Assignment 1 & 2 Design Rationale
Lab 7 Team 3
Jingyun Geng, Tawana Mbaya, Yee Lim

Our design implementation can be split into the following 9 categories and subcategories. Each category will contain details on how and why we have chosen to implement concepts in a particular way.

1. Player and Estus Flask

To display the hitpoints of the character, a new method will be added to the player class that, when called, returns a string of the players name and health in the format (Name 100/100), this method will then be called within the playTurn method and its value will be printed to the console. It is done this way so that the current hit points of the character are updated each turn in case they change.

In order to implement the estus flask. A new class called DrinkEstusFlask needs to be added which extends the action class. This class will have the number of drinks remaining as an attribute as well as methods which will set it's menu description, the starting number of flasks and the character that the player inputs in order to perform the action. In addition to this the class will also contain methods which check if the player is out of estus flasks and notify the user in the case that they attempt to perform the action with 0 flasks left. There will also be a method used to reset the number of flasks to the original amount. The final method in the class will be it's execute() function which when called will consume an estus flask and raise the player hit points by 40% of the maximum hit points. By implementing the estus flask as an action it makes it simple to add to the menu without the need to make many new modifications.

Some changes were made to the original design during implementation. Firstly the DrinkEstusFlask was separated into a drink action and an estus flask item. This allows the estus flask to handle its own execution and ensures that the drink class is able to be scaled as more drinks are added to the game

2. Bonfire

The Bonfire is displayed in the game as the letter "B" and is where the player first starts the game (Firelink Shrine). The Bonfire only has one action "Rest at Firelink Shrine Bonfire". Once the player is within the radius of the Bonfire, It will print the options in the console and if the player chooses to rest, it will reset the player's health/hit points to the maximum, refill estus flask to maximum charge and lastly resets the enemies' position, health and skills.

This can be implemented by having a method resetHealth() in the Player class or ResetManager where it will reset the player's health/hit points to its initial health/hit points. Similarly we can create a method resetEstusFlasks() to reset the DrinkEstusFlask count. resetUndead(). resetSkeleton(), resetYhorm() can be implemented in the classes containing the enemies (Undead, skeleton,Yhorm) and their original positions and skills. It will wipe out all undeads from the map. Hence once the "Rest at Firelink Shrine Bonfire" is selected, it will call ResetManager which will contain a method that calls all these methods and it will reset all its features. By implementing in this way, making the bonfire class have an association relationship with the classes of enemies, DrinkEstusFlasks, and Player class. We can avoid having to repeat ourselves in the bonfire class. It also means that each of the classes are

responsible for implementing changes to their own attributes which can avoid having multiple classes changing attributes and reduce messiness in the overall implementation.
Classes used: DrinkEstusFlask, ResetManager,Player, Application, Menu, Undead, LordOfCinder
Classes created: Bonfire

Changes to Bonfire implementation : Since the reset enemies position, health and skills is now optional, I will no longer be implementing functionality resetting the enemies to their original location and thus the reset[Enemies]() methods in Bonfire anymore. The functionality of resting at Bonfire now only contains refilling the player's Hitpoints and refilling the Estus Flask to maximum charges. To implement Bonfire functionality I created a BonfireResetAction class that extends from Action class. BonfireResetAction contains the execute method that resets the health and Estus Flask count. Then it will return a string indicating the status of health and Estus Flask. The menuDescription method returns the string that is to be printed to the console to the user to choose from. Finally I decided to implement the BonfireResetAction class methods separated from Bonfire class because then for future implementations, if we were to add more options to Bonfire, we can just add it to the array of actions that Bonfire is capable of doing instead of modifying a large part of Bonfire class. I no longer require ResetManager as I am not resetting any enemies anymore.

