Design Rationale

Class Diagram:

In our proposed design, we have decided to make all the weapons inherit from melee weapon class that we had found in the code given to us, since all the weapons available at this current stage are melee weapons. However it is possible to make the weapons inherit from the abstract item class as well since they are technically items that the player can equip and that the vendor can sell. We have set the estus flask to inherit from the abstract item class since it is an item in the game that is not a weapon. One other way to implement this could have been to make the estus flask an attribute in the players stats, however this would make changing the effects of the estus flask more complicated.

We created the SoftResetAction class and RestAction classes to manage these actions when they occur and to use them to call the ResetManager class. This would also mean that RestAction and SoftResetAction would be dependent on ResetManager. The Resettable interface is responsible for resetting the Estus flask, player stats and enemy stats.

For enemies, we created an abstract class called Enemies where the more specific types of enemies would inherit from. It is not necessary to have made this class, however we feel that by creating this class we make it easier for adding the same attributes and methods to all enemy types.

As for Valley and Cemetery, those classes will inherit the abstract ground class as they are also different types of terrain that can be anywhere on the map. We then made the existing Floor and Dirt classes interface with the soul interface. This is because when a player dies, a “Soul Token” is created on the ground where the player died and if the player were to fall into the valley and die then the soul token would not spawn in the valley but outside on a floor tile next to it.

Lastly, our vendor simply has an association with the abstract weapons class as the vendor is able to sell some of these weapons. Also, by making it associated with the abstract weapons class we can now easily add more weapons to the vendors shop list.

Sequence diagram - StormRuler.getActiveAction():

We have decided to create a class for each action of the Storm Ruler so there will be a Charge class and a WindSlash class and both classes will extend the WeaponAction class. In the Storm Ruler class, we will have an ArrayList attribute name weaponActions that contains all the action that can be performed by the Storm Ruler (by now it will consist of Charge and WindSlash). When we want to perform the Storm Ruler's action, we will loop through the weaponActions to get the action we want to perform. Since both Charge and WindSlash inherit the WeaponAction class, both of them will have an execute method that takes in an Actor and the GameMap as parameters where Actor is the target that the

WeaponAction will perform to. By getting the Storm Ruler's action (Charge or WindSlash) and call its execute method, we can perform the Storm Ruler active action.

Sequence diagram - RestAction.execute():

This sequence diagram presents the interactions required to perform rest action between objects. We have decided to create a class named RestAction to perform rest features if the player chooses to rest around the Bonfire on the map. During the execution, a reset manager instance is created using the static factory method getInstance(). Then, SoftResetAction class invokes the run() method from the resetManager object to traverse through a list of types Resettable which is an interface class that is implemented by Player class, Enemies class, as well as EstusFlask class. An object that implements Resettable will be added to this list during its instantiation. A value of type enum obtained from the Abilities class, REST, is passed into the run() method as one of the arguments. The resetManager object invokes the resetInstance() method from each resettable element to perform specific reset features depending on the class that the resettable element is from. Next, the resetManager object calls its cleanUp() method to remove non-existing instances in the map from the list. Lastly, a prompt message is returned to inform that the player is resting.

Sequence diagram - SoftResetAction.execute():

This sequence diagram presents the interactions required to perform soft reset action between objects. We have decided to create a class named SoftResetAction to perform reset features if the player gets killed on the map. It has the identical implementation as the ResetAction class, but SOFT_RESET of type enum is passed in as the argument instead.

Sequence diagram - Player.resetInstance():

This sequence diagram presents the interactions occurring when the restInstance() method of the Player class is executed. Firstly, this player instance calls its heal() method and passes the maxHitPoints attribute of its superclass as an argument to refill the player's health back to maximum. Next, the program invokes the method locationOf() from the map argument to retrieve the current Bonfire location that the player is resting at and store it in an attribute called lastBonfire which is of type Location if the provided abilities argument is REST. If the provided abilities argument is SOFT_RESET, the method locationOf() from the map argument is invoked to retrieve the current location that the player dies at. Then, the x and y coordinates of this location are obtained through the invocation of methods x() and y(). The passed in argument, direction, denotes that the player dies at a valley if its value is not null, otherwise it denotes that the player does not die at a valley. If the value is not null, we modify the x and y coordinates to find the opposite location of the corresponding valley object, i.e. if direction.equals("East"), the opposite location should be at the west of the valley. The at() method is invoked from the map argument of GameMap class to retrieve the location using the modified x and y coordinates. After that, we acquire the current ground by calling the getGround() method from the modified location object. If this ground is soul-able, which can be either Dirt or Floor that implements Soul interface, downcasting is executed so that this soul object can be passed into the transferSouls() method as an argument. Consequently, the token of souls of this player will be transferred to this ground after invoking the transferSouls() method. Dirt and Floor are assumed to be the only soul-able ground in the map since it does not make sense if the token of souls appears on walls, valleys, or other terrain types that are included at the current phase. Lastly, this player is respawned at the location stored in the lastBonfire attribute by calling the moveActor() method from the map argument.