

FIT2099

Assignment 1

Team 2

Tan Hong Yi

Afrida Jahin

Lee Jia Yi

7th September, 2021



Table of Contents

Table of Contents	1
Design Rationale	2
Requirement 1: Player and Estus Flask	3
Requirement 2: Bonfire	4
Requirement 3: Souls	5
Requirement 4: Enemies	6
Requirement 5: Terrains - Valley and Cemetery	6
Requirement 6: Soft reset	7
Requirement 7: Weapons	8
Requirement 8: Vendor	9
Interaction Diagram	11
Interaction Diagram 1: Drink Action	11
Interaction Diagram 2: Buying Item at Vendor	13



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 requirements given 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

Requirement 1: Player and Estus Flask

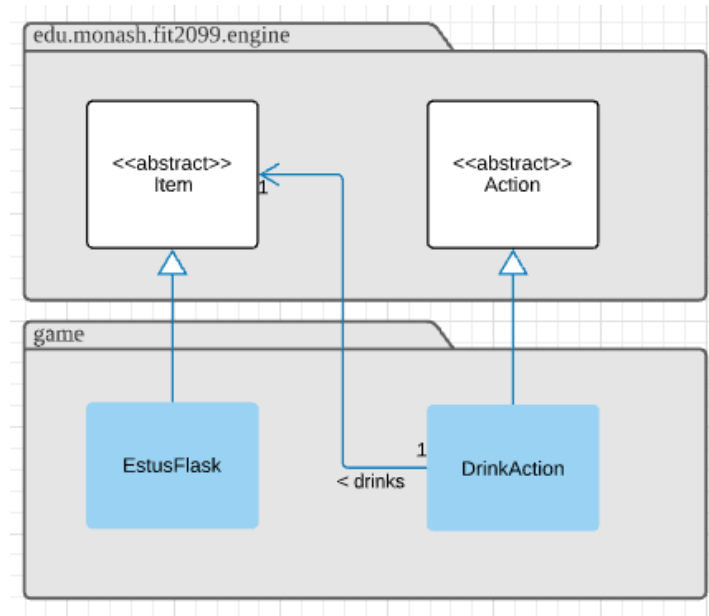


Figure 1: Class Diagram for Player and Estus Flask

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 print 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 `chargers`) 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.

