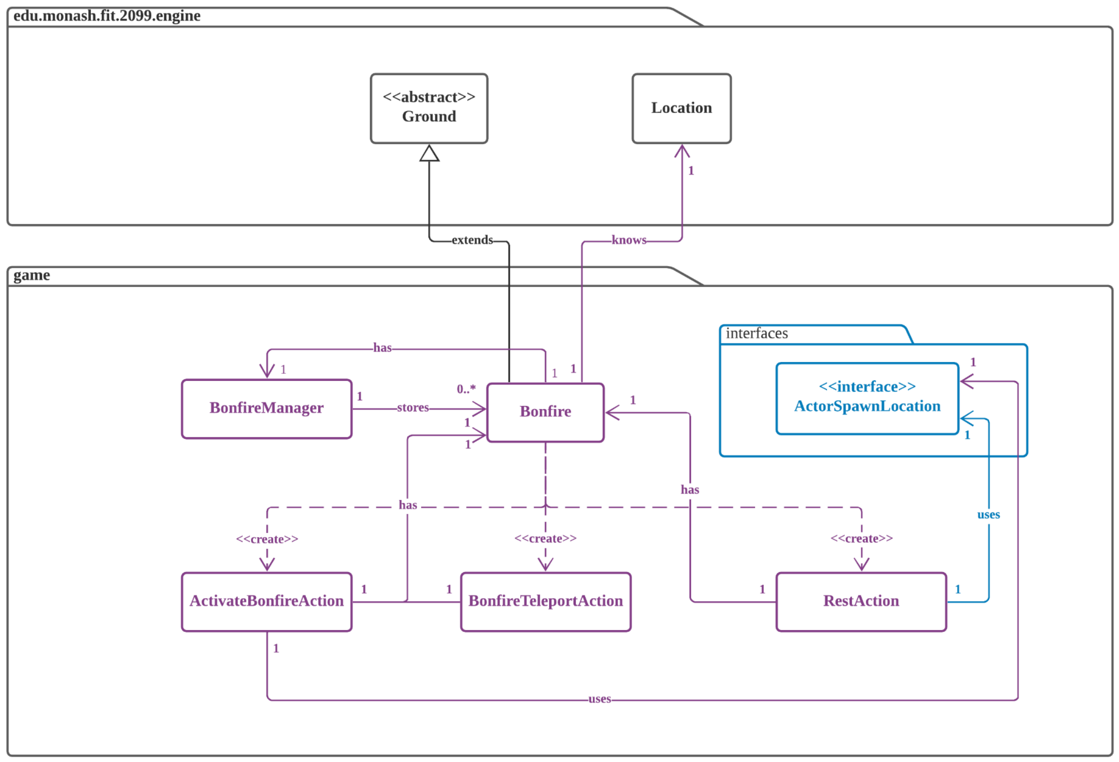
As mentioned previously, there is an equal chance that the Chest will turn into a mimic or token of souls when it is opened by the Player. Thus, inside the execute() method of the OpenChestAction class, we create a Mimic instance and a TokenOfSoul instance so that the Chest can be replaced by one of these items when it is removed. This shows that the OpenChestAction class depends on both the Mimic class and TokenOfSoul class to implement its requirement. Without these classes, the OpenChestAction class cannot fulfill the requirements completely. Thus, OpenChestAction class <<create>> Mimic class and TokenOfSoul class.

5) **<u>Mimic class knows Location class:</u>**

In the requirement 4 of Assignment 3, when the Mimic is still conscious during the reset, the Mimic will be transformed into Chest and placed in the original position where it is spawned. Thus, we create a Location instance attribute which references the initial location where Mimic is transformed from Chest. By doing that, we can create and place the Chest back at that initial position after reset. Thus, there is an association relationship between the Mimic class and Location class, where Mimic class knows the Location class.

