

FIT2099

Assignment 3

Team 2

Tan Hong Yi

Afrida Jahin

Lee Jia Yi

17th October, 2021

Table of Contents

Table of Contents	1
Design Rationale	2
Class Structure in the Game Package	3
Requirement 1: Player and Estus Flask	5
First changes in Requirement 1: Player and Estus Flask	6
Second changes in Requirement 1: Player and Estus Flask	7
Requirement 2: Bonfire	8
First changes in Requirement 2: Bonfire	9
Second changes in Requirement 2: Bonfire	11
Requirement 3: Souls	12
First changes in Requirement 3: Souls	12
Requirement 4: Enemies	13
First changes in Requirement 4: Enemies	13
Second changes in Requirement 4: Enemies	15
Requirement 5: Terrains - Valley and Cemetery	16
First changes in Requirement 5: Terrains	16
Requirement 6: Soft reset	17
First changes in Requirement 6: Soft Reset	18
Requirement 7: Weapons	19
First changes in Requirement 7: Weapons	20
Requirement 8: Vendor	21
First changes in Requirement 8: Vendor	22
Second changes in Requirement 8: Vendor	22
New Requirements (Assignment 3)	23
Requirement 1: New Map and Fog Door	23
Requirement 2: Updated Bonfire	24
Requirement 3: Aldrich the Devourer (Lord of Cinder)	25
Requirement 4: Mimic / Chest	26
Requirement 5: Trade Cinder of Lord	27
Interaction Diagram	28
Interaction Diagram 1: Drink Action	28
Changes in Interaction Diagram 1: Drink Action	28
Interaction Diagram 2: Buying Item at Vendor	32
Changes in Interaction Diagram 2: Buying Item at Vendor	32
Interaction Diagram 3: Actions allowed when interacting with Bonfire	35

Design Rationale

The design rationale will explain and justify the reasons for the choice of design for the Design O'Souls game. The rationale and the class diagram will clearly show the new classes created in order to satisfy the requirements given and also how the new classes relate to and interact with the other classes in the existing system to deliver the required functionality. This design rationale will also include the roles and responsibilities of any new or modified classes, as well as the pros and cons of the proposed system.

The classes and relationship lines in **blue** are the newly designed class diagrams, otherwise they are originally designed class diagrams. The **first changes** for the class diagram represents the changes for the **second** assignment while the **second changes** are for the **third assignment**.

The requirements(Assignment 2) changed are as followed:

- Requirement 1: Player and Estus Flask
- Requirement 2: Bonfire
- Requirement 3: Souls
- Requirement 4: Enemies
- Requirement 5: Terrains - Valley and Cemetery
- Requirement 6: Soft reset
- Requirement 7: Weapons
- Requirement 8: Vendor

Additional requirements for the third assignment are as followed:

- Requirement 1: New Map and Fog Door
- Requirement 2: Updated Bonfire
- Requirement 3: Aldrich the Devourer (Lord of Cinder)
- Requirement 4: Mimic / Chest
- Requirement 5: Trade Cinder of Lord

Class Structure in the Game Package

The given and new classes created which are located in the game package were initially not grouped and packaged accordingly.

The classes in the game package are repackaged and is shown as followed:

Table 1: Structure of Classes in the Game Package

Package	Class
Behaviour	<ul style="list-style-type: none">- AttackBehaviour- FollowBehaviour- WanderBehaviour- RangedWeaponBehaviour
Enemy	<ul style="list-style-type: none">- AldrichTheDevourer- CinderOfTheLord- Enemy- LordOfCinder- Mimic- Skeleton- Undead- YhormTheGiant
Enums	<ul style="list-style-type: none">- Abilities- Status
Ground	<ul style="list-style-type: none">- Bonfire- BurningDirt- Cemetery- Chest- Dirt- FireKeeper- Floor- Fogdoor- Valley- Wall
Interfaces	<ul style="list-style-type: none">- Behaviour- Consumable- Resettable- Soul
Player	<ul style="list-style-type: none">- ActivateBonfireAction- AttackAction

		<ul style="list-style-type: none"> - DeathAction - DrinkAction - EstusFlask - OpenChestAction - Player
Reset		<ul style="list-style-type: none"> - ResetAction - ResetManager - RetrieveSoulsAction - TokenOfSouls
Vendor		<ul style="list-style-type: none"> - BuyAxeAction - BuyBowAction - BuyMacheteAction - BuySwordAction
Weapon		<ul style="list-style-type: none"> - Broadsword - DarkmoonLongbow - GameWeaponItem - GiantAxe - StormRuler - YhormsGreatMachete
Weapon	Action	<ul style="list-style-type: none"> - ChargeAction - BurnGroundAction - SpinAttackAction - SwapAttackAction - WindSlashAction
<ul style="list-style-type: none"> - Application - BonfireManager 		

Requirement 1: Player and Estus Flask

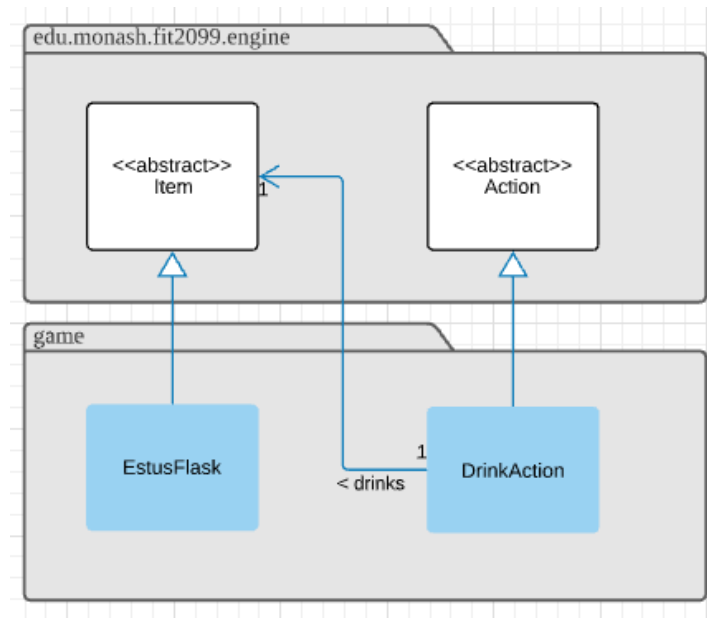


Figure 1A: Old Class Diagram for Player and Estus Flask

The Player class extends the Actor class, which has attribute “hitpoints” to keep track of a player’s health. In order for the player’s health to be shown on the console, a display statement is added to the `showMenu()` method in the Menu class.

As the console prints out all the actions available to the player, a `DrinkAction` class is added so that the player can consume the drink whenever it is available. The reason the class is named `DrinkAction` instead of `DrinkEstusFlaskAction` is because other types of drinks may be introduced in the future. For example, buff drinks which can be used to increase stats temporarily. Therefore, `DrinkAction` will be able to keep track of which item the player is drinking and the description of the drink action can also be changed in `menuDescription()`. An `EstusFlask` class which extends the `Item` abstract class is also implemented for the player to keep track of the number of Estus Flask (by using an attribute called `numberOfEstusFlask`) available by accessing the inventory (list of items).

Another option to track the number of Estus Flask available is by creating multiple `EstusFlask` objects in the inventory, but this method will not abide by the design principle ‘**Classes should be responsible for their own properties**’. Hence, our final decision would be the first method, where an attribute is used to keep track of the chargers.

First changes in Requirement 1: Player and Estus Flask

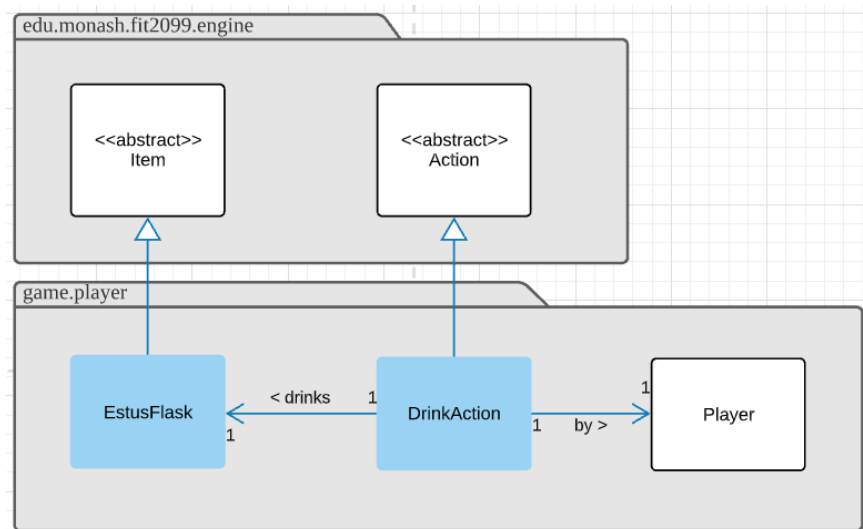


Figure 1B: First Changes Class Diagram for Player and Estus Flask

The drink action is added to the player's list of actions, where the Estus Flask can be drunk when the hotkey, 'a' is inputted in the console.

Second changes in Requirement 1: Player and Estus Flask

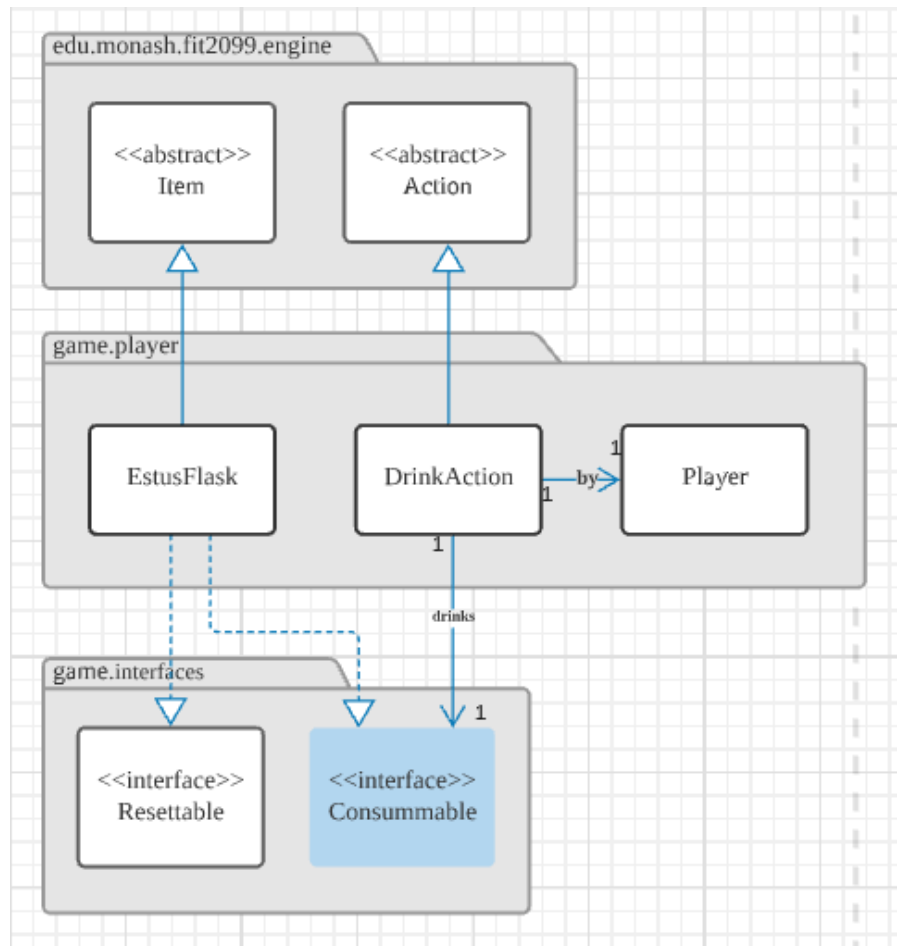


Figure 1C: Second Changes Class Diagram for Player and Estus Flask

The drink action is now generalised to other consumables as well and not focusing only on the Estus Flask. The Estus Flask class is refactored to implement the Resettable and Consumable class in order for the reset and drink action to take place. The drink action is also added to the player's list of actions using the `getAllowableActions` method in the Estus Flask class. This way, **DIP** can be applied so that it makes the system easier to extend later.

