DESIGN RATIONALE

The overall design choices are made keeping in mind the extensibility of the game. The details of the design choices for each requirement are as follows-

Requirement 1: Player and Estus Flask

Design Choices:

1. **Display Health/Hit Points**
   To display the health/ hit points of the Player, we can implement code in the pre-existing playTurn method of the Player class so there is no need for a new class. The attribute of hit points is already found in the Actor class so a new health system doesn't need to be implemented. We only need to display it. This can be done by calling the methods of the Display class.

2. **Hold Estus Flask:**
   The player needs to hold an Estus Flask, for this feature, we can create a new EstusFlask class that extends the Item class with private attribute quantity. We need to extend the Item class so that we can use it as an item and print out its attribute(number of charges) in the console.

3. **Cannot Drop Estus Flask:**
   The player cannot drop the Estus Flask so we can override the pre-existing DropItemAction method to return null. We need to override the DropItemAction so that the player is not allowed to drop the Estus Flask. This will be a good design choice as we can make use of the pre-existing system to fulfill this requirement.

4. **Drink Estus Flask**
   The player should be able to drink the Estus Flask, so we can create a new class called DrinkEstusFlaskAction that we will put as an allowable action for the EstusFlask class. We need to create this class so that we can use the execute method of the Action class to access the player's instance and add the health of the player.

5. **Display Charges of Estus Flask**
   To display the number of charges of the Estus Flask in the console, we can implement code in the pre-existing MenuDescription method of the Action class. This will be a good design choice as the pre-existing system is sufficient to fulfill this requirement.

Requirement 2: Bonfire

Design Choices:

1. **Create Bonfire Class**
   The player starts the game at the Firelink Shrine(Bonfire), for this, we can create a new Bonfire class that extends the Ground. We need to extend the Ground class so that we can access the methods and private attributes that are already present in the Ground class.
   We do not extend the Actor class as it does not have most of the attributes an Actor should have, like isConcious, hurt, etc. However, since BonFire is a Ground object, we cannot add it into the map using methods in GameMap. Hence, we need to modify the initially given map so that there's a Bonfire (displayed as B) on the map.

2. **Create ResetAction class**
   To perform the Rest at Firelink Shrine action, we can create a class named ResetAction that extends Action and has a private attribute of the class ResetManager. The purpose of creating this new class is so that we can get the Map as a parameter which will allow us to access all the actors in the map and check if they are resettable. Then, it will only reset those actors who are resettable to their original position, while others will be removed from the map. Another advantage of extending the Actor class is we can use methods in the Action class too like menuDescription if we want to.

Requirement 3: Souls

Design Choices:

1. **Create new method rewardSystem at AttackAction class**
   When the player slays/kills enemies, the player gains a certain number of souls from them.
   We have implemented this feature inside the pre-existing AttackAction class along with a helper function named rewardSystem. When the Player kills a target, the helper function is then called to get the appropriate rewards for the particular slain enemy, which is then handled in the AttackAction execute function. This function will then use the methods in Player class to increase the souls of players.

2. **Override methods of Soul class in Player class**
   Methods like addSouls and subtractSouls need to be overridden so that player's souls can increase or decrease after implementing the methods.

Requirement 4: Enemies

Design Choices:

1. **Enemies are not allowed to enter Bonfire**
   To prevent enemies from entering Bonfire, we have chosen to use the pre-existing game engine to achieve the desired result. In the existing game, there is a method named canActorEnter under the Ground class which we have modified to check if the user has the capability to enter the floor.

2. **Enemies will attack if the Player is in the surrounding tiles.**
   To implement such a feature, we have decided to override the getAllowableActions method found in the Player class to have an AttackAction. This allows all Enemy Classes to be able to interact with the Player in their own playTurn method.

3. **Enemies will follow if the Player is in the surrounding tiles.**
   In the pre-existing game engine, there is already a Behavior system implemented along with a class named FollowBehavior. With both of them, we will be able to achieve the desired result by just adding the FollowBehavior to the particular enemy class instance when the Player is found in the method playTurn.

4. **Enemies can Randomly attack using active skills**
   To allow enemies to have a chance to use their weapon's active skills, we have decided to run a random check when an AttackAction is present in the parameters given in playTurn(Refer to Requirement 4.2 above), given that the random check passes, a weapon skill class (Action Class made for each specific active skill) will be executed instead.

5. **Display hit points, maximum hit points, and the weapon equipped by the enemy**
   In the lifetime of the game, all instances of enemy classes will be in either of the Behavior (WanderingBehavior and FollowBehavior) that is in the pre-existing game engine. By changing the menuDescription of the WanderingBehavior class, we can create specific messages based on the actor affected. FollowBehavior is omitted as it is only used when in combat.