3. **Souls**

We can make a static final variable updatedSouls (initialised to 0) which contains the number of souls the player has at any given time. We can keep track of this by having a getSouls() method in the class that will return the current number of souls of the player. We will increase the updatedSouls only when the player has slain enemies, hence we can make a method increaseSouls(enemy), when the parameter is which enemy is slain. Then it will increment updatedSouls depending on the enemies. Lastly we will have resetSouls() for when either the player has died or starting over. By making the variables static final, we are avoiding excessive use of literals and not repeating the same integers multiple times.
Interface used: souls
Class used: Player
Class created: Souls
Changes to Souls implementation: Instead of creating a whole new class containing the functions of souls, I have just implemented the methods requiring souls. So when needing the current soul count of the player, it is much easier to just do player.getSouls() instead of player.souls().getSouls(). It reduces the need for dependency on the player class. Instead of creating a method increaseSouls(enemy) that is specific for when enemies are slain, I implemented the methods substractsSouls(int souls) and addSouls(int souls) that can be used for general increasing and decreasing souls. To implement the increase of souls when enemies are slain, I will use the enemies class that directly increases the souls from that class. It is much better in terms of further implementation if we are to add more enemies being slain, instead of altering player and enemies class we can just add a new enemies class.

4. **Enemies**

For enemies, there are two kind of normal enemies, Undead and Skeleton, these two classes both extends Actor and implements Soul, in the constructor of undead, there are

name undead, displaychar “u”, with hit point, in the playTurn method includes detect player, follow player, attack player and wander actions, it will loop through the actions. Override the hurt method, apply the damage when it is the target and 10 % chance instantly die.
For the skeleton, the constructor will contain a name, display car “s”, hit point 100 and randomly generate a weapon between broadsword and giant axe. The play turn method is the same as undead, add a revival method which will be checked in AttackAction to give it 50% chance to revive.

5. **Yhorm the Giant (Lord of Cinder)**

The design of Lord of Cinder is similar to other enemies with weapon Machete, but one more method called isEmberForm to check hp of it, in the play turn, it will call isEmberForm if returns true it will add a new action BurnGroundAction.

6. **Terrains (Valley and Cemetery)**

In order to implement the valley class, first the method canActorEnter() in the class valley must be modified to allow players to enter. Next the playTurn() method within the player class would need to be modified to include a check if the player moved this turn and if so, check the class of Ground the player stepped on. If the class is a valley, then a soft reset is performed. By doing it like this we ensure that code regularly checks the player location and functions properly without the need to modify much of the existing code.

The only changes made to the implementation of the valley class is that instead of the player’s playturn method checking if the player has fallen into the valley, this job is done by the valley itself. This would prevent the player class from becoming cluttered in the case that more special ground types were added. It also better follows the principle of classes managing their own data.

To create a cemetery a new class needs to be created named “Cemetery” which extends the ground class. Within the constructor whe character by which cemeteries can be identified will be determined. The method will also contain a method spawnUndead() which generates a random number from 1-4 and if the number is 2, an undead will be spawned in one of the locations next to the cemetery thereby making it a 25% chance that one is spawned.

7. **Soft Reset/Dying in the game**
    - **Token of souls (extra feature)**

In order to implement a soft reset the player class must be modified to implement the “Resettable” interface, The method resetInstance() then needs to be added which stores the original location of the player with the variable “startLocation”. The method will check if the current player hit points are <=0 using the isConsious() method. If they are, the method will read the current location of the player,reset player health , fill the estus flask charges, and update the players current position to the position of the firelink shrine bonfire.

Some changes made to this when implementing would be that the player location is not stored within the resetInstance(). Instead the player is relocated within the if statement after isConsious() is false.

A new class called TokenOfSouls will also be created which extends PortableItem. The class will have an attribute “souls” which is used to store the number of souls the token holds. Upon death, if the location of the player is not a valley, an instance of the class will be created and the token will be placed in the location that the player died. If the player was in a

valley then the location to the right is checked, continuously until a non-valley location is found where the token can then be dropped.
Implementing the soft reset and token of souls in this way is beneficial as it maximises the use of existing code through the use of interfaces and inheritance, thereby reducing the need to repeat ourselves.
The changes made to this implementation are that Token of souls extends item instead of portable item, ensuring that the player does not have the option to drop it. The logic for placing the token in the case that a player fell into a valley was also changed because situations may arise where the previous logic would not be possible, such as if a valley were located at the edge of the map.