Requirement 2: Bonfire

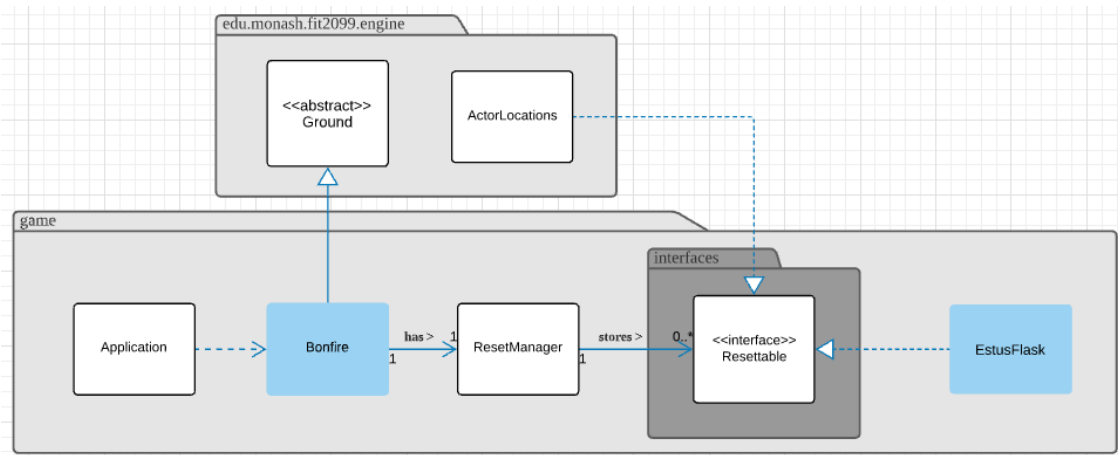


Figure 2A: Old Class Diagram for Bonfire

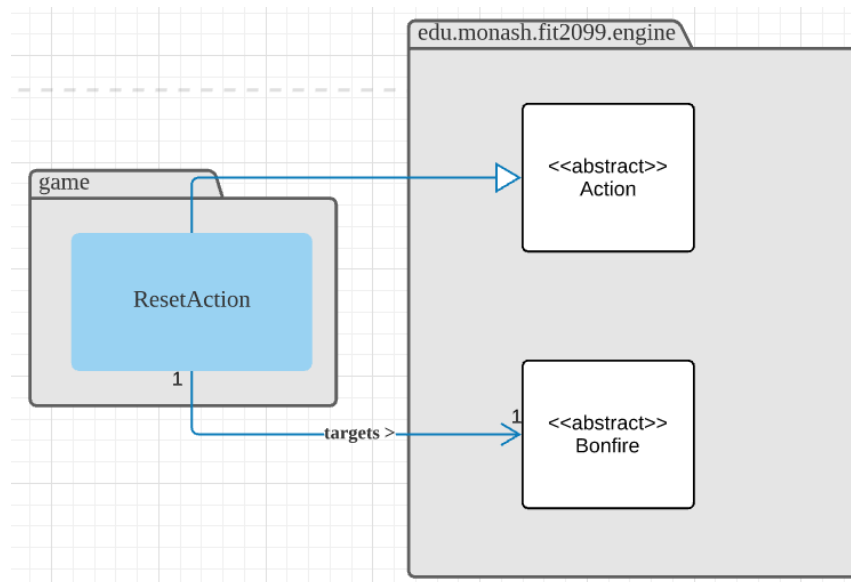


Figure 3A: Old Class Diagram to Reset in Bonfire

A new Bonfire class which extends the Ground abstract class is created so that the Bonfire class can be displayed on the console as 'B' and the player can also interact with the bonfire to refill the player's hitpoints to the max hitpoints, refill the number of Estus Flask back to 3, and reset enemies' position, health, and skills. In order for the bonfire to have that reset functionality, an Action class called ResetAction will be introduced so that the player knows ResetAction can be performed on Bonfire. Besides, the Bonfire class will have ResetManager as its attribute to

reset resettable instances such as Actors (can be accessed in ActorLocations via Actor Iterator), ActorLocations, and EstusFlask. The advantage for both the ActorLocations class and the EstusFlask class of implementing the Resettable interface is that it would be able to reduce dependencies (**ReD design principle**) between the ResetManager class and the ActorLocations class, as well as between the ResetManager class and the EstusFlask class. Besides, it also satisfies the design principle '**Classes should be responsible for their own properties**', where both the ActorLocations class and the EstusFlask class are capable of resetting their own properties. Furthermore, all different class instances can be categorised as resettable instances and, therefore, similar instances can be stored together.

First changes in Requirement 2: Bonfire

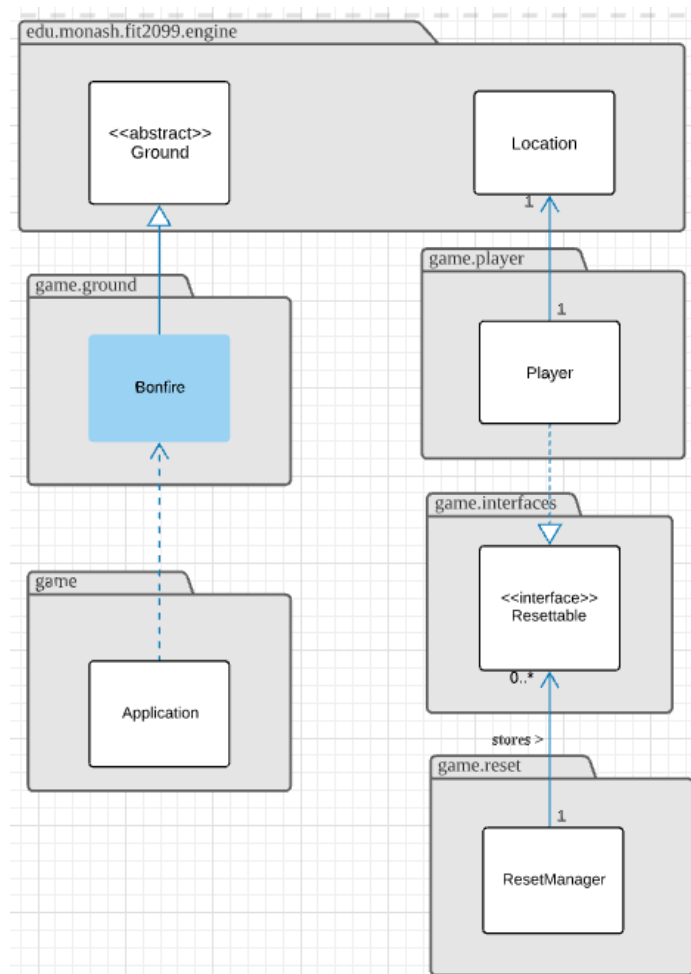


Figure 2B: First Changes Class Diagram for Bonfire

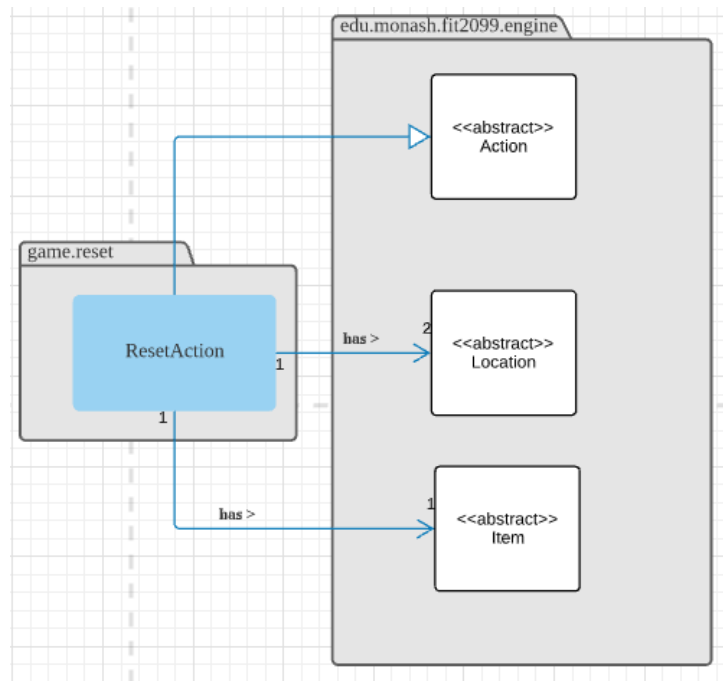


Figure 3B: First Changes Class Diagram to reset using ResetAction

1. Bonfire does not store ResetManager as its field as ResetManager is a global singleton instance that can be accessed anywhere.
2. Estus Flask does not implement Resettable anymore. Player has the EstusFlask as an item in inventory, so we have Player to implement Resettable, and EstusFlask can be reset together with the Player instance.
3. ActorLocations do not implement Resettable because we will store Location of the Player before he dies in ResetAction.
4. ResetAction does not need to target Bonfire to trigger ResetManager because ResetAction can get the ResetManager to do the reset anytime since ResetManager is a global singleton instance.

ResetAction is implemented this way to satisfy the design principles such as **Open/Closed Principle (OCP)** and **Liskov Substitution Principle (LSP)**. In `processActorTurn()` in `World` class, actions for every actor will be obtained, in order to maintain the original implementation, we have to follow these 2 principles, otherwise adding this new reset feature without following OCP might break the old features.

Second changes in Requirement 2: Bonfire

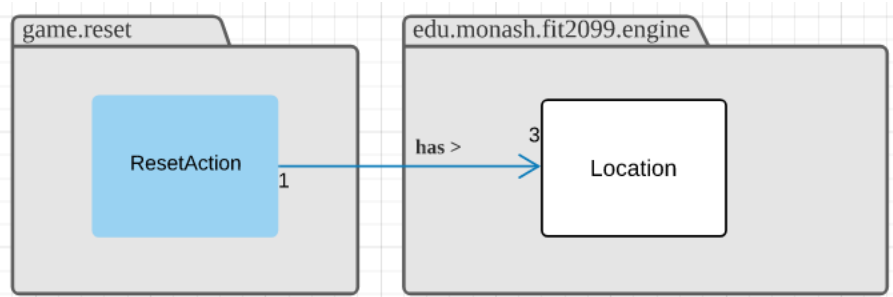


Figure 3C: Update relationship of Bonfire with Location

Now the `ResetAction` will keep track of the player's respawn location so the bonfire will now have one more `Location` type attribute (that makes it 3 `Locations` in Bonfire) to store the location of where the player should be respawned.

Requirement 3: Souls

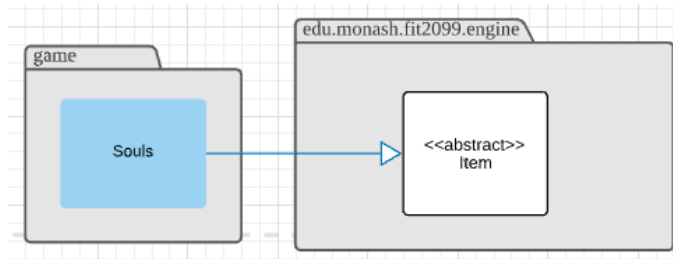


Figure 4A: Old Class Diagram for Souls

A new class called Souls, which extends from the Item abstract class, is created and has an integer attribute to keep track of the number of souls a player has. One of the reasons for the Souls class to inherit the Item class is that the player can interact with Souls instance (token of souls) after the player dies, and the player can regain the souls by using PickupItemAction class methods. This implementation not only satisfies the feature (resetting souls in Player and allowing the player to interact with it later) in Requirement 6, but also achieves an important design principle, **Reusing the code (Reusability)**. Besides, this implementation also reduces dependencies (**ReD design principle**) between the Souls class and the Player class as souls instance can be stored as an item in the actor's inventory. Another way to keep track of souls in the player is to create an integer attribute in the Actor class. However, this alternative does not work for the part where the player can interact with souls that have a specific displayCharacter on the map.