<u>Class Diagram on Bonfire (Assignment 3: Requirement 2):</u>

UML Diagram on Bonfire



1) **<u>Bonfire class <<create>> ActivateBonfireAction class, BonfireTeleportAction class and RestAction class</u>**

The Bonfire class has a dependency with the ActivateBonfireAction, BonfireTeleportAction and RestAction, as these classes are required for the Bonfire to perform its actions, such as the rest action, the bonfire teleport action and the action to allow bonfire to be activated. Thus, we create the instances of these classes in the Bonfire class. The Bonfire classes will then use the attributes from these classes in its execute() method later on to perform the action.

2) **<u>ActivateBonfireAction class, BonfireTeleportAction class and RestAction class has Bonfire class</u>**

The ActivateBonfireAction class, BonfireTeleportAction class and RestAction class each have an association relationship with the Bonfire class because all three of them need to interact with a target Bonfire. Thus, we will create a Bonfire type attribute named targetBonfire which references the Bonfire instance. The target Bonfire is provided through constructor injection so that the client can access the Bonfire and its methods.

3) **Bonfire class has a Location class.**

Each of the Bonfires knows its own location so that we can easily retrieve its location when needed through a getter method. Thus, we create a bonfireLocation attribute which is of Location type that references the location where Bonfire is located in the map. This is originally implemented in Assignment 2 and it is re-used in Assignment 3.

4) **ActivateBonfireAction class, RestAction class uses ActorSpawnLocation interface.**

ActivateBonfireAction class, RestAction class stores an attribute of ActorSpawnLocation interface to spawn the actor at the latest Bonfire he interacts during reset. These classes will use methods that are declared in the ActorSpawnLocation interface, such as the setLatestSpawnPoint() method to spawn the actor at the latest Bonfire he interacts during reset. Thus, there will be an association relationship between the ActivateBonfireAction class and RestAction class with the ActorSpawnLocation interface.

This satisfies the Dependency Inversion Principle where these classes depend on abstractions instead, which is the interface. This ActorSpawnLocation interface was originally implemented in Assignment 2 and it is now re-used in Assignment 3.

5) **Bonfire class has a BonfireManager class and BonfireManager class stores the Bonfire class**

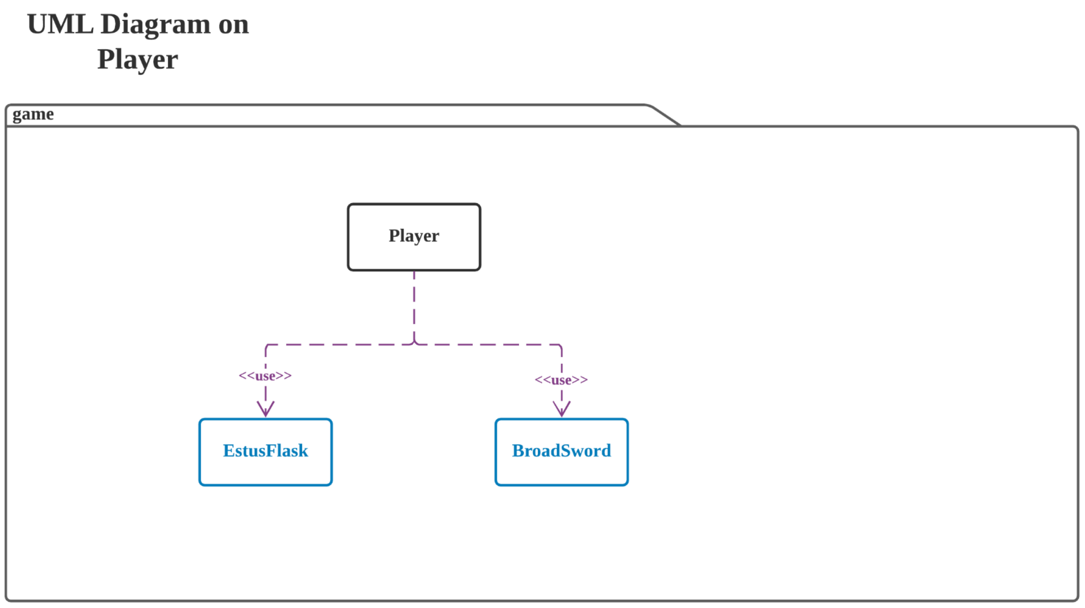
The Bonfire class has an association relationship with the BonfireManager class and the BonfireManager class stores the Bonfire class. The relationship for the Bonfire class to the BonfireManager class is a one-to-one relationship and the BonfireManager class has a one-to-many relationship with the Bonfire class as the BonfireManager is designed to store more bonfires, in an event that in the future there are more maps introduces and more bonfires would be required in those maps