8. Weapons

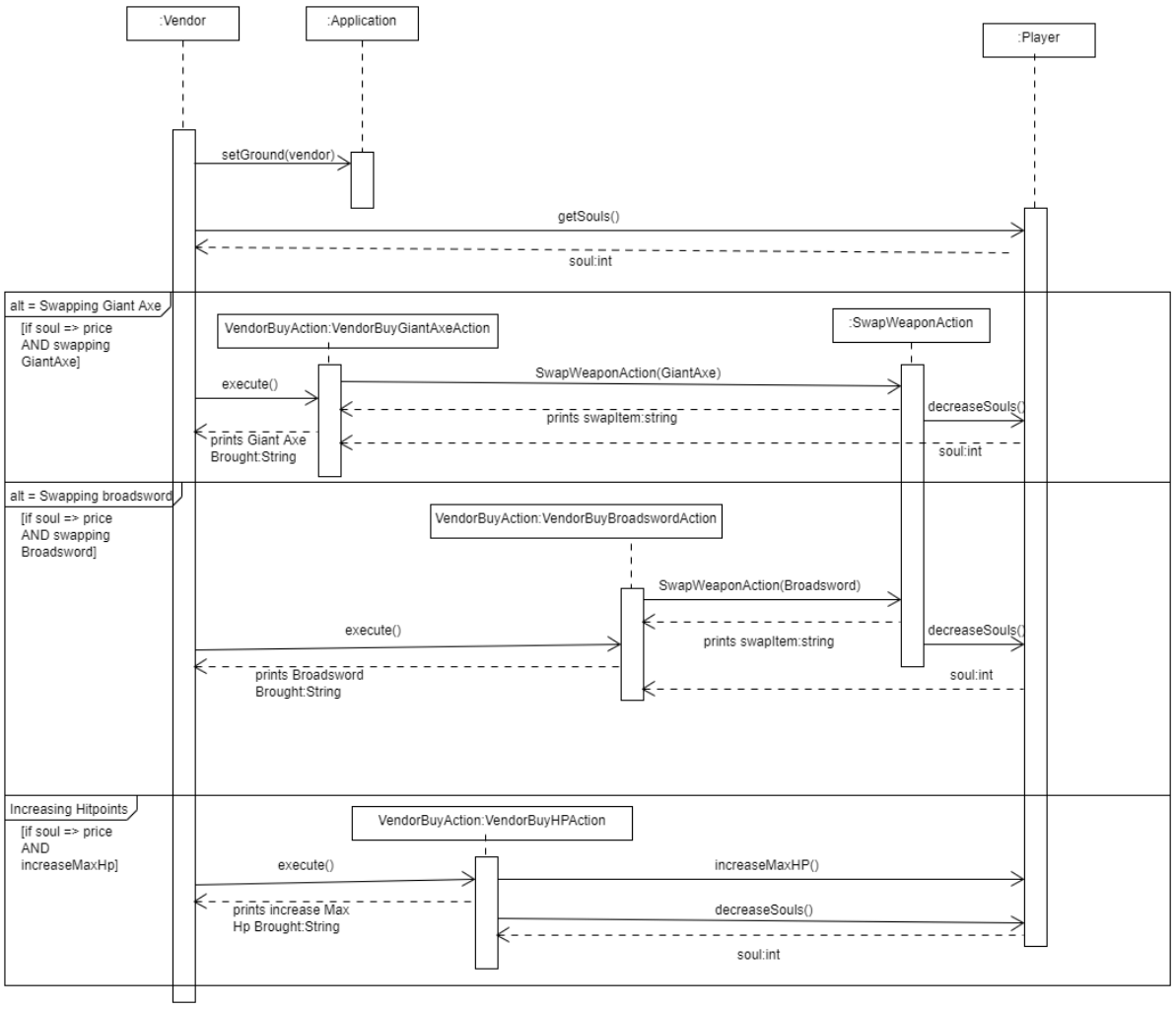
For weapons design all the weapons will extend GameWeaponItem with name, damage, verb and hit rate, and a cost method returns the cost of souls to the vendor.
Broadsword will override the damage with 20% chance to double the damage.
Giant Axe will override the getAllowableActions method and add SpinAttackAction to actions which can attack the enemies/player in 9 locations near the actor who is using it.
Machete will override chanceToHit which will call isEmberForm first to check the hp of Lord of Cinder, if it's below 50%, it will increase the hit rate.
StormRuler will override the getAllowableActions method and add StormRulerChargeAction to actions which will call charge method in StormRuler class to charge the weapon, there's another action for this weapon called StormRulerStumAction which will be execute in AttackAction, in the execute method inAttackAction, it will check if the player is holding the StormRuler and it's fully charged, this action also call the charge method in StormRuler which check the charge time of StormRuler and change the damage and hit rate also change the attack description.