Requirement 2: Bonfire

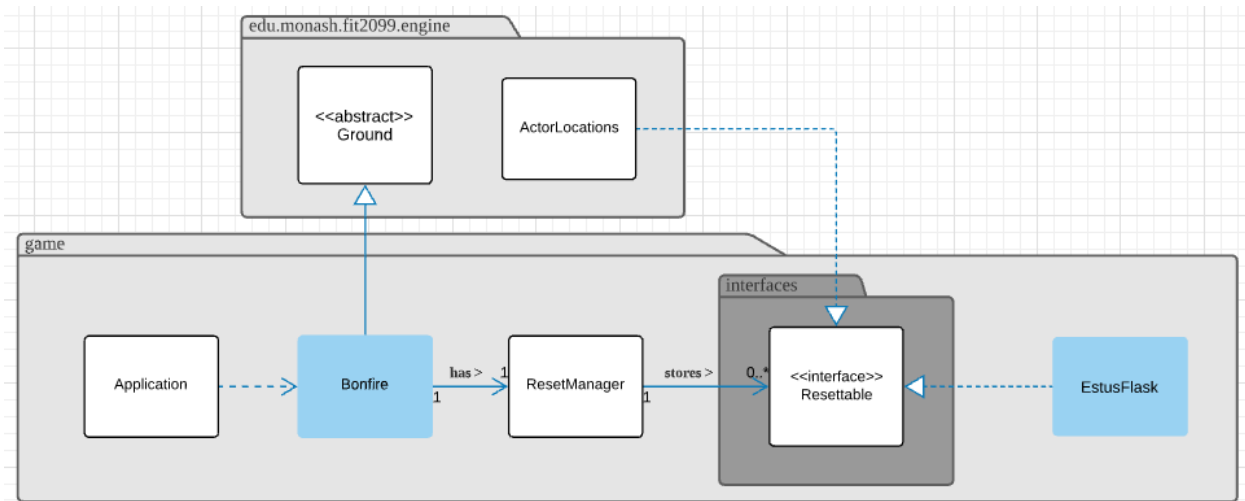


Figure 2: Class Diagram for Bonfire

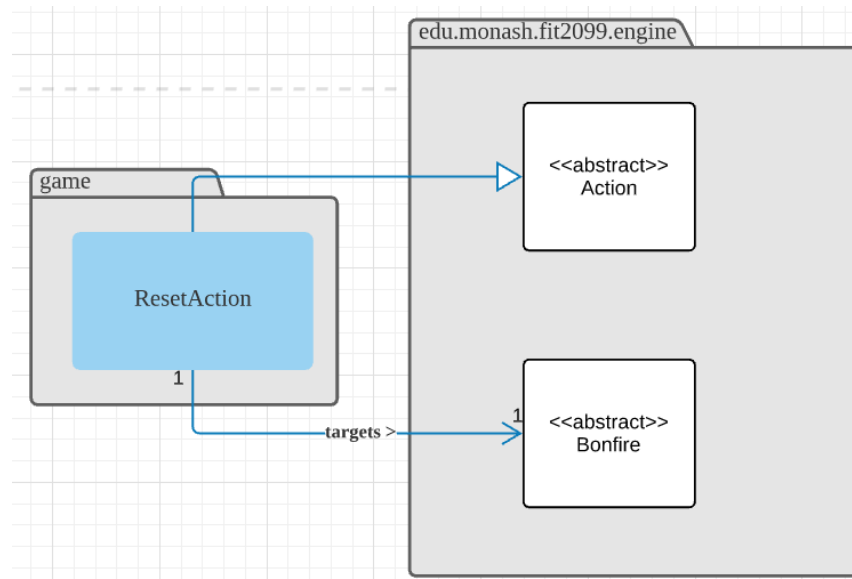


Figure 3: Class Diagram to Reset in Bonfire

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.

Requirement 3: Souls

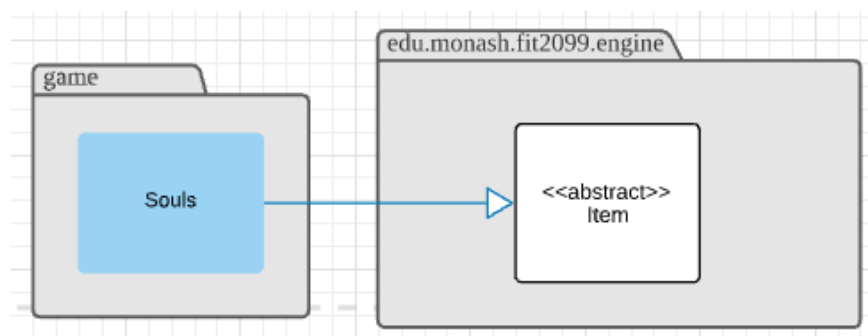


Figure 4: Class Diagram for Souls

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.

Requirement 4: Enemies

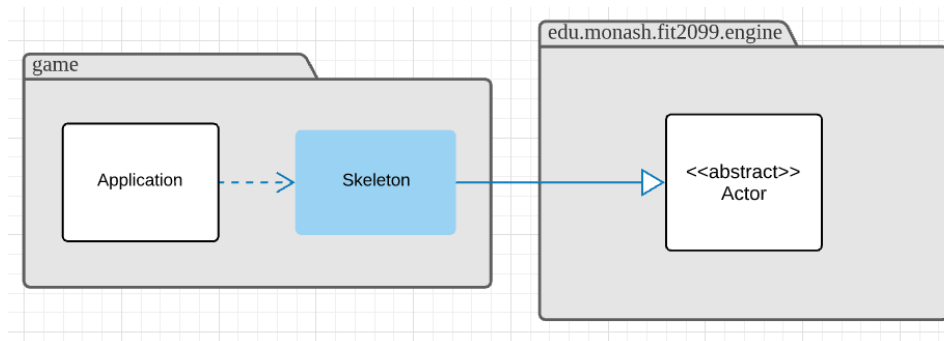


Figure 5: Class Diagram for Enemies

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.