6) **Application class <<create>> BonfireManager class and injected into Bonfire class**

The BonfireManager instance is created in the application class and injected into each of the Bonfire through dependency injection, or more specifically, constructor injection. This is an improvement made after we learnt about dependency injection as we initially made

BonfireManager a singleton class. As a result, this made the code more reusable, testable and maintainable.

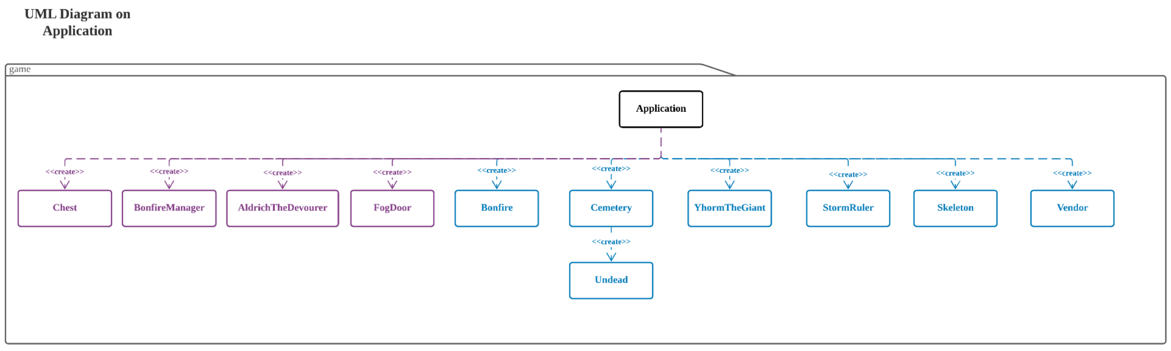
Class Diagram on Player :



Player class <<use>> EstusFlask class and Broadsword class

In Assignment 2, we know that the Player will initially hold and can use the EstusFlask instance and BroadSword instance in its inventory list attribute. Thus, we create and add an EstusFlask instance and BroadSword instance into the inventory list attribute inside the constructor of the Player class. The Player class depends on the EstusFlask class and BroadSword class to implement its requirements. Thus, there is a dependency relationship between the Player class with both Estus Flask and BroadSword class such as Player class <<use>> EstusFlask class and Broadsword class.