9. Vendor

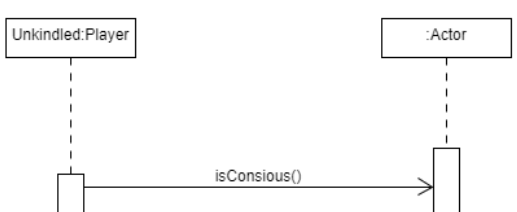
Souls are to be traded to the vendor to purchase weapons and upgrade the player's attributes. The vendor, called Fire Keeper, will be situated at one location for the entirety of the game, and once the player gets within its radius it will display a menu containing all the options available to be purchased. Using the console, when the player chooses an option, vendor class will call the souls class and if getsouls() > price of the chosen option. The vendor class will replace the weapon of the player by calling SwapWeaponAction class and changing to the chosen weapon with its new abilities and attributes. If the player were to choose the increase Maximum HP option, it will call the player class where it will have an increaseHealth(int number) method. We can create a static final variable increaseHP = 25, and call this method, the player class will increment the players hit points to 25. By making all the integers (prices) static final variables, we can avoid excessive use of literals. Also by calling the player class and weapon class it makes each of the classes responsible for changing their own attributes. Vendor will have an association relationship with SwapWeaponAction, Player, GameWeaponItem and Application. As The Vendor class will create attributes of the classes and require the methods in them.
Class used: SwapWeaponAction, GameWeaponItem, Application (to input location of vendor), display, Menu
New class: Vendor
Changes to Vendor implementation: instead of calling the now deleted souls class for getsouls, I implemented the getSouls() method in the Player class. Instead of the method

being called increaseHealth(int number) in player, the correct method is “increaseMaxHp(int number), this is more specific and differentiates the heal() method and increaseMaxHp() method from each other. Similarly to Bonfire, I extended the action classes to make VendorBuyAction which then is extended by again to make VendorBuyHPAction, VendorBuyGiantAxeAction and VendorBuyBroadswordAction. The VendorBuyAction class is the parent class of all the actions that Vendor wants to sell. It also serves the purpose that if any of the VendorBuy[Item]Action classes malfunctions, VendorBuyAction will print that the player does nothing instead of crashing. The reason why I didn’t implement the actions using resetManager was because in the case we want to delete the particular action of the Vendor, it is much more future proof to delete the single class then deleting the class AND modifying the resetManager class. It is better to have the implementation of a particular action in one situated place instead of splitting it into 2 parts and it would be much easier to read and modify for future purposes.

Below is the Uml sequence diagram for Vendor()



Below is the UML Sequence Diagram for the Soft reset.



UML Class Diagram