Class Diagram

For all the weapons in the game, we realized that they can be classified into two big groups (Sword and Axe). Like what we had mentioned in assignment 1, other than making all the weapons inherits from MeleeWeapon class, we now created two new classes (Sword class and Axe class) to let the weapons to inherit from. This could help us in the future when there are more variety of weapons (but all of them are either sword or axe) so we can just inherit them from the Sword and Axe class and in these two classes we just keep the features that all the sword weapons or axe weapons will share.

In assignment 1, we also proposed that Vendor has an association with abstract weapon class so that the vendor is able to sell the weapons. However, we decided to change the implementation by creating 3 different classes for each item that Vendor is selling, which are BuyBroadswordAction class, BuyGiantAxeAction class and IncreaseMaxHpAction class. Since Vendor inherits the Actor class, he has the getAllowableActions() method. To let the Vendor able to sell the items, we create instances of these three classes and add them under the getAllowableActions(). We designed in this way because this implementation is easier to maintain and we can just simply create more classes and add its instance under Vendor’s getAllowableActions() when the Vendor has new products to sell.

Sequence diagram - RestAction.execute()

The enum class has changed from Abilities class to Status class. Another cleanUp() method of resetManager object is called before looping through the resettable list to clear the enemies killed before taking a rest. This execute() method of the RestAction class interacts with three other classes, including ResetManager, Status, and Resettable. The ResetManager class interacts with the lifeline of this RestAction class through the invocation of its getInstance() method, which returns an object of its own class, named “resetManager”. The enum class Status is involved in this lifeline since REST is passed in as an argument of the run() method of the ResetManager class. The RestAction class’s lifeline interacts with this object by invoking its run() method. Throughout the lifeline of this resetManager object, the self message symbol is used to represent the invocation of its own method, cleanUp(). The loop fragment is used to model the iteration through resettableList. Since each resettable element of resettableList implements the Resettable interface, this interface interacts with the object resetManager’s lifeline.

Sequence diagram - SoftResetAction.execute()

The enum class has changed from Abilities class to Status class. Another cleanUp() method of resetManager object is called before looping through the resettable list to clear the enemies killed before triggering soft reset.All interactions between SoftResetAction and other classes, including ResetManager, Status, and Resettable, are the same as above. The only difference is that SOFT_RESET is passed in as an argument of the run() method of the ResetManager class.

Sequence diagram - Player.resetInstance()

Instead of using the heal() method, the hitPoints of the player is reseted to maxHitPoints. A loop is added to reset the charge counts of the estus flask. Modification of x and y axis of the location is added for REST action since we would like to save the exact location of the bonfire while the player rests at the adjacent squares of bonfire, so that when soft reset is triggered, the player will be spawned at the top of the bonfire. Instead of transferring the souls to the ground, the souls are transferred to a token which will then be placed at the dying location. This resetInstance() method of Player class interacts with three other classes, including GameMap, Location, and Token. The self message symbol represents the invocation of its own attributes, methods or local objects that are from the same lifeline as Player class. Those attributes include “hitPoints”, “maxPoints” and “lastBonfire”, methods include “transferSouls()”, and local objects include “x”, “y” and “location” which are created within this lifeline. The loop fragment is used to model the iteration through this.getInventory(). The optional execution fragments symbolize that if the condition is met, the code within this fragment will be executed, else, continue. The lifeline of Estus Flask class begins and ends within this loop fragment since it interacts with the Player class’s lifeline only when the right item is found to perform downcasting and invoke its resetChargeCount() method. The object of class GameMap which is passed in as an argument and named “map”, interacts with the Player class’s lifeline through the invocation of locationOf(), at(), and moveActor() method. This method returns an object of the Location class, named “location”. The Player class’s lifeline interacts with this location object by invoking its x(), y(), and addItem(token) method. The alternative fragment describes only one of the options, either the status is REST or SOFT_RESET, will be executed. The Token class interacts with this lifeline as its getInstance() method is invoked and returns an object of its class, which is named as “token”. Lastly, the attribute lastToken that is of Location class is involved in this lifeline through the invocation of its removeItem() method. Self message symbol is not used here since we are not only calling on the attribute itself, but the method of the attribute’s type class. Enum class is not presented here.