First changes in Requirement 3: Souls

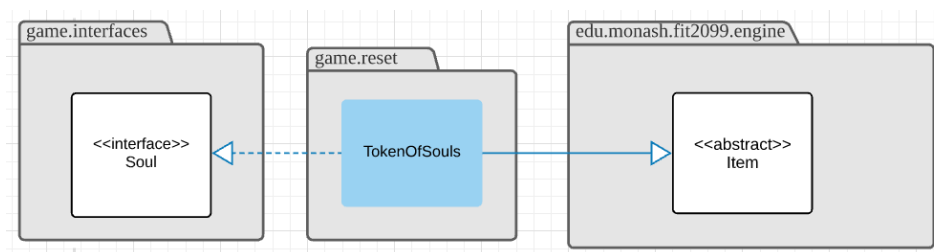


Figure 4B: New Class Diagram for Souls

1. Name of the class changes from Souls to TokenOfSouls. (yes, just name changes)
2. TokenOfSouls is now implementing a Soul interface so that it can use the same methods that Player class uses to transfer souls. This satisfies **OCP** by following the original implementation to transfer souls. This implementation also follows **LSP**, **abstraction** and **encapsulation** design principles as it uses an instance of subclass when transfer of souls is expecting an instance of base class, uses high level abstraction and hides the inner implementation that does the transferring of souls.

Requirement 4: Enemies

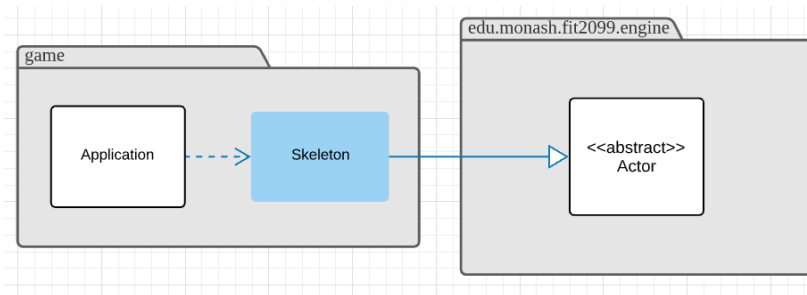


Figure 5A: Old Class Diagram for Enemies

Another type of enemy, Skeleton, which inherits from the Actor abstract class, is implemented, just like the other enemies, Undead and also Lord Of Cinder. The Skeleton class will be implemented similarly to other enemies' classes so that similar properties and capabilities can be accessed and managed by the Actor class (we are **reusing the existing code**). Special features, such as the Skeleton can resurrect itself with a 50% success rate after the first death, and the Ember Form by Lord Of Cinder, can be treated as Status type capabilities, where they will be triggered under certain circumstances.

First changes in Requirement 4: Enemies

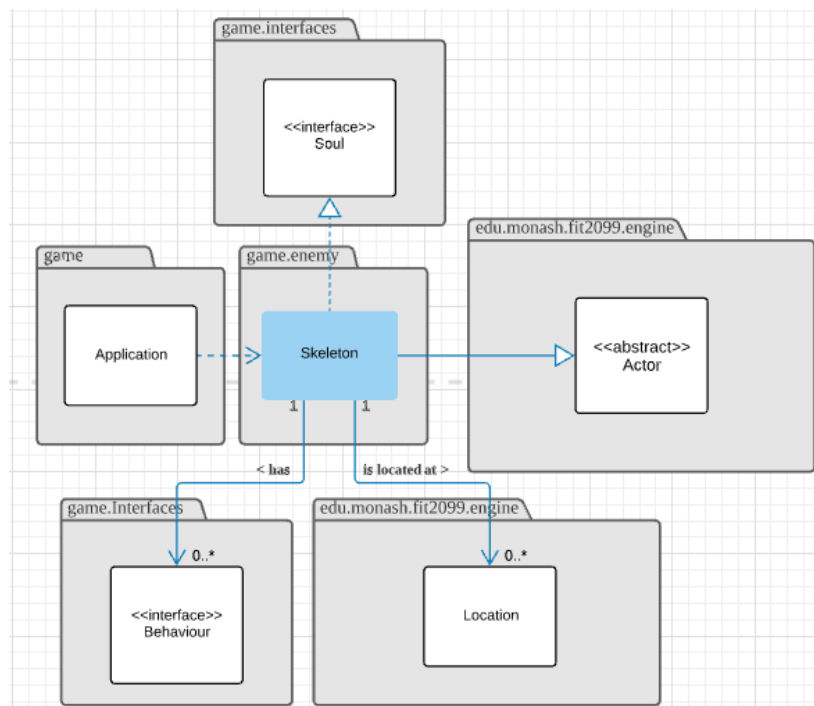


Figure 5B: First Changes Class Diagram for Skeleton

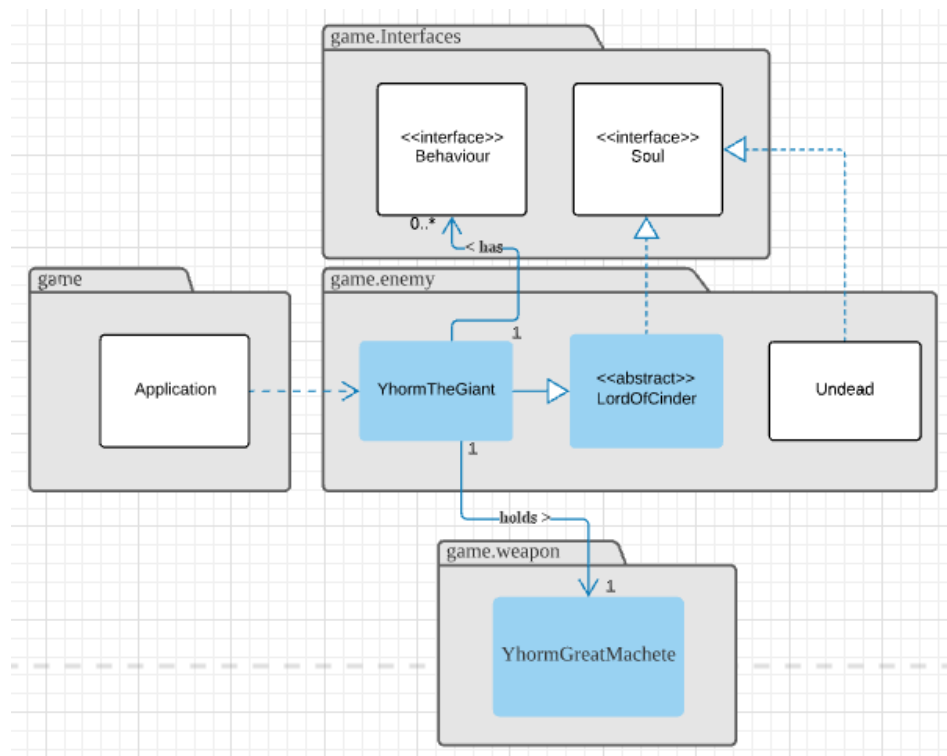


Figure 5C: First Changes Class Diagram for Yhorm the Giant and Undead

A new class, `YhormTheGiant`, is created where it extends from the abstract class, `LordOfCinder`. The `LordOfCinder` class is changed to an abstract class as it represents the boss of the game and can be used to create other bosses (**Reusability**). The `Skeleton`, `Undead`, and `YhormTheGiant` classes are changed to implement the `Soul` class in order for souls to be transferred to the Player once they are killed (**Open/Closed Principle**). All the enemy classes will consist of a behaviour instance that stores the enemy's behaviour accordingly. The `Skeleton` class is also dependent on the location as the resurrected skeleton must know their initial location in order to be resurrected in their original location.

Second changes in Requirement 4: Enemies

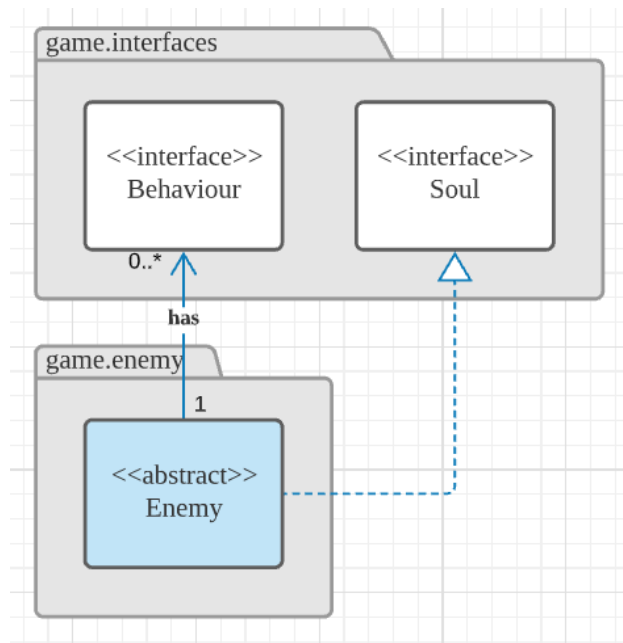


Figure 5D: Second Changes Class Diagram for Enemies Abstract Class

Since all the enemies have similar behaviours and actions, a new `Enemy` abstract class is created where it will be extended by the other enemies (**Liskov Substitution Principle**). The abstract class will also implement the `Soul` class since the player will gain soul when the enemies are killed (**Open/Closed Principle**).

Requirement 5: Terrains - Valley and Cemetery

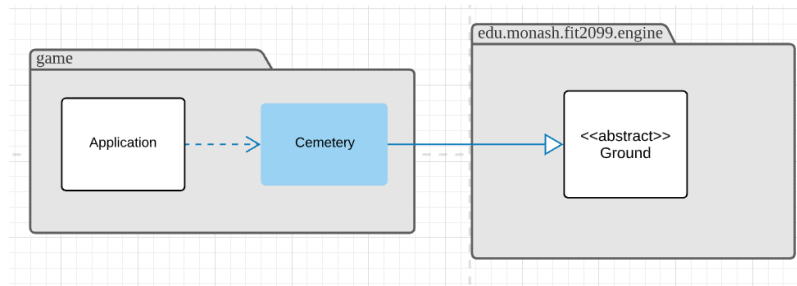


Figure 6A: Class Diagram for Terrains

A new class called Cemetery that inherits the Ground abstract class (because Cemetery has specific displayCharacter as well) will be added. The features of the valley and cemetery, which instantly kill the player and have a 25% success rate to spawn Undead respectively, can be implemented as capability and method respectively. Most importantly, the Cemetery class can be reused as several cemeteries are required to be placed on the game map. (**Reusability**)

First changes in Requirement 5: Terrains

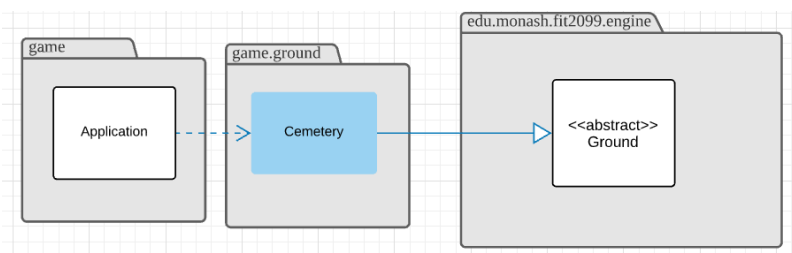


Figure 6B: New Class Diagram for Terrains (relationship remains unchanged)

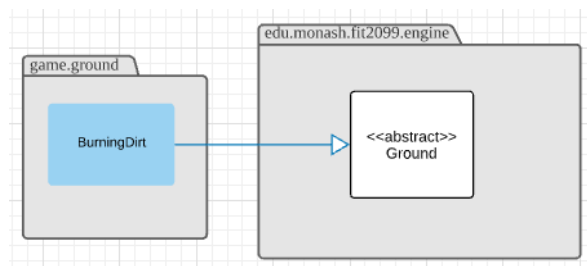


Figure 6C: New Class Diagram for Terrains

A new class, BurningDirt, that inherits from the Ground abstract class, is created. This class is used when Yhorm The Giant is in Ember Form, where the surrounding dirt will burn for 3 rounds and hurt the player's hit points by 25 (**Single Responsibility Principle**).