Requirement 5: Terrains - Valley and Cemetery

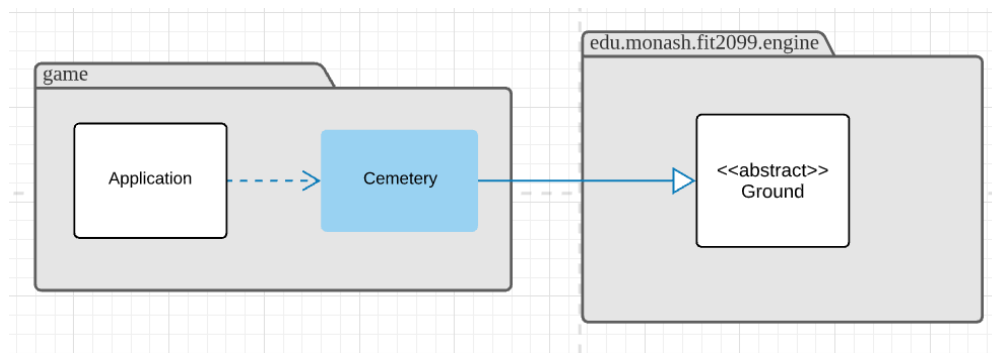


Figure 6: Class Diagram for Terrains

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**)

Requirement 6: Soft reset

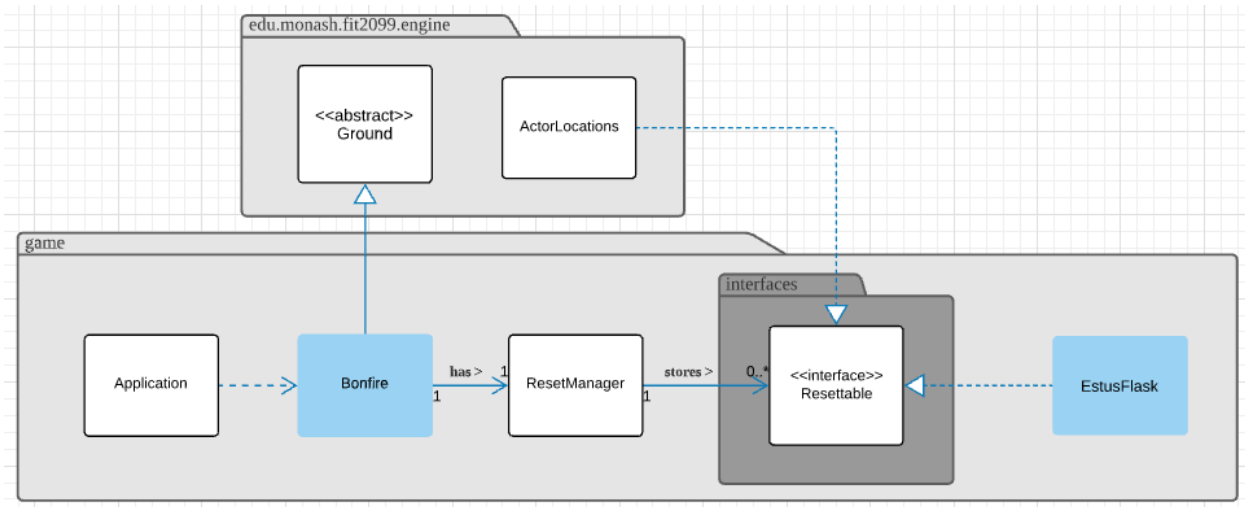


Figure 7: Class Diagram for Soft Reset
Note: Same as class diagram in Requirement 2