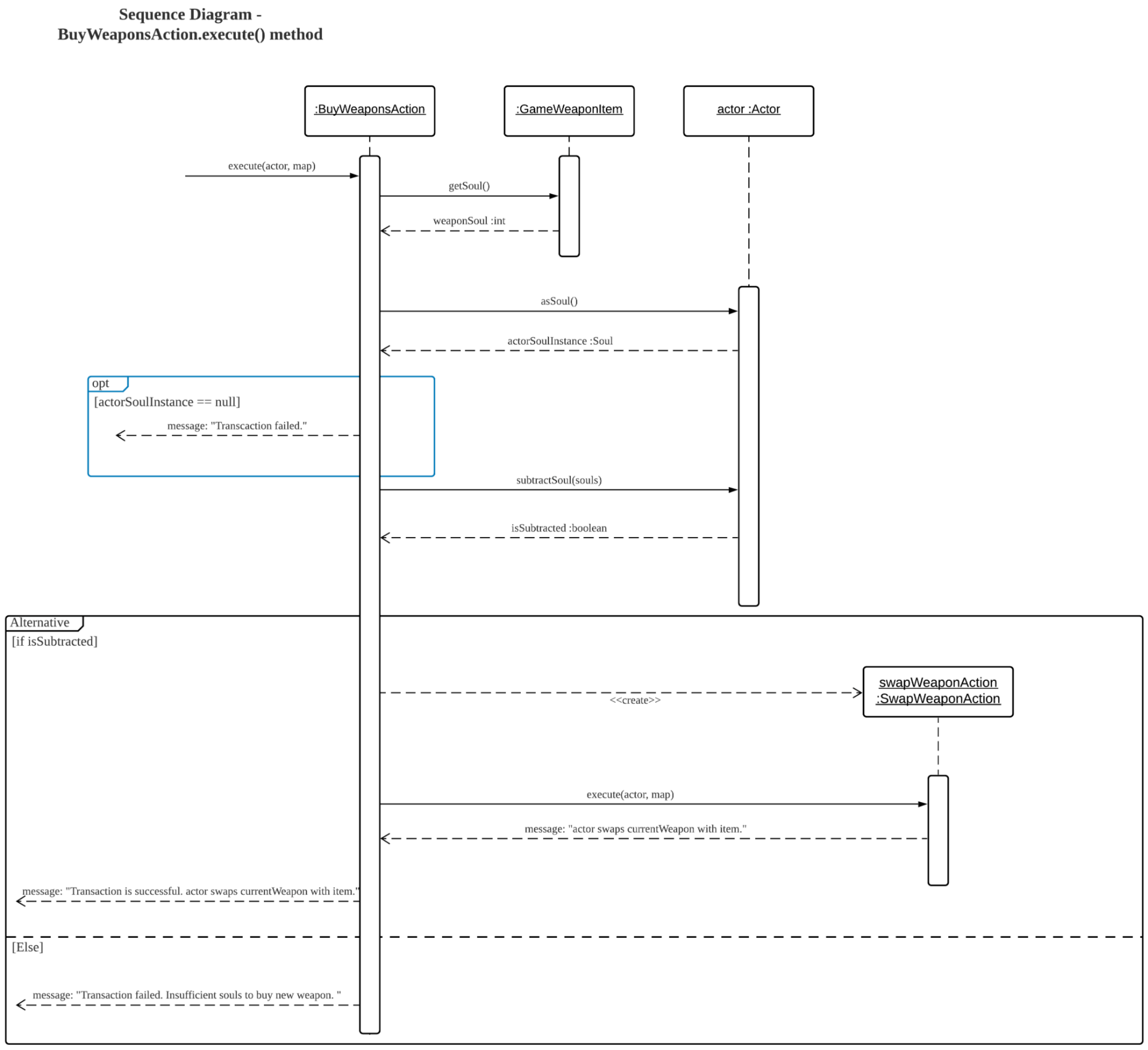
Class Diagram on Application:



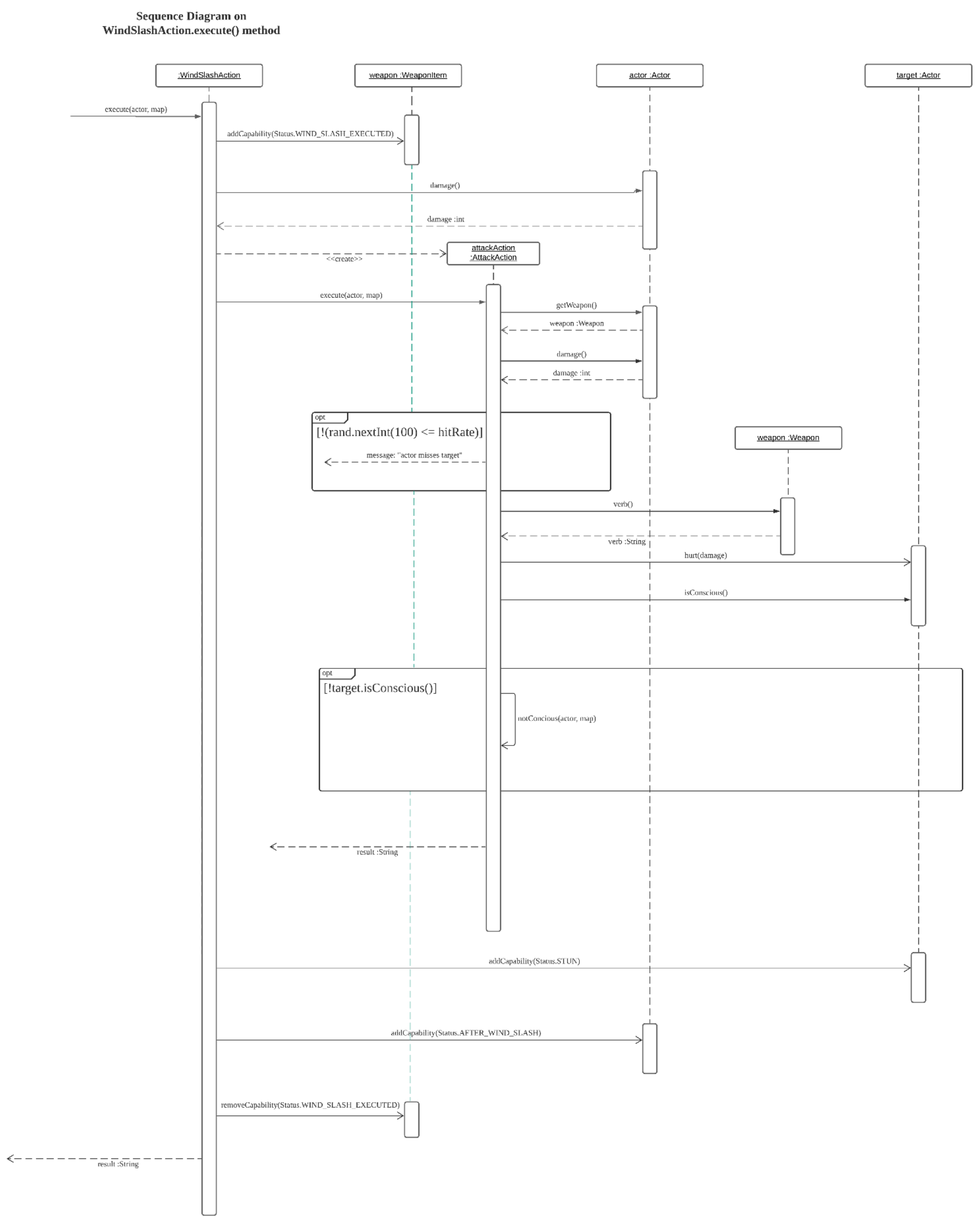
Dependency relationship:

The application class creates the Bonfire, Cemetery, YhormTheGiant, StormRuler, Skeleton in its main method. **For Assignment 3, it also creates Chest, FogDoor, BonfireManager, and AldrichTheDevourer.** 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.