Requirement 6: Soft reset

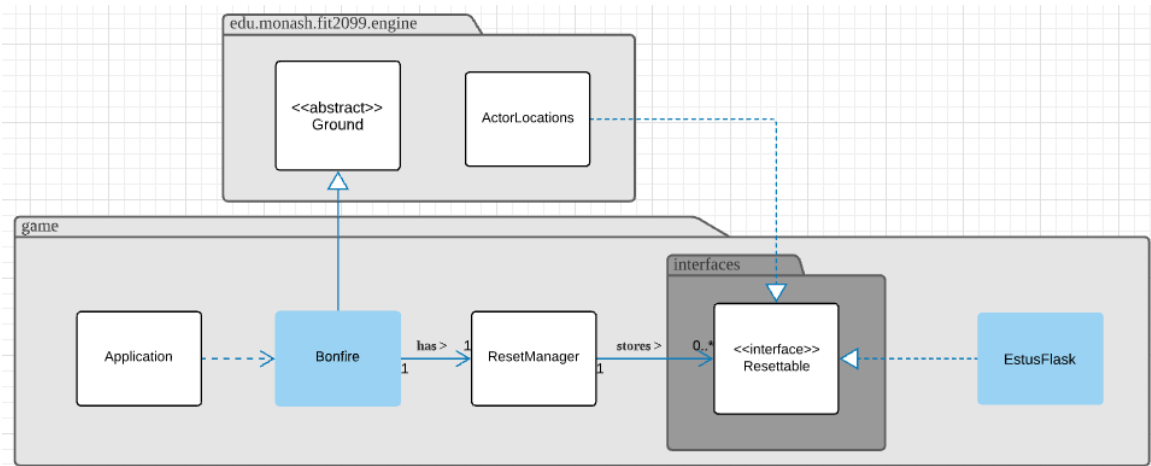


Figure 7A: Class Diagram for Soft Reset
Note: Same as class diagram in Requirement 2

To implement a soft reset when the player dies, the player can have a Status type capability to reset the game when the player's hit points drop to zero. As it is mentioned in Requirement 2 and 3, the player can reset resettable instances via the ResetManager class and the souls instance will be shown at where the player died (position before the player died by falling into the valley). The difference between the 2 types of reset is that the player can reset resettable instances by interacting with the bonfire in Requirement 2, while a soft reset is by checking the player's hit points, where a hit points of 0 and below will trigger the ResetManager in the Bonfire class. **(Reusing existing code, Reusability)**

First changes in Requirement 6: Soft Reset

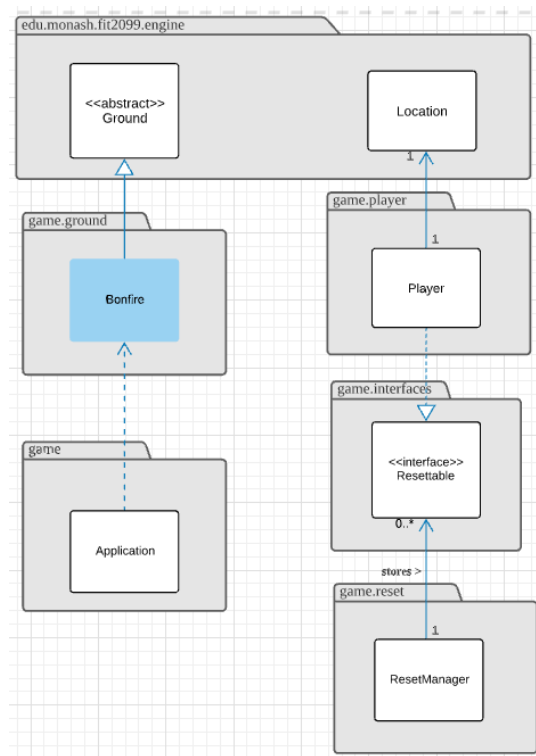


Figure 7B: New Class Diagram for Soft Reset
Note: Same as new class diagram in Requirement 2

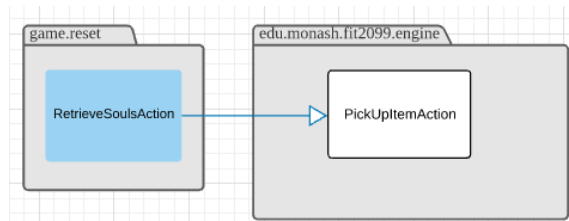


Figure 7C: New Class Diagram for Retrieving Lost Souls

1. Both Soft reset and “Rest at Bonfire” use the same ResetAction class. However, there will be 2 kinds of constructor, one will accept the player's location before he dies as argument while another one is without any argument. Soft reset is triggered using the constructor that takes the player's location as argument because ResetAction will need to know where to drop the player's soul as TokenOfSouls.
2. We also apply **OCP** and **LSP** to add RetrieveSoulsAction so that this action can be returned during the Player's playturn() (original implementation that follows **OCP**) and substitute an abstract action with concrete action.

Requirement 7: Weapons

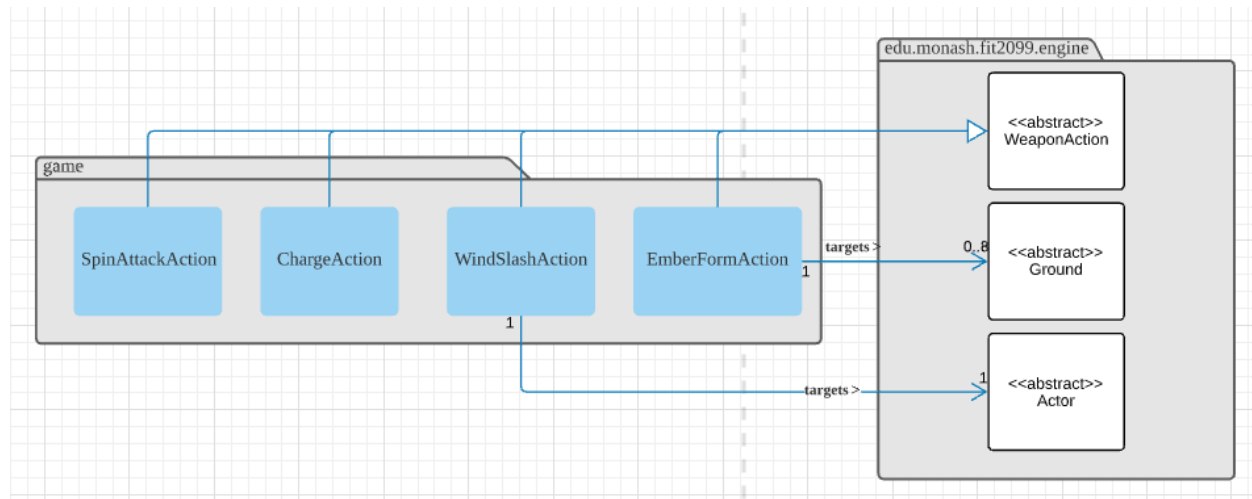


Figure 8A: Class Diagram for Actions in Weapons

All of the melee weapons are each made as a class, which extend from the GameWeaponItem class. The classes will contain methods to provide the weapon's Passive skills, and call on their Active skills. Each weapon has specific capabilities that can be activated by the holder (the active skills). These skills are implemented in different WeaponActions classes, such as SpinAttackAction class (Giant Axe), ChargeAction class, WindSlashAction class (both for Storm Ruler) and EmberFormAction class (for Yhorm's Great Machete). Although the capabilities of different weapons are mixed, they can be differentiated by their display character.

First changes in Requirement 7: Weapons

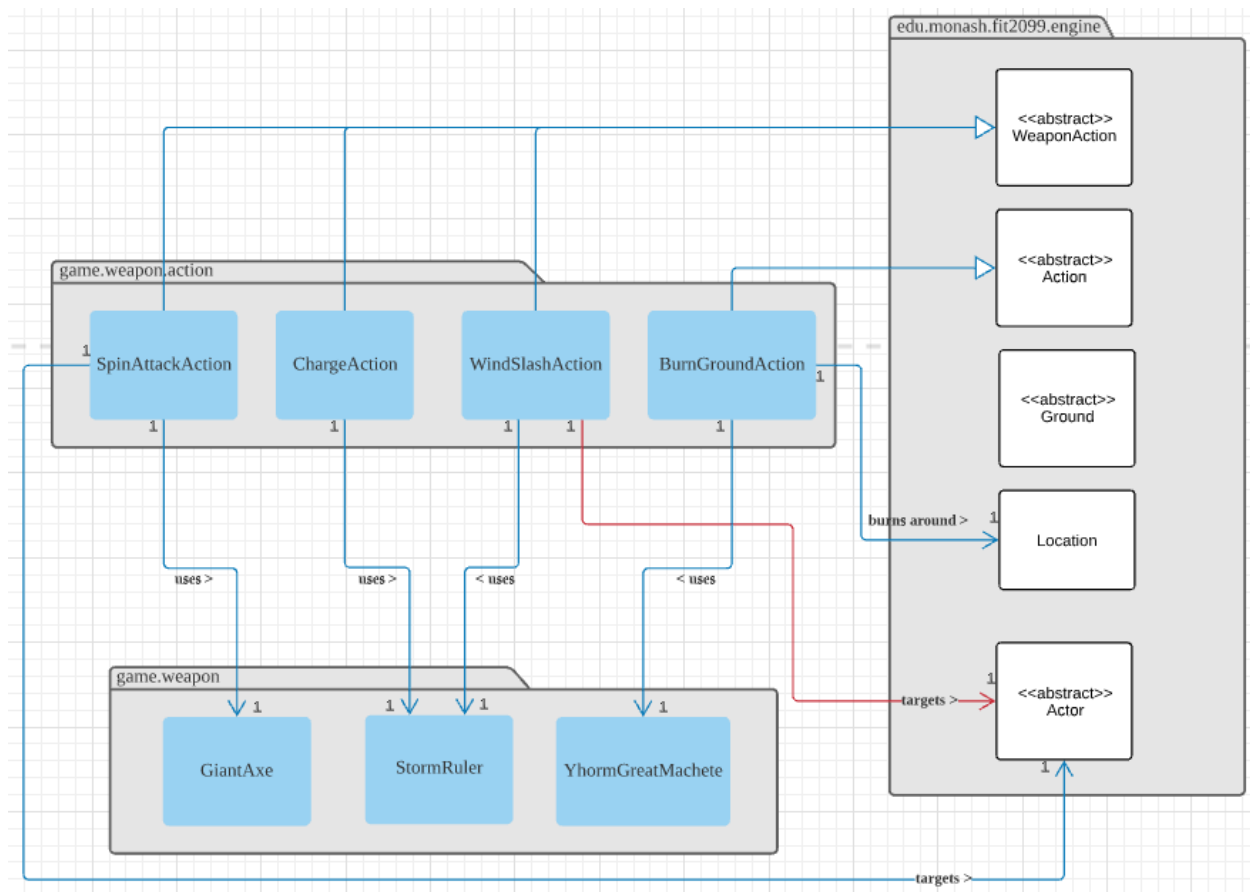


Figure 8B: First Changes Class Diagram for Action and Weapons

The weapons that were previously grouped and will be created using the `MeleeWeapon` class are now created separately, with each weapon having its own independent class due to the weapons' own functionality and active skills (**Single Responsibility Principle**). The new classes, `Broadsword`, `GiantAxe`, `StormRuler`, and `YhormsGreatMachete` were created by extending the `GameWeaponItem` abstract class where their active skills can be called by overwriting the `getActiveSkill()` method from the parent class (**Reusability**). The existing `MeleeWeapon` class is also deleted since all the weapons have their own classes and therefore do not serve any purpose or functionality. The `BurnGroundAction` class, which initially extends the `WeaponAction` abstract class, is changed to extend the `Action` class because the action is required to be added to the list of actions for Yhorm The Giant and will be called when it is in Ember Form.