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)**

Requirement 7: Weapons

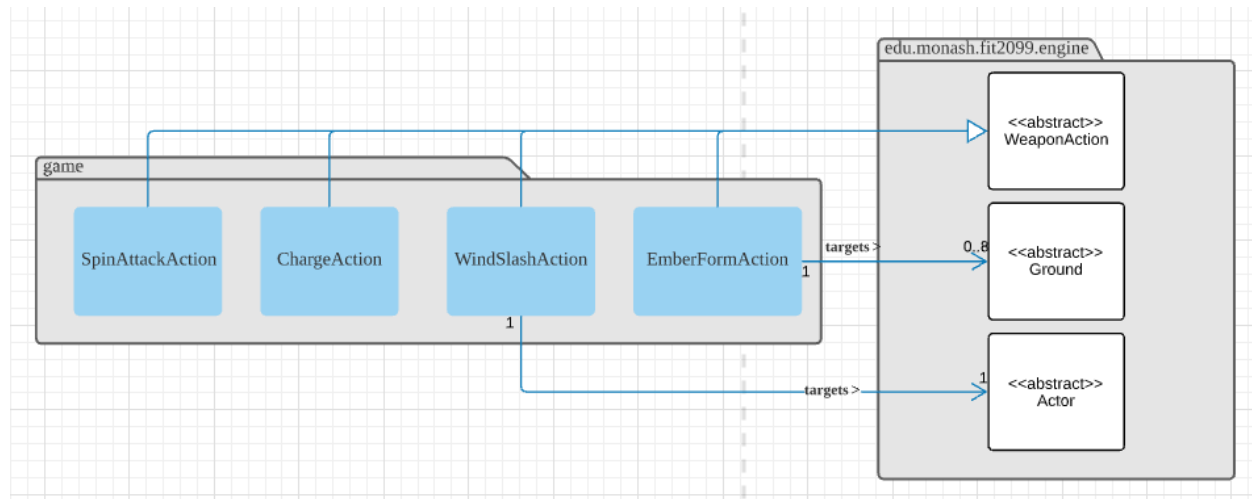


Figure 8: Class Diagram for Actions in Weapons

Since all of the weapons are melee weapons (Sword, Axe), the weapons will thus be grouped into the existing class called `MeleeWeapon` without creating any other extra classes where the weapons can be constructed using this class. Another thing is that weapons have specific capabilities (ability or status) that can be activated by the holder. Therefore, the activation of the capabilities is 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. Speaking of passive skills of the weapons, they can be implemented inside the `MeleeWeapon` class itself as they do not need any action class to activate the skill.