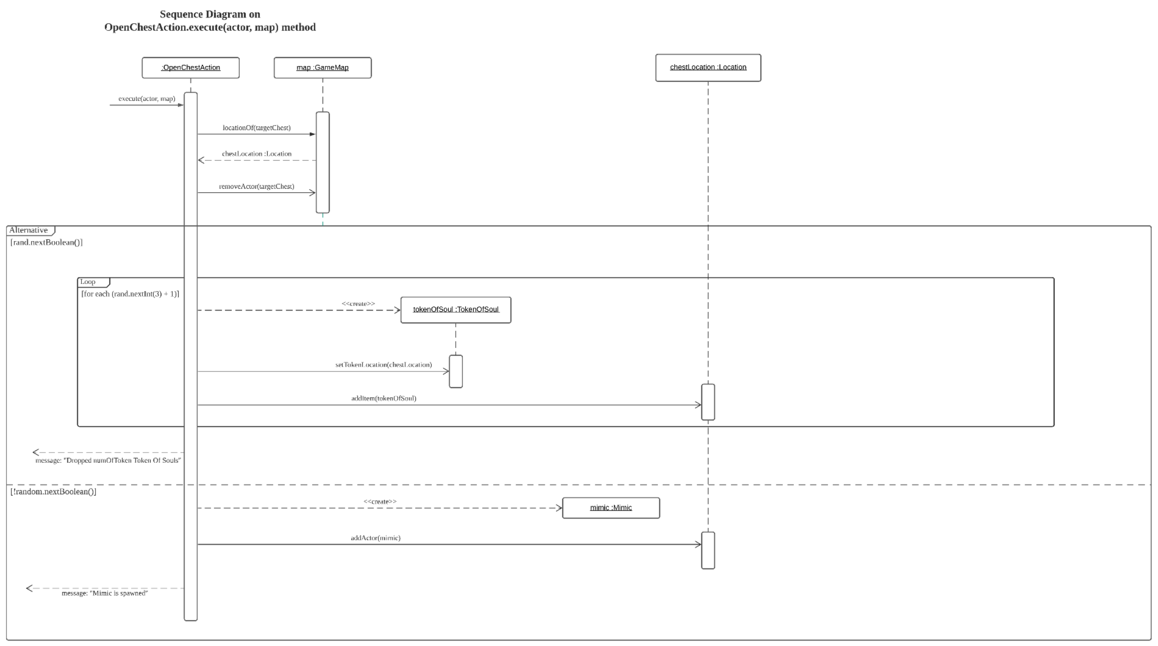
Sequence Diagram : BuyWeaponsAction.execute()


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 Diagram : WindSlashAction.execute()



Sequence Diagram on Assignment 3 Requirement 4: OpenChestAction.execute()



Design rationale for the OpenChestAction.execute(actor, map)

When the execute() method of the OpenChestAction instance is called, meaning the chest is opened, the method will invoke the locationOf() method on the input map parameter to find the location of the Chest instance referenced by the targetChest attribute in the OpenChestAction class. The method will then assign the returned location to a variable named targetLocation. We need to obtain the location of the targetChest instance in order to remove it from the map and replace it with either a Mimic or tokenOfSoul instance. After that, the method will then invoke the removeActor() method on the input map parameter to remove the targetChest instance from the map.

After removing the targetChest, there is a 50% chance that the targetChest will be replaced by a TokenOfSoul item instance or a Mimic enemy instance. Thus, we use a random generator to generate a boolean value, rand.nextBoolean() which determines the outcome of the opened chest. We decide to use rand.nextBoolean() instead of rand.nextInt(2) because there are only 2 outcomes when the chest is opened. However, the con is that this method will not work if the outcome of the opened chest is more than 2.

Then, if the returned boolean value is true, the method will create several TokenOfSoul instances which number ranges from 1 to 3 inclusively. We use the random number generator to generate the random number of TokenOfSoul instances. Each instance takes in the integer 100 as its parameter because each of them is worth 100 souls. Then, we will reuse the chestLocation obtained previously to add these tokens onto the correct location in the map. The method will invoke the addItem() method on the chestLocation to add these tokenOfSoul instances onto the map location. The method will then return a string describing the exact number of TokenOfSoul instances that have dropped.

Otherwise, if the returned boolean value is false, the method will create a Mimic instance. Similarly, we will reuse the chestLocation obtained previously to add the Mimic instance onto the correct location in the map. The method will invoke the addItem() method on the chestLocation to add these Mimic instances onto the map location. The method will then return a string describing that a Mimic is spawned in the map.