Requirement 8: Vendor

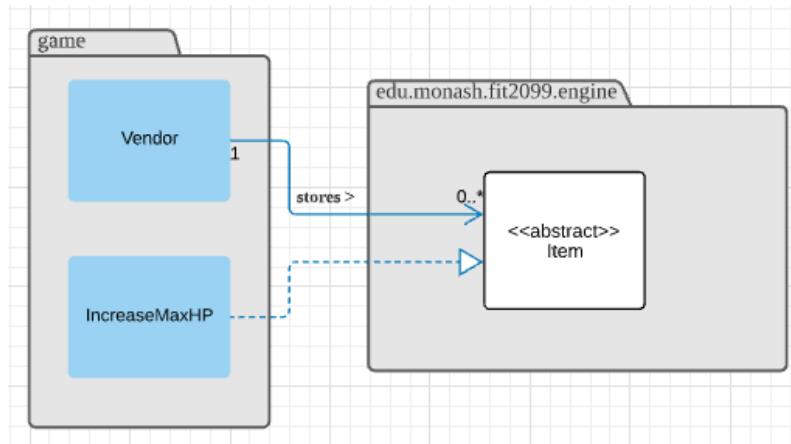


Figure 9A: Old Class Diagram for Vendor

A new FireKeeper class which extends the Ground abstract class is created so that it can be displayed on the console as 'F' and the player can interact with the vendor to purchase one of the two weapons in exchange for Souls. In order for the vendor to be able to sell those weapons, two Action classes called BuySwordAction and BuyAxeAction will be introduced so that the player knows they have 2 items to choose from. The FireKeeper class will also only allow an actor to buy weapons if they have the BUY trait (which only the player will have). The 2 Buy Action classes inherit the parent class WeaponAction, making the child classes be able to use the execute() and menuDescription() methods for executing the buying sequence and display appropriate messages in the console (**DRY** and **Reusability**). Both the Buy Action classes is able to create an instance of their designated weapon (using BroadSword for BuySwordAction and GiantAxe for BuyAxeAction), retrieve the player's currency amount to decide if they can buy the weapon or not (using Player's numberOfSoul) and also swap out the purchased weapon with the player's current weapon (using SwapWeaponAction). Thus each of the Buy Action classes make it possible for the whole process to happen all in one class (**Single Responsibility Principle**) .

First changes in Requirement 8: Vendor

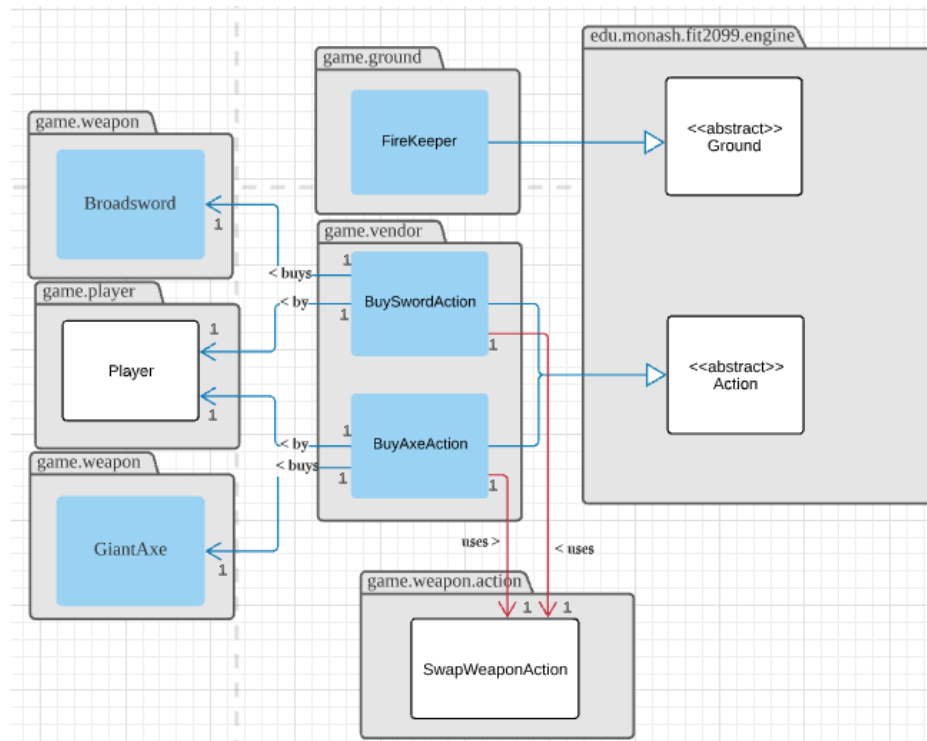


Figure 9B: New Class Diagram for FireKeeper(Vendor)

Note: relationship lines are in red (to distinguish from line in blue)

Previously, we had a Vendor class which will be the one and only class responsible for all the actions related to the Fire Keeper (also made **Classes should be responsible for their own properties** possible). However, each Action class is only able to execute one specific action and also hold one hotkey for that particular action. It would also be impossible to implement the vendor into the map using Ground class if the Vendor class already inherits from Action.

Thus, we split the three requirements for vendor into three classes: FireKeeper class which inherits from Ground to display the vendor on the map and have the player be able to buy items from them via `allowableActions()`, BuySwordAction that inherits from WeaponAction which will create an instance of Broad Sword and swap it with the player if the player chooses to buy it, and BuyAxeAction that also inherits from WeaponAction which will create an instance of Giant Axe and swap it with the player if the player chooses to buy it.

Second changes in Requirement 8: Vendor

2 new Buy Action classes, BuyMacheteAction and BuyBowAction, included under `allowableActions()` for FireKeeper. The 2 new classes will trade the weapons Yhorm's Great machete and Darkmoon Longbow.

New Requirements (Assignment 3)

Requirement 1: New Map and Fog Door

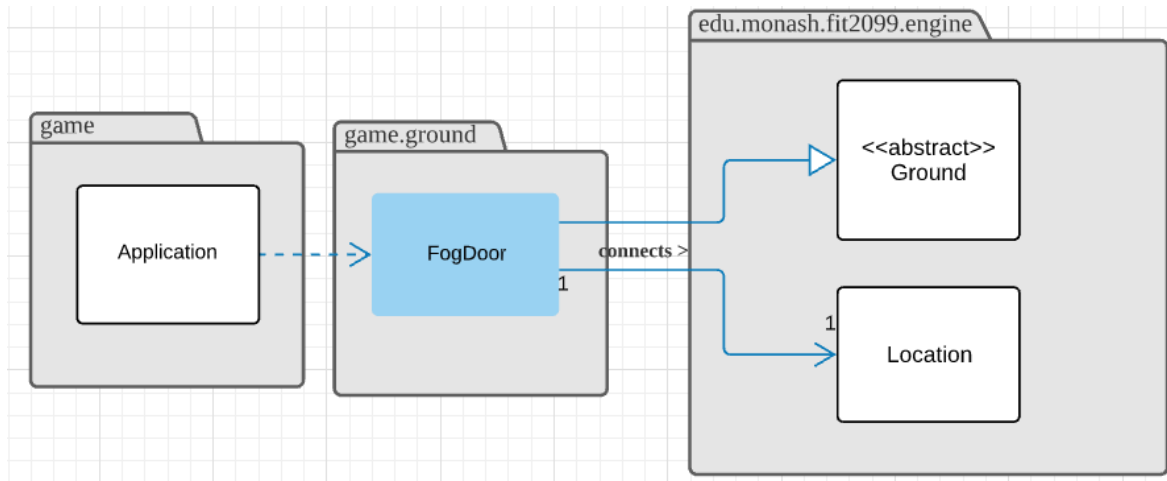


Figure 10 : Class Diagram for FogDoor class

Requirement 1 allows the game to have another map so we initialise a new map in Application and FogDoor instance can be initialized in ground factory. We allow the FogDoor to inherit from abstract Ground class so that when the player is near or stepping on the “=”(FogDoor), the player can make an optional action to travel to a new map by overriding allowableActions() from abstract Ground class. This way, we can **polymorph** allowableActions() and follow **Liskov Substitution Principle**.

Requirement 2: Updated Bonfire

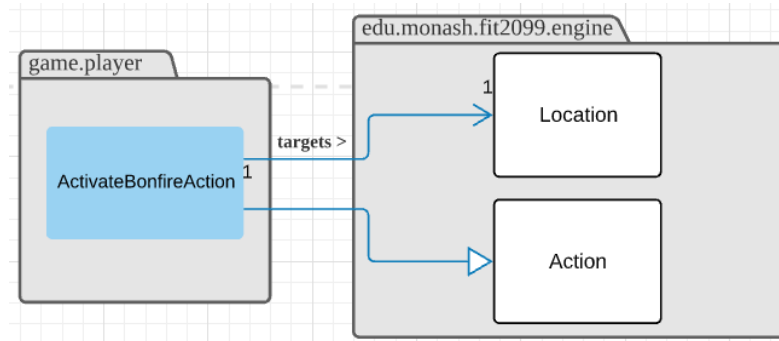


Figure 11: Class Diagram for ActivateBonfireClass class

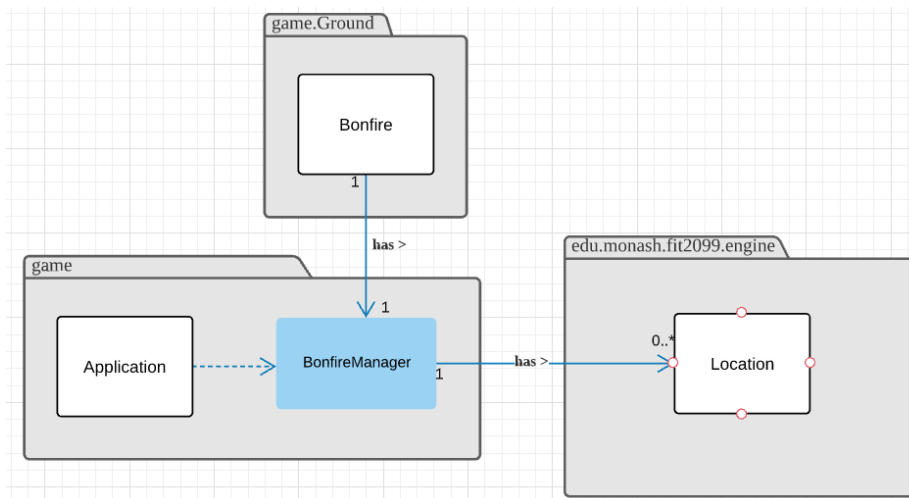


Figure 12: Class Diagram for BonfireManager class

Since we will have a new bonfire on the new map, the bonfire on the new map is made to be able to interact with only if it is activated beforehand. Therefore, we create a new class that inherits from Action class so that when the player gets near or stands on the bonfire, this action can be returned and displayed to the menu. It has an association with the Location class because the action will activate the bonfire at that location. We are applying the Liskov **Substitution Principle** and **Open/Closed Principle** to allow **polymorphism** and reuse the original implementation.

BonfireManager class is also created to keep track of all the locations of bonfires. BonfireManager acts as an attribute of bonfire because bonfire has to know what locations that the player can teleport to when the player interacts with the bonfire.

Requirement 3: Aldrich the Devourer (Lord of Cinder)

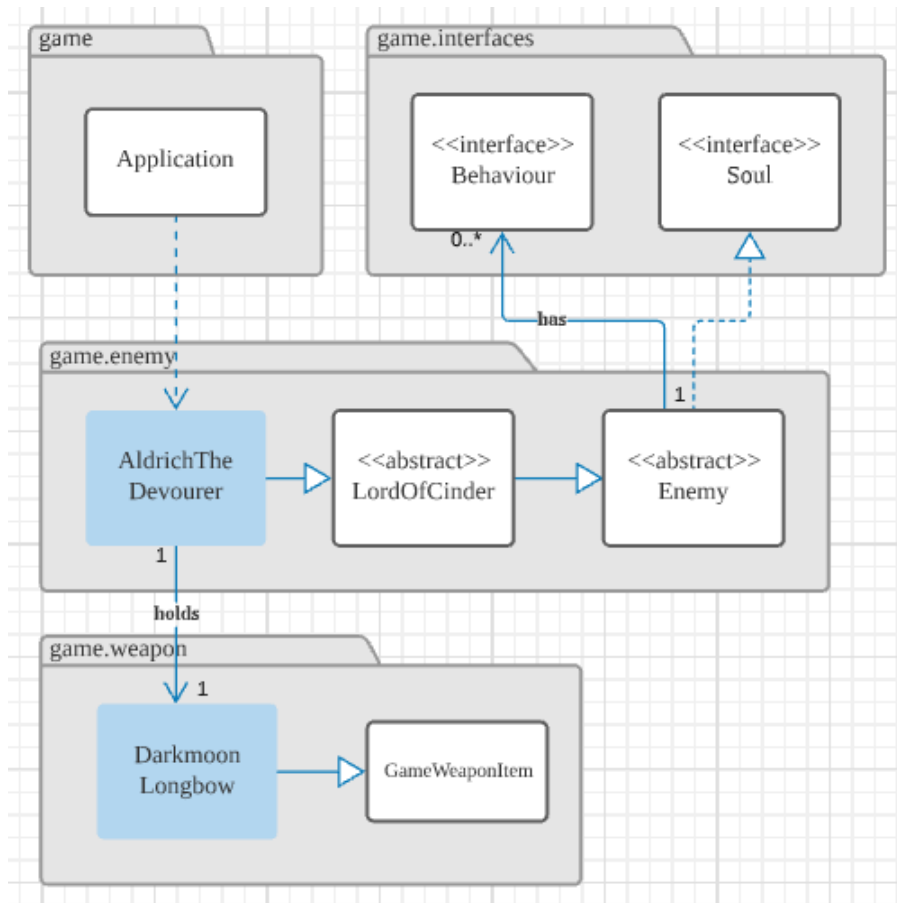


Figure 13: Class Diagram for AldrichTheDevourer class

Since there is a second boss, a new class, `AldrichTheDevourer`, is created where it extends from the abstract class, `LordOfCinder` (**Reusability**). Another new weapon, `DarkmoonLongbow`, is also created by extending the `GameWeaponItem` class, and this weapon is held by `AldrichTheDevourer`. Since all Lords of Cinder will drop cinder (corpse) when they die, the new class, `CinderOfTheLord`, can be associated with the `LordOfCinder` class and will be dropped in the `DeathAction` class (**Liskov Substitution Principle**). The cinders can be easily differentiated by their names, as the name of the cinder is dependent on the name of the lord.