Requirement 8: Vendor

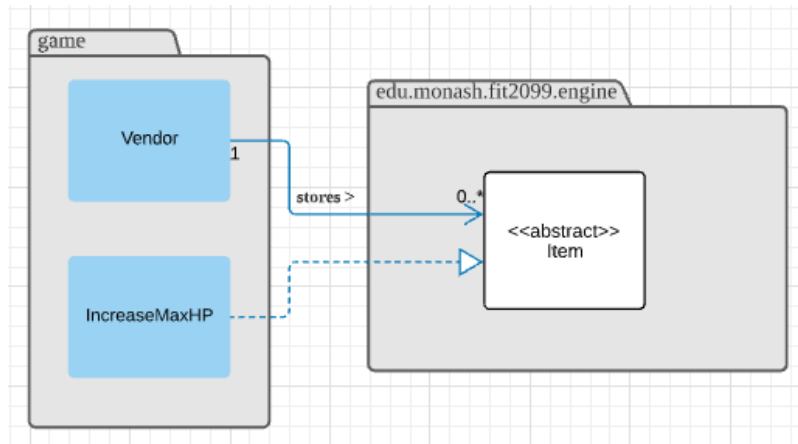


Figure 8: Class Diagram for Vendor

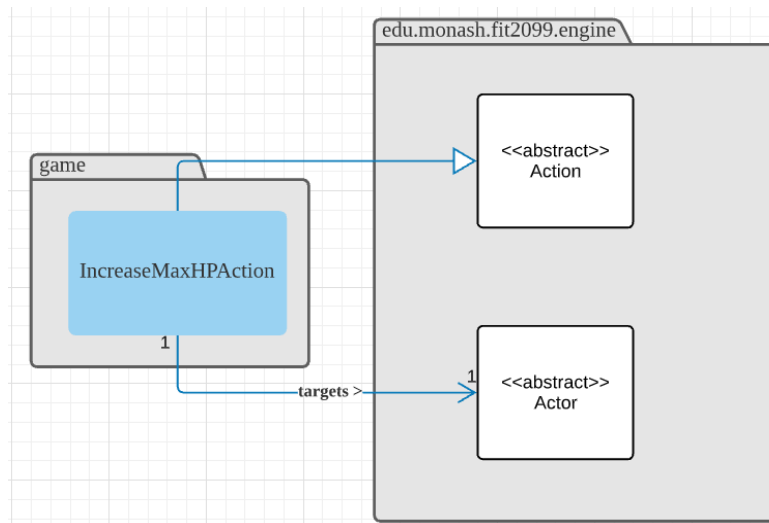



Figure 9: Class Diagram to Increase Maximum HP

A new class called Vendor will be implemented where it stores a list of items where the `getAllowableActions()` will be used to find out what actions can be performed on each item. A new class called IncreaseMaxHP which inherits from the Item class, is also implemented so that the actions that can be performed on this item can be obtained using the same method mentioned above. Alternatively, we can create a Buyable interface that has a similar method of obtaining all the actions available. This allows the classes which implement Buyable to find out the available actions. However, a new class called IncreaseMaxHP still has to be created and the new class will have attributes such as capabilities (which is quite similar to Item class) (**DRY Don't Repeat Yourself design principle**). This method is not appropriate for now since there is



only 1 different type of object and will only be considered when the vendor is given more flexibility to sell many different items and non-item objects.

In order for the IncreaseMaxHP item to work, an additional Action class called IncreaseMaxHPAction is created, while the weapon items in the vendor can be bought using the existing class called SwapWeaponAction.

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 number of Estus Flask will then be decremented using the `decrementNumberOfEstusFlask()` method and a message "a: Unkindled drinks Estus Flask (numberOfEstusFlask/3)" will be displayed.

The figure for the drink action can be clearly seen in Figure 10.

