Design Rationale

UML Class Diagram 1 - Player

Estus Flask

- At the start of the game, the *Player* will have an Estus Flask instance that can be used throughout the game, therefore an *EstusFlask* class is created.
- Since *EstusFlask* cannot be picked up or dropped the class does not extend from the *PortableItem* class, instead, it extends the abstract *Item* class.
- The *player* can use *EstusFlask* to heal themselves by using the *UseEstusFlaskAction* class. This class cannot exist without an *EstusFlask* instance, so there is a simple association between the *EstusFlask* and *UseEstusFlaskAction*. Furthermore, *UseEstusFlaskAction* class extends from the abstract *Action* class because it is an action the *Player* can perform.

Bonfire

- Bonfire is a location in the middle area of the map where enemies cannot enter, therefore, a new class named *Bonfire* is created to represent the bonfire.
- The Bonfire class extends the Ground because the Player should not be able to pass through it and it is part of the map, not something that can act or move.
- The application class creates an instance of the Bonfire class when the game is started, therefore Bonfire is dependent on this class.
- When players rest at the bonfire, all their health, Estus Flask charges and enemy attributes will be reset. Therefore, a *UseBonfireAction* class is created which extends from the abstract *Action* class for the *Player* to perform this reset feature. *UseBonfireAction* has a simple association with the *Bonfire* class and the *ResetManager* class since when the *Player* executes this action all the following objects stated above will be reset.

UML Class Diagram 2 - Enemies

Enemies (Undead, Skeleton and Lord of Cinder)

- When the *Player* kills enemies, the *Player* will gain a certain number of souls from the enemies, so there is a simple association between the *Player* and enemies that shows the enemies will give the souls to the *Player* when they die.
- Enemies implement the *Souls* interface because they store souls and give them to the *Player* when they die.
- The *Player* also implements the *Souls* interface to buy from the *Vendor* and to give the souls to the Token of Souls when the *Player* dies.
- When the *Player* is in the surroundings of the *Undead* and *Skeleton*, they will follow the *Player* and start to attack the *Player*. Therefore, there is a simple association between the *FollowBehaviour* and *Undead* and *Skeleton*.

Ground (Valley and Cemetery)

- For the *Valley*, when the *Player* steps on it, the *Valley* will instantly kill the *Player*, so we created a new Action class *KillPlayerAction* to show the *Valley* has an action to kill the *Player* when the *Player* is next to a *Valley*.
- For the *Cemetery*, it has a chance to create instances of the *Undead* enemies around it every turn, so there is a simple association between the *Cemetery* and *Undead*.

Player

- When the *Player* dies, the number of souls will return zero and the token of souls will appear at the location of death and store the number of souls that the *Player* previously had. Therefore, we created a new class *TokenOfSouls* to store the number of souls when the *Player* died at the location of death. Then, when the *Player* is next to the *TokenOfSouls*, the token gives the *CollectTokenAction* to the *Player* and when the *Player* executes that action, the souls that were in the token are given back to the *Player*, the

TokenOfSouls instance is destroyed, and the *Player* is moved to where the *TokenOfSouls* location was.

- Besides, when the *Player* dies, the game will execute a soft reset, so each of the *Actor* will be reset which is similar to the *UseBonfireAction*. Hence, there is a link between the *Actor* and *Resettable* because all Actors can be reset but not all Actors have to be reset.

UML Class Diagram 3 - Weapons

Weapons

- There are several weapons the *Player* can use throughout the game. Each weapon has its own respective values for the price, HP and success hit rate attributes. Thus, *BroadSword*, *StormRuler*, *GiantAxe* and *YhormGreatMachete* classes are created which extend the *WeaponItem* class. This also ensures that weapons can be instantiated with their default attribute values when they are sold by the *Vendor*.
- Each weapon also has its own specific set of skills, therefore a new package called *skills* is created containing all the skills classes for better categorization and organization.
- Each skill extends the *WeaponAction* class as they are all an action a weapon can perform.
- Each weapon has its own set of skills, therefore each weapon class has a simple association with their respective skills classes.

UML Class Diagram 4 - Vendor

Vendor, BuyAction and Exit

- The *Vendor* represents an *Actor* in the game in which the *Player* can interact to trade souls for weapons. Therefore, a *Vendor* class is created which extends the *Actor* class.
- The *Application* class creates the instance of the *Vendor* object, so it is dependent on the *Application* class, which is the main class for the entire game.
- The *Vendor* performs the action of selling and the *Player* performs the action of buying or exiting. If the *Player* chooses to buy, then a subset of actions will be represented with specific buying options. This was done to reduce a cluster of actions presented to the *Player* when the *Player* just wants to pass by the *Vendor* without buying anything.
- *BuyBroadSwordAction*, *BuyGiantAxeAction*, *IncreaseHPAction* and *ExitBuyAction* classes were created as extensions of the *Action* class, as they are actions the player can perform once they have executed the BuyAction. Therefore, each class has a simple association with the *BuyAction.*
- To ensure that weapons have default attribute values, instances of the *GiantAxe* and *BroadSword* classes are created when they are bought by the *Player*.

UML Interaction Diagram 1 - BuyAction()

- When the player executes the *BuyAction*, first removes the move capabilities and gives the buying capabilities to the player.
- The player can choose between several weapon buying options, or increase their HP.
    - Player chooses to buy a Broadsword (same for Giant Axe):
        - If the player has enough souls, then they are able to purchase the weapon. A new instance of the weapon class will be created which will be returned to the player and added to their inventory. A message will then be outputted saying “Player buys Broadsword”, to indicate that a weapon has been purchased.
        - If the player does not have enough souls they are not able to purchase a weapon. A message will then be outputted saying “Not enough souls”, indicating that the purchase has not been made.
    - Player chooses to upgrade their HP:
        - If the player has enough souls then they are able to reset their HP. Player is an extension of the *Actor* class which has a method called inreasePlayerHP(). This method is called and the player's HP is increased.
        - If the player does not have enough souls they are not able to increase their HP. A message will then be outputted saying “Not enough souls”, indicating that their HP has not been increased.

UML Interaction Diagram 2 - UseBonfireAction()

- When the player executes the *UseBonfireAction*, it first creates an instance of the *ResetManger* and then calls the run method that will go through all Resettables and call *resetInstance()* on all of them.
- This will make each instance in the map reset depending on its class. For example, the *Player* instance will be fully healed and the Estus Flask is refilled, the *Skeleton* instances will be reset to their initial positions and their HP is set to max and *Undead* instances will be completely removed from the map. *Yhorm* and *Vendor* classes do not do anything for their *resetInstance()* methods.
- After all the instances have been reset, the *ResetManager* instance is destroyed and the UseBonfireAction prints out a message: "Player rested at bonfire."