Requirement 4: Mimic / Chest

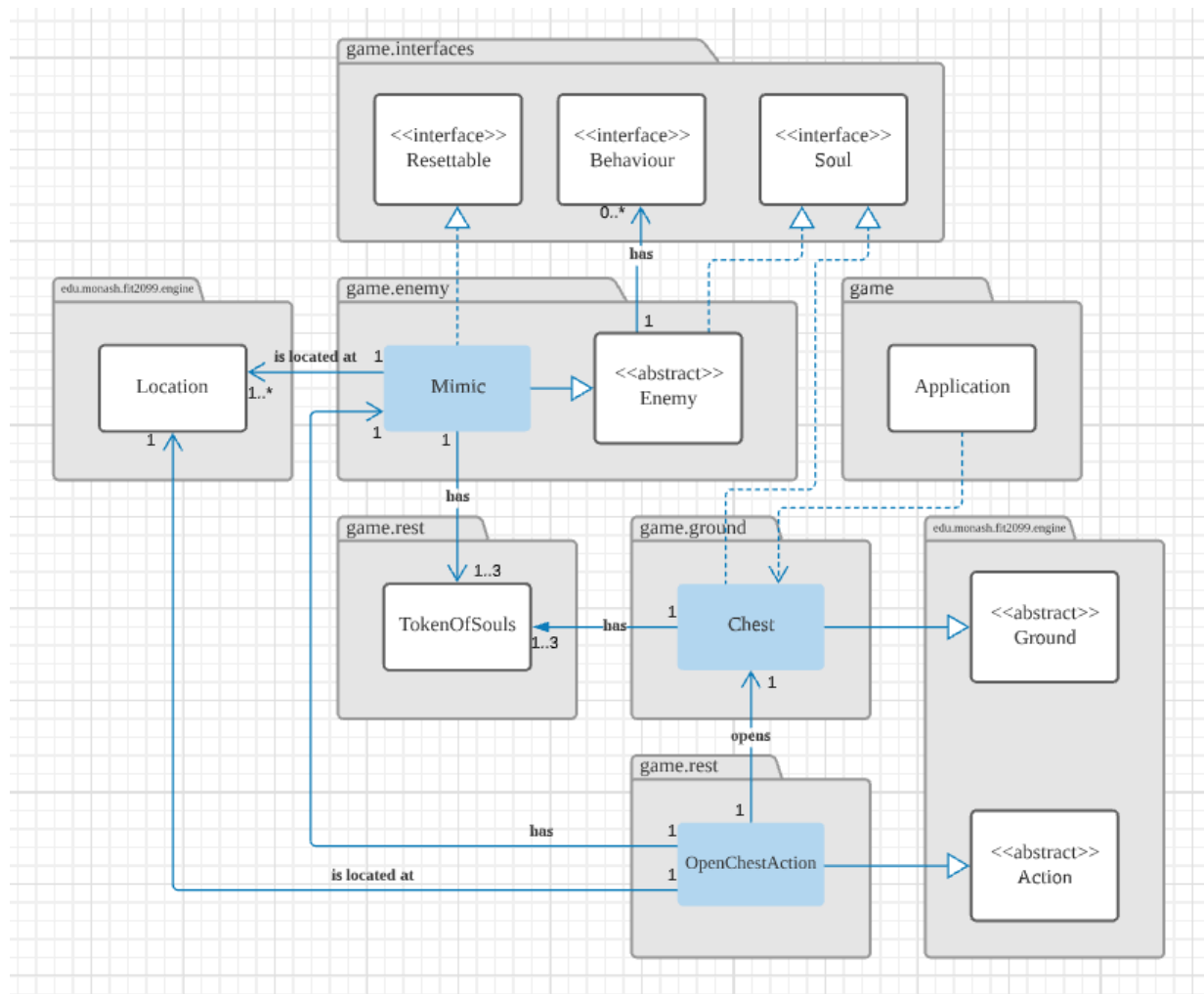


Figure 14: Class Diagram for Mimic and Chest class

A new class, Chest, is created and is extended from the abstract Ground class. Both Chest and Mimic have an association with TokenOfSouls, as once the chest is opened with OpenChestAction (a newly created class which extends from abstract Action class) (**Single Responsibility Principle**), the chest either gives 1 to 3 tokens of souls worth 100 souls each or creates a Mimic, a new enemy that drops 1 to 3 tokens of souls when killed. The Mimic class is created by extending the abstract Enemy class (**Liskov Substitution Principle**) and therefore allows reusability of the code. The Mimic class also has an association with the Location class, as each Mimic needs to be placed back in its original position and transforms back into a chest during reset. This is also the reason for Mimic to implement Resettable and as Mimic is the only enemy that gets reset.

Requirement 5: Trade Cinder of Lord

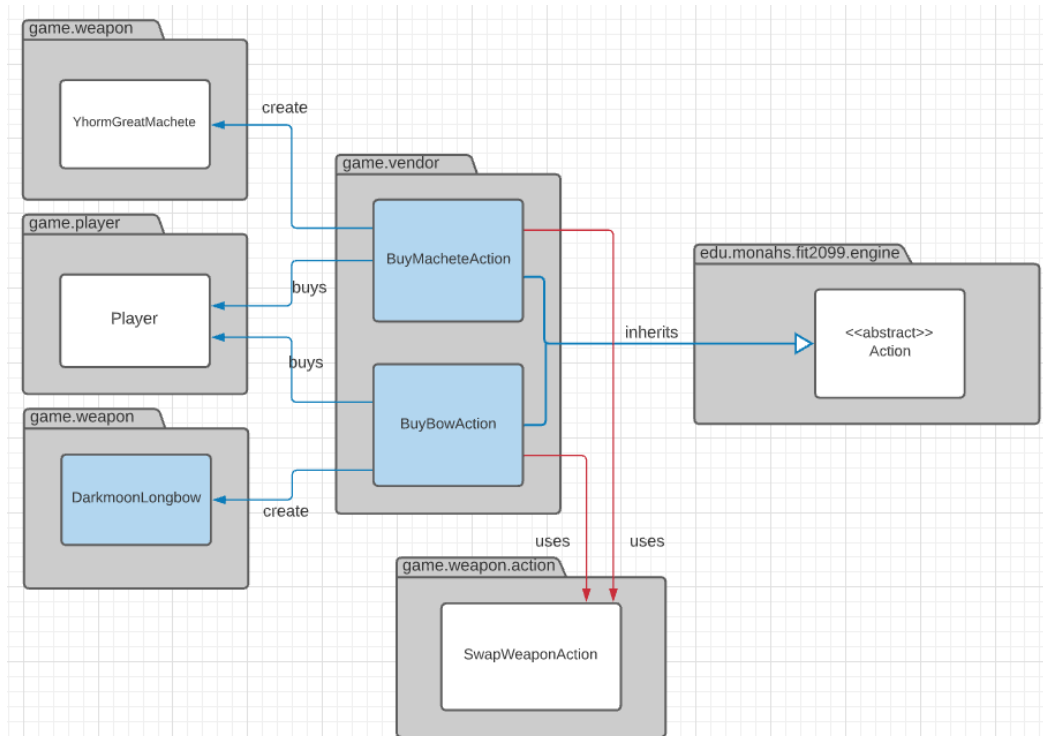


Figure 15: Class Diagram for New Actions for FireKeeper (Vendor)

The Vendor (FireKeeper) is capable of trading two additional weapons, which are the two Lord of Cinder weapons. Similar to the Action classes for sword and axe, two new classes, `BuyMacheteAction` and `BuyBowAction` are made. The menu option to buy the weapons will be available in the console when the player is adjacent to the vendor. The big difference here with the other weapons is, instead of souls, the player must trade the `CinderOfLord` (corpse of the Lord of Cinder) to receive their corresponding weapon. The exchange process is the same as the previous buy action classes. The only difference being the `execute()` method, instead of checking for enough souls, will check if the player carries the corresponding `CinderOfLord` to deny or proceed with the trade.

The same principles apply for these 2 new classes as well. The 2 Buy Action classes inherit the parent class `WeaponAction`, making the child classes be able to use the `execute()` and `menuDescription()` methods for executing the buying sequence and display appropriate messages in the console (**DRY** and **Reusability**), and the 2 classes are capable of executing for the whole process all in one class (**Single Responsibility Principle**).

Interaction Diagram

Interaction diagram is a type of UML diagram that is used to visualize the interactive behavior of a system. Interaction diagrams that are commonly used are sequence diagrams and communication diagrams. The purpose of the interaction diagram is to capture the dynamic behaviour of the system and is used to represent how one or more objects in the system are connected with each other.

Interaction Diagram 1: Drink Action

The first interaction diagram will be for Requirement 1, where the player drinks a health potion called Estus Flask. The Estus Flask has three charges, and each charge will heal the player with 40% of the maximum hit points.

In the drink action, when a player chooses to consume the Estus Flask, the method `getDrink()` is used in the Player class to obtain the Estus Flask. The number of Estus Flask is then obtained with the `getNumberOfEstusFlask()` method where the integer is then checked with the `canDrink()` method. The return value will be true if there is remaining available Estus Flask and false when the number of Estus Flask is 0 and a return message "No available Estus Flask " will be displayed on the console to the player. If the return value is true, the maximum hit points will be obtained through the getter of the maximum hit points and the `heal()` method in the Player class will be overwritten and used to increase the hit points by 40% of the maximum hit points. The message "a: Unkindled drinks Estus Flask (numberOfEstusFlask/3)" will be displayed after the number of Estus Flask is decremented using the `decrementNumberOfEstusFlask()` method.

The figure for the drink action can be clearly seen in Figure 16A.

Changes in Interaction Diagram 1: Drink Action

Instead of checking the number of Estus Flask with a `canDrink()` method, the number of Estus Flask is compared to 0, and if it is true, a return message "No available Estus Flask" will be displayed. Otherwise, another comparison will be made between the hit points and the maximum hit points of the player which are obtained through the `getHitPoints()` method and `getMaxHitPoints()` method. The menu will display the message "The player's hitpoints is full" if the player's hitpoints is not less than the maximum hit points. If the hit points is lower than the maximum hit points, the number of Estus Flask will be decremented by `decrementNumberOfEstusFlask()` method and a message "a: Unkindled drinks Estus Flask (numberOfEstusFlask/3)" will be displayed.

The figure for the new drink action can be clearly seen in Figure 16B.