6. **Undead Class**
   Undead Class is used to create all instances of the enemy Undead that is going to be present in the game. The Undead Class will be extended from the class Actor, which will have all the attributes needed to fulfill all the basic stats of the actor (such as Maximum Hit Points, Inventory, Weapon stats, etc).

   a. **25% to spawn from Cemetery**
   Under the Cemetery Class, the tick method will be overridden to do a random check every turn where the random check passes, a new instance of the Undead Object will be created.

b. 10 % chance to die instantly - Create DieByChanceAction class

At every turn, a random check will be done inside the method playTurn. If the check passes a condition, then the method playTurn will return a new DieByChanceAction object. Then, this Undead object will die and be removed from the map. Besides that, we can use menuDescription to display a message that the Undead Object was dead.

7. Skeleton

Skeleton Class is used to create all instances of the enemy Skeleton that is going to be present in the game. The Skeleton Class will be extended from the class Actor, which will have all the attributes needed to fulfill all the basic stats of the actor (such as Maximum Hit Points, Inventory, Weapon stats, etc).

a. 50% chance to resurrect on the first death

The Skeleton Class will have a private attribute firstDeath of type boolean which will be used to keep track of the number of times that particular instance of Skeleton has resurrected.

On the first death, there will be a condition check in the AttackAction Class to check if the Skeleton has already been resurrected once. In the case that it has not been resurrected once before, a random check will be done where if passed the Skeleton's hit points will be amended to the same as its max hit points.

8. Yhorm the Giant

LordOfCinder Class is used to create the instance for the first boss of the game YhormTheGiant (a.k.a. Lord Of Cinder) that is going to be present in the game. The LordOfCinder Class will be extended from the class Actor, which will have all the attributes needed to fulfill all the basic stats of the actor (such as Maximum Hit Points, Inventory, Weapon stats, etc).

a. Ember Form

i. Enrages and uses weapon's active skill when hit points are dropped below 50%

When the hit points of the boss Yhorm the Giant drop below 50%, a method named increasedStats will be called in the class to increase the hit rate of his Weapon Yhorm's Great Machete to 90% (60% + 30%).

ii. Burns surrounding tiles

A new class named EnragedBossFollowBehavior that extends from FollowBehavior will be created. The main reason why our team has chosen to extend the FollowBehavior Class instead of implementing the Behavior interface is that the FollowBehavior Class contains most of the attributes and methods that are useful for the implementation of EnragedBossFollowBehavior. This behaviour class object will be added to LordOfCinder class.

In this Class, the method getAction will change all the surrounding tiles using the class BurningGround that extends from Ground. Previously added BurningGround will also be removed when turnCount reaches 3 which will be incremented each round using the tick method from the Ground class.

iii. **Drops Cinder Of a Lord item when slain**
To implement this feature, we will be creating a new CindersofALord class that extends the Item class. Then, it will be initialised as an object in YhormTheGiant class and it will be added to the inventory.

Requirement 5: Terrains(Valley and Cemetery)

Design Choices:

1. **If player steps on the valley, it will be instantly killed**
For this feature, we modify the canActorEnter method of the Ground class so that the player has the capability to enter the Valley. We will also override the tick method to kill the player if it steps on the Valley. Then, the reset features will be triggered and executed. (See Requirement 6 below)

2. **Cemetery has a 25% success rate to spawn/create Undead**
For this feature, we will create a new Cemetery class that extends the Ground class. We need to extend the Ground class so that we can access its methods and private attributes. To create Undead, we will override the tick method to do chance summon.

3. **Place several cemeteries on the map.**
For this feature, we will call instances of the Cemetery class in the Application class to add them to the map.

Requirement 6: Soft Reset/ Dying in the Game

Design Choices:

1. **Print the classic Souls game phrase when the player dies/gets killed**
To implement this feature, we will modify the execute method of the AttackAction class to check if the player is alive by adding some lines of code that do the checking. If the player is dead, a message will be printed in the console. This will be a good design choice as we will be making use of the pre-existing system to fulfil this requirement.

2. Execute RESET features through ResetAction class

Please refer to Requirement 2.2 for information about the ResetAction class.

3. Token of Souls appears at the player's dying spot/location

For this feature, we will create a new class called TokenOfSouls which extends the Item class and has private attribute numberOfSouls. On Player's death, the execute function in AttackAction will be handling the spawn of the Token of Souls to the place where Player has done his/her last MoveAction.

4. The player can get lost souls only by interacting with the Token of Souls

For this feature, we will create a new class called RetrieveSoulAction that extends the Action class. We will then put it under allowable action for TokenofSouls. This allows the user to interact with the item and retrieve all the lost souls from the last game.