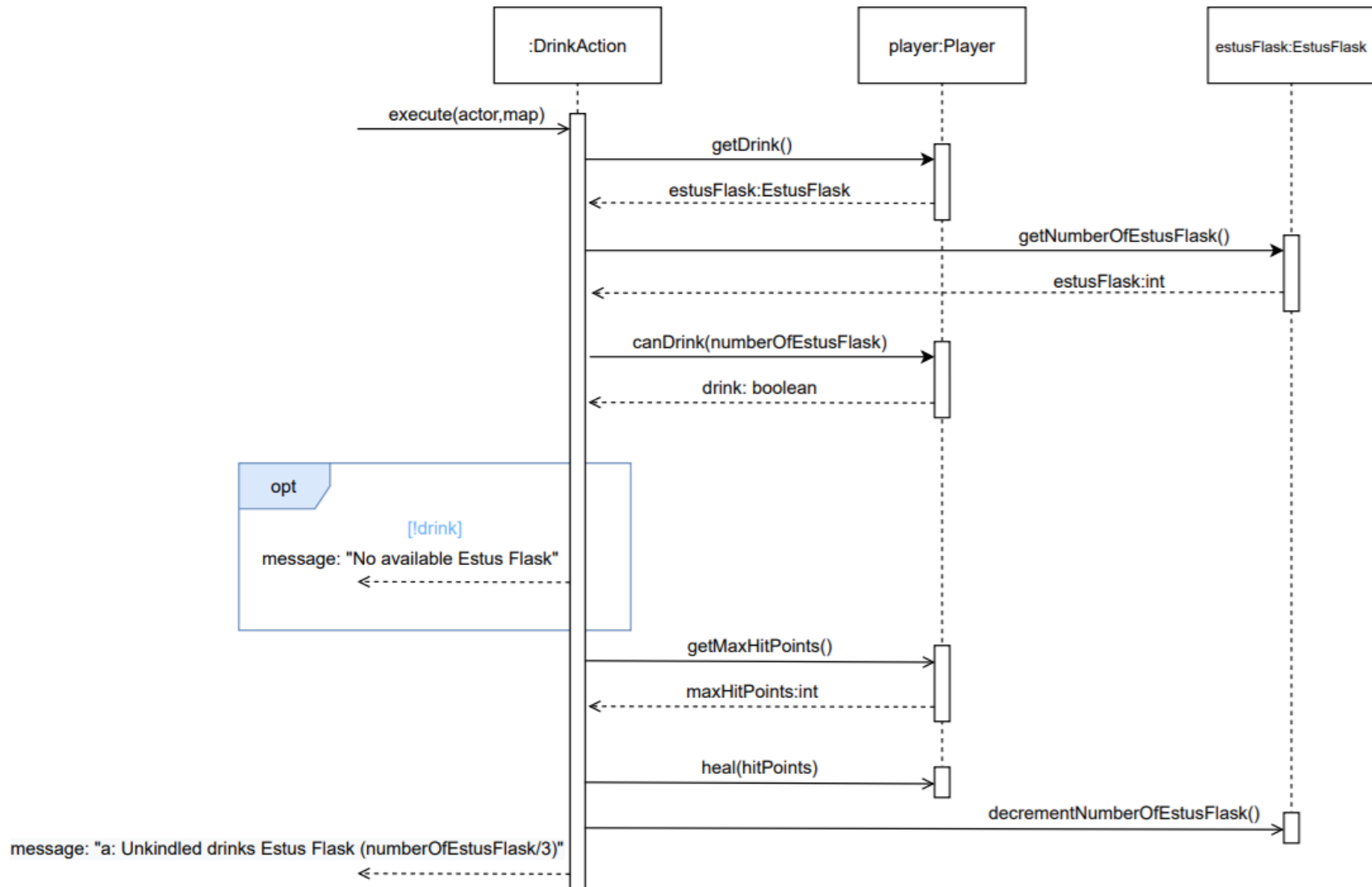


Figure 10: Sequence Diagram for DrinkAction

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 3 item options to choose from: buying a Broadsword, buying a Giantaxe, and buying to increase the player's maximum hit points(HP).

The console menu will have 3 additional options that the player may choose to act on when they are adjacent to the Vendor. When the player chooses any of those options, the Vendor will first check the amount of Souls the Player currency has. If the numberOfSouls is lower than the purchaseAmount required to buy the choses item the players will not be given the item and then notify the player in the console via the message "Not enough Souls", and if they do the process will proceed.

When choosing to buy the Broadsword, the Vendor will subtractSouls() from the Player first. Then it will getMeleeWeapon() from the MeleeWeapon class, which will send back the instance of broadSword that is created back to Vendor. The Vendor will then swap the Broadsword with the Player's current weapon and remove the older one using SwapWeaponAction() and execute(). Finally, it will return a confirmation message saying "You receive Broadsword" to the console.

When choosing to buy the Giant axe, it will follow a similar process to the Broadsword. The Vendor will subtractSouls() from the Player first. Then it will getMeleeWeapon() from the MeleeWeapon class, which will send back the instance of giantAxe that is created back to Vendor. The Vendor will then swap the Giant axe with the Player's current weapon and remove the older one using SwapWeaponAction() and execute(). Finally, it will return a confirmation message saying "You receive Giant Axe" to the console.

When choosing to pay to increase maximum HP, the Vendor will subtractSouls() from the Player first. Then Vendor will send the instance of the Player to the IncreaseMaxHPAction class using the method changeMaxHP(). Inside this class, that Player instance's maximum hit points will be increased by 25, then it will send the new maxHitPoints amount to the Vendor, who will then display the new maximum HP value in the console with the confirmation message "Max hitpoint increased to [the new max HP value]".

The figure for the drink action can be clearly seen in Figure 11.