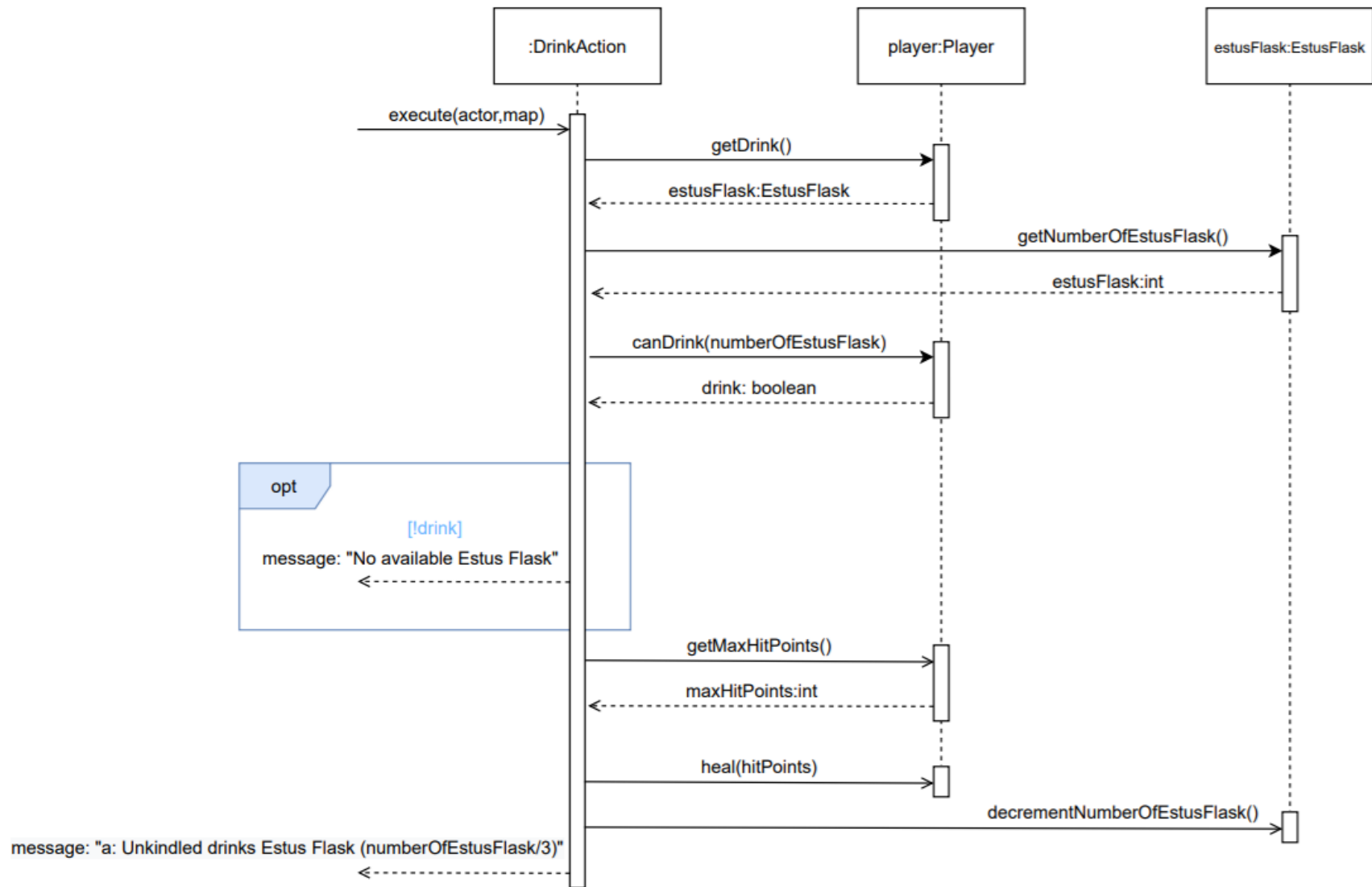


Figure 16A: Old Sequence Diagram for DrinkAction

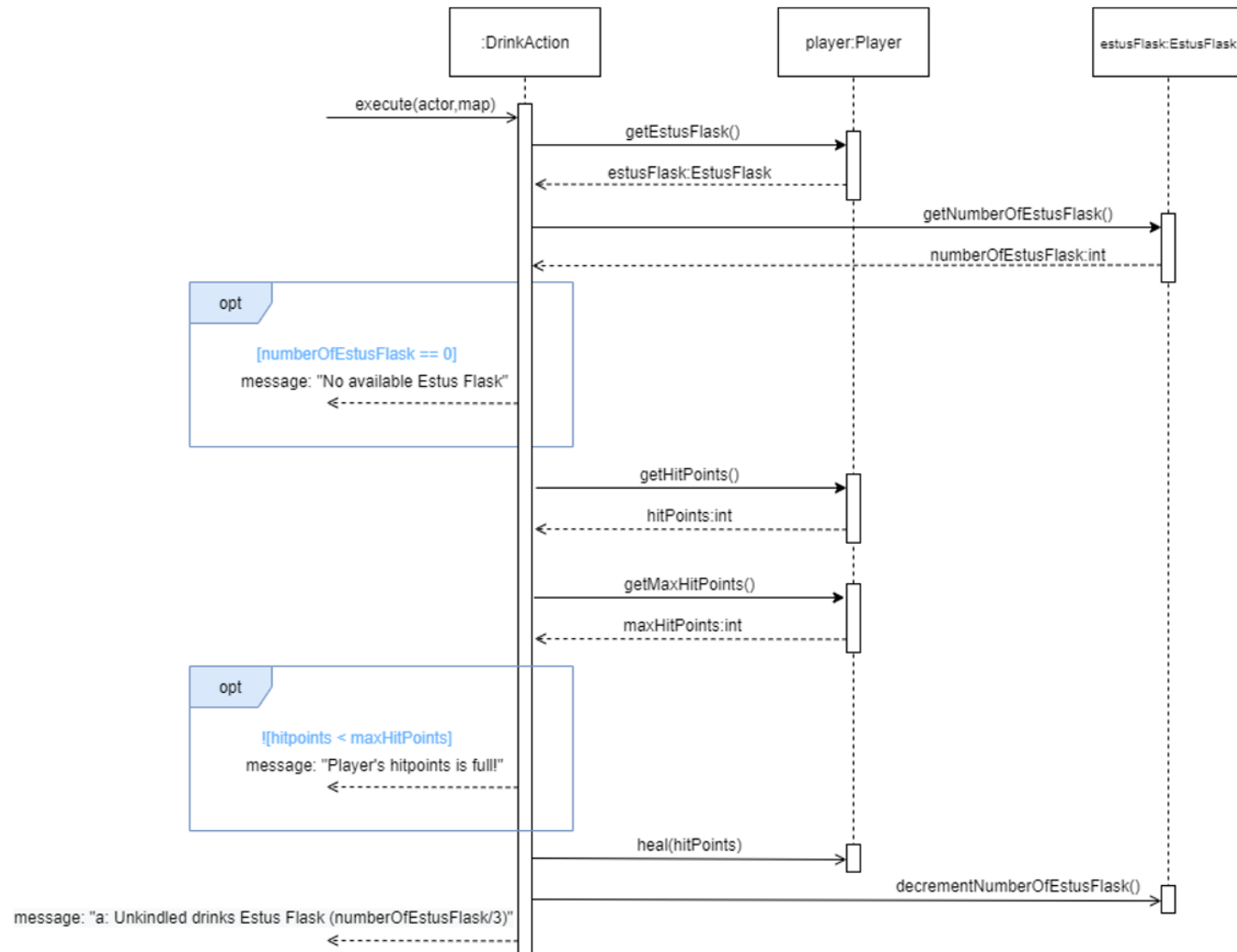


Figure 16B: First Changes Sequence Diagram for DrinkAction

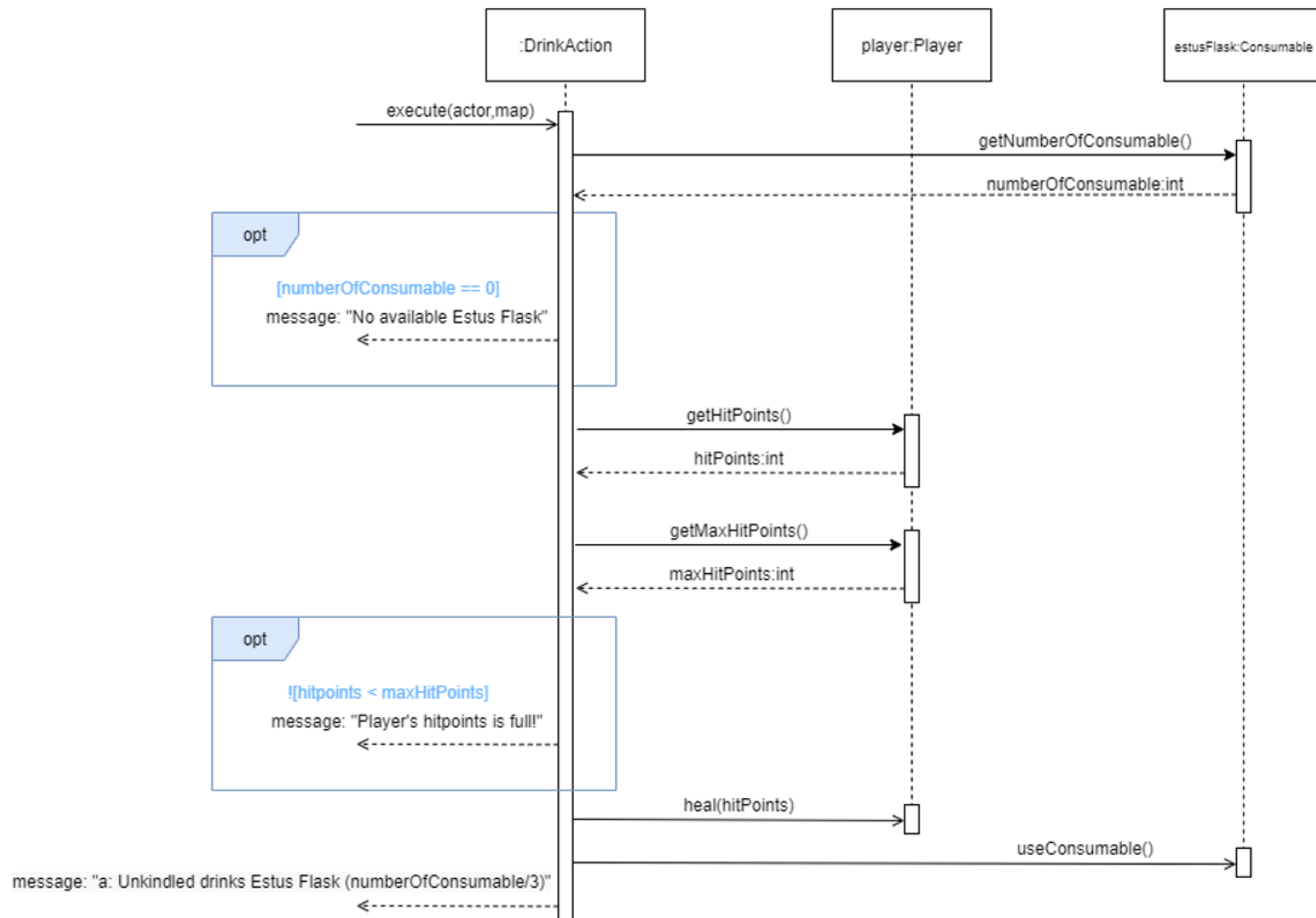


Figure 16C: Second Changes Sequence Diagram for DrinkAction

Interaction Diagram 2: Buying Item at Vendor

The second interaction diagram is related to Requirement 8, the process of buying items from the vendor when the player chooses to do so. The vendor has 2 item options to choose from: buying a Broad sword, or a Giant axe.

The console menu will display 2 additional options, under hotkey B and G, that the player may choose to act on when they are adjacent to the Fire Keeper(F), which is the vendor. When the player chooses an option, the FireKeeper will then execute the exchange for one of their weapons by calling their respective classes.

When choosing to buy the Broad sword, the FireKeeper will call the BuySwordAction class. In the class, the player's information will be taken and a BroadSword will be created. As the action method is executed, the method will retrieve the player's numberOfSoul (the currency). If the player has at least the same amount of souls as the sword price, the price amount will be subtracted from the player's soul. Then, using the SwapWeaponAction class the player's current weapon will be replaced by the new broadsword. It will then print a message of confirmation to show if the transaction was a success. In the case that the player does not have enough souls to buy the weapon, a different message will be displayed to show the unsuccessful transaction.

When choosing to buy the Giant Axe, the FireKeeper will call the BuyAxeAction class. In the class, the player's information will be taken and a GiantAxe will be created. As the action method is executed, the method will retrieve the player's numberOfSoul (the currency). If the player has at least the same amount of souls as the axe price, the price amount will be subtracted from the player's soul. Then, using the SwapWeaponAction class the player's current weapon will be replaced by the new giantaxe. It will then print a message of confirmation to show if the transaction was a success. In the case that the player does not have enough souls to buy the weapon, a different message will be displayed to show the unsuccessful transaction.

Changes in Interaction Diagram 2: Buying Item at Vendor

Previously, we used only a Vendor class for execution of the actions needed to get player information, retrieve the new weapon and swap that weapon. However, as mentioned under the rationale under Requirement 8, Vendor will be incapable of executing more than one action since it is an extension of the WeaponAction class. Thus, we decided to split the purchase process into 2 of the new Buy classes, one for each of the weapons.

The figure of both previous and latest vendor interactions can be clearly seen below in Figure 17.

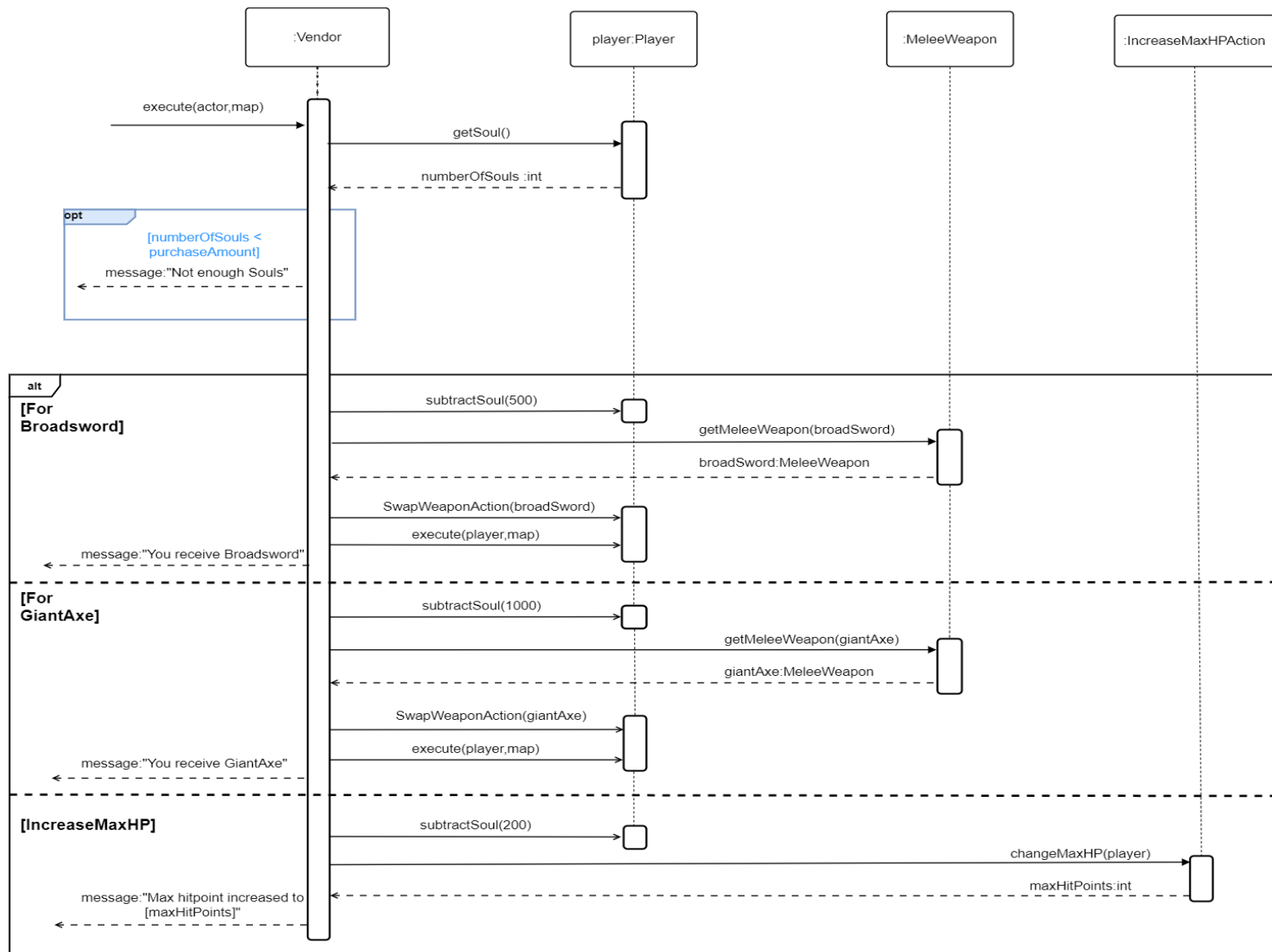


Figure 17A: Old Sequence Diagram for Buying Item at Vendor

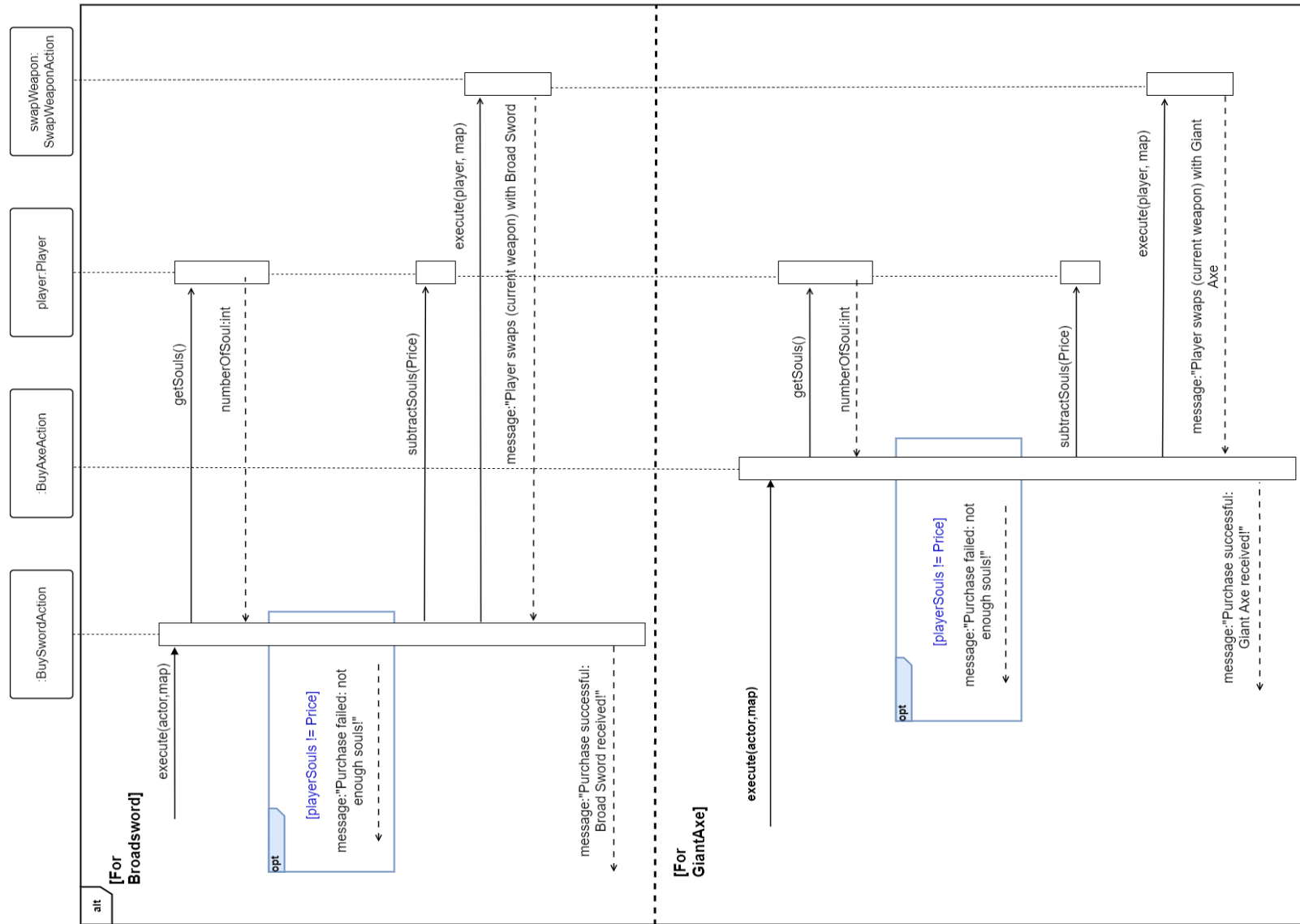


Figure 17B: New Sequence Diagram for Buying Item at Vendor

Interaction Diagram 3: Actions allowed when interacting with Bonfire

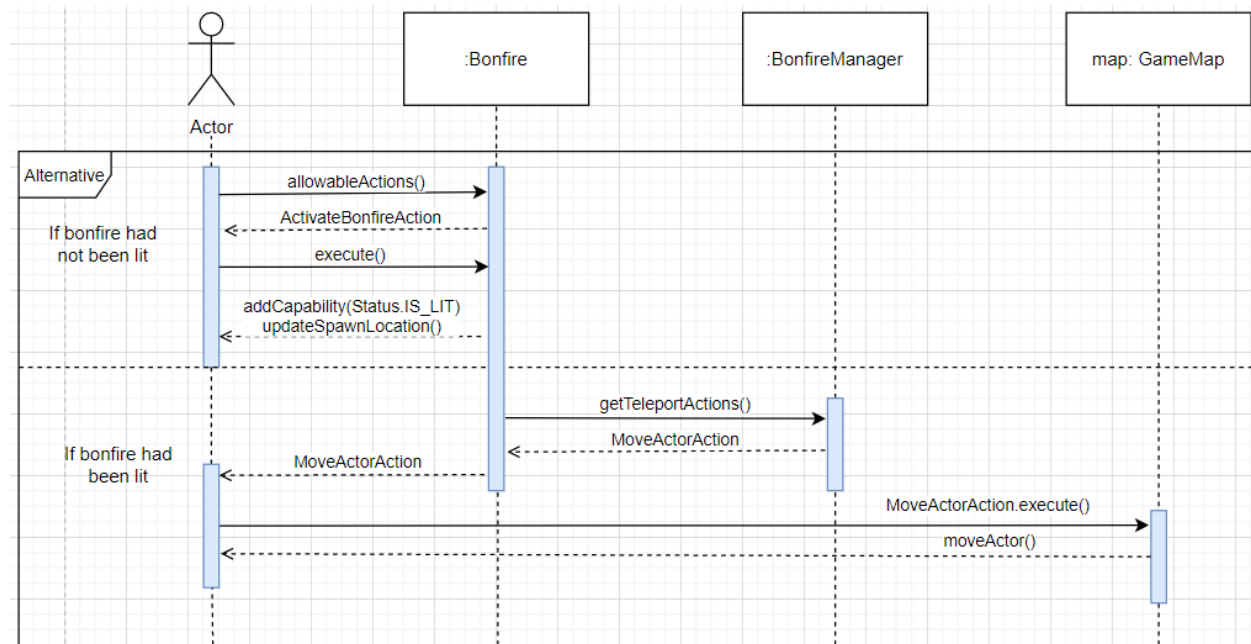


Figure 18: Sequence diagram for Bonfire interaction

When the player interacts with a bonfire, allowable actions can be obtained using the `allowableActions()` method and displayed to the console. The possible actions are `ActivateBonfireAction(Lit Bonfire)` and `MoveActorAction(Teleport to another bonfire)` depending on whether the bonfire had been lit beforehand.

If it is `ActivateBonfireAction`, it will add the `IS_LIT` status to the bonfire and update the player's respawn location(because interaction with the bonfire will update the spawn location).

If the player chooses to teleport, the locations that can be teleported will first be obtained from the bonfire manager and then returned to the player. Number of `MoveActorAction` returned depends on the number of locations being stored in the bonfire manager.