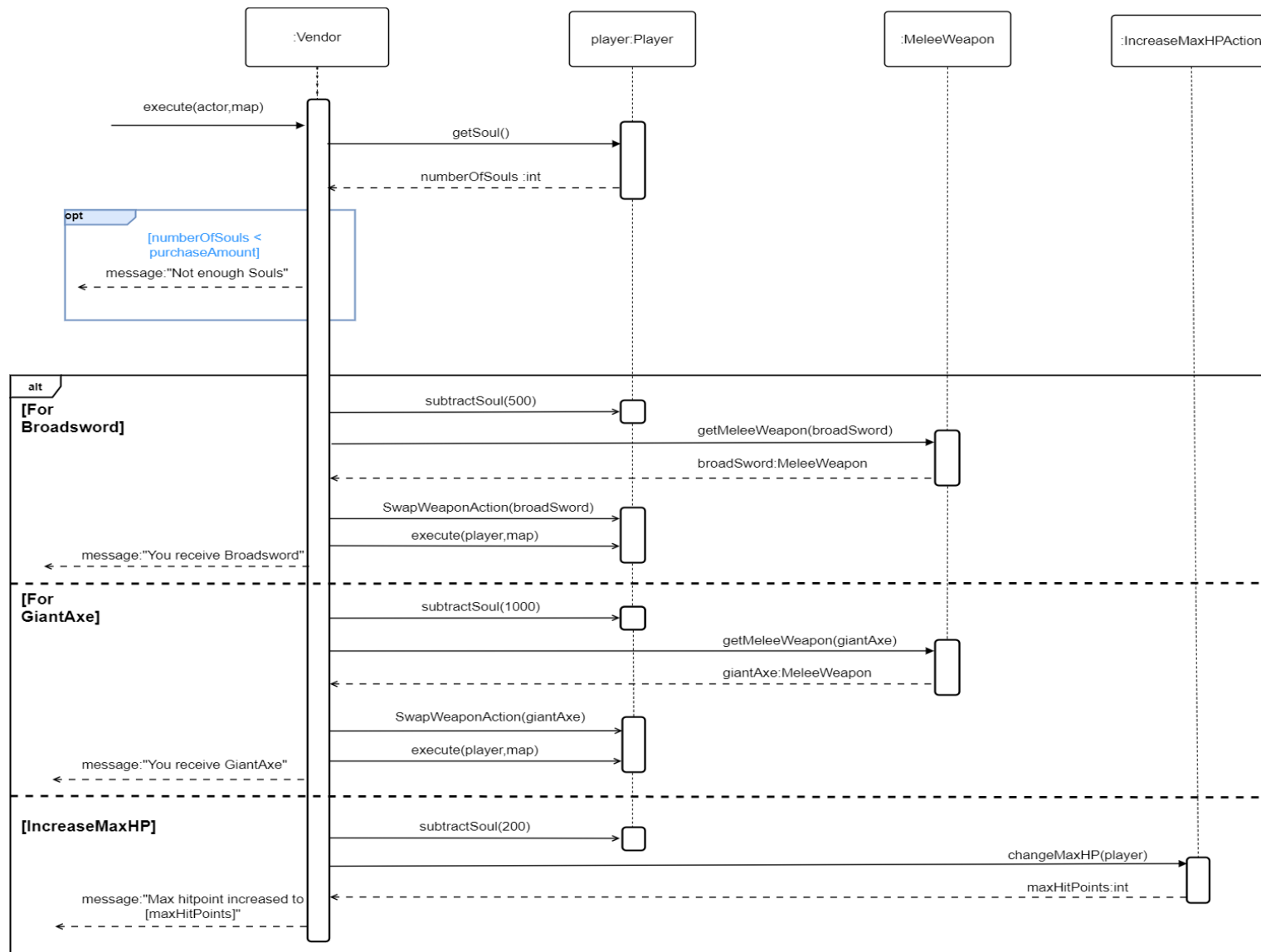


Figure 11: Sequence Diagram for Buying Item at Vendor