DESIGN RATIONALE

The overall design choices are made keeping in mind the extensibility of the game.
The changes that have been made from the assignment 1 design rationale have been highlighted in yellow.

The details of the design choices for each requirement are as follows-

Requirement 1: Player and Estus Flask

Design Choices:

A new Player class was not created as the pre existing system is sufficient to fulfil all the requirements. To display the health/hit points of the Player, we will **call the println() method of the Display class** in the pre existing playTurn method of the Player class. Using pre-existing methods will prevent repeating ourselves and we can reuse old code **(DRY Principle)**.

A new Estus Flask class was created which extends the Item class (Inheritance). 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. The Player initially holds the Estus Flask, so we will create a new instance of the Estus Flask in the Player class and add it to the Player's inventory using the addItemToInventory() method from the Actor class. Since the Player isn't allowed to drop the Estus Flask, we will override the getDropAction() method from the Item class to return null.

We also created **a new DrinkEstusFlaskAction class which extends the Action class** so that the Player can drink the Estus Flask **(Abstraction)**. In the execute() method of the DrinkEstusFlaskAction class, we will first retrieve the EstusFlask instance by calling the getEstusFlask() method. Then we will **check if the Estus Flask has any charges left by using the getChargesLeft()** method and also **retrieve the maximum hit points of the Player by calling the getMaxHitPoints()** method. This method then calls the **heal() method to increase the Player's hit points by 40 % of the maximum hit points**. Finally, it calls the **drink() method from the EstusFlask class to reduce the charges left by 1**. The purpose of creating this class is 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 **(Single Responsibility Principle)**.

We will then override the getAllowableActions method in the EstusFlask class in which we will **add the DrinkEstusFlaskAction to the allowableActions ArrayList**. Lastly, to display the number of charges of the Estus Flask in the console, we will **override the menuDescription method in the EstusFlask class** to display the charges of the estus flask in the appropriate format. **This is a good design choice as the pre-existing system is sufficient to fulfill this requirement.**

Requirement 2: Bonfire

Design Choices:

A new bonfire class which extends the Ground class was created so that we can easily put it on the map while still having access to all of the Ground capabilities **(Inheritance)**. We do not extend the Actor class as it does not have most of the attributes an Actor should have, like isConcious(), hurt(), etc. 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.

If the Player wants to enter the Bonfire, we will first check if the actor is allowed to enter by **overriding the canActorEnter() method from the Ground class**. This method will check if the **Player has the ability to REST**. If this method returns true, the Player will be allowed to enter the Bonfire. Doing this will allow us to ensure that only the Player can the Bonfire and not enemies.

Since Bonfire is a Ground object, we cannot add it to the map using the methods from the GameMap class. Hence, **we will manually add it to the map by modifying the map ArrayList in the Application class**. The Bonfire is represented by the character 'B'.

The purpose of creating a general Bonfire class rather than a specific FireLink Shrine Bonfire is so that different types of bonfires can be included in the game in the future. Doing this will **improve the code's maintainability and flexibility.**

Note: The reset features that can be executed at the Bonfire have been described in Requirement 6: Soft Reset/ Dying in the game

Requirement 3: Souls

Design Choices:

When a Player kills enemies, the Player gains a particular number of souls from them. To implement this feature, we have modified the Player class to implement the Soul interface. In the Player class, **we have overridden the addSouls(), subtractSouls() and transferSouls() methods from the Soul interface**. The purpose of overriding these methods is so that the Player can gain or lose souls depending on the action performed.

We have also added the number of souls that a player gains(when an enemy dies) as a parameter to the constructor of each enemy class (Undead, Skeleton, Lord of Cinder). Then, we created a die() method in each enemy class in which we **call the transferSouls() method to transfer the current instance's souls to another Soul instance**. Lastly, we print a message in the console, which indicates that the enemy has been killed and the Player has gained a particular number of souls. In the Enemies class, we have also overridden the transferSouls() method which has a Soul object as the parameter. In this method, we **call the addSouls() method which we have previously written in the Player class to add the soulReward to the Player depending on the type of enemy that has been killed**.

By reusing the same transferSouls() method each time, we can reduce the amount of code needed to perform this action and prevent repeating ourselves (DRY Principle).

Requirement 4: Enemies

Design Choices:

A new Enemies class was created which extends the Actor class and implements Resettable and Soul (Inheritance). The Enemies class will have 3 instance variables- an arrayList of behaviours that an enemy can perform, initial location of the enemy, and number of souls which the Player gains if an enemy is killed **(Open/Closed Principle)**.

We will also **override the playTurn(), resetInstance(), isExist(), transferSouls(), getAllowableActions() and toString() methods from the Actor class**. Apart from this, we will have an **attackPlayer() action** that checks if the enemy has the ability to attack the Player. If the ability is present, we add the followBehaviour to the enemy using **add()** method. Then, we check if the enemy has any weapon skills, and if so, we add it to a list of allowable actions that the enemy can perform.

Lastly, we have a die method that removes the enemy from the map when the enemy dies by calling the **removeActor()** method. Then, we call the transferSouls() method to give the Player a specific number of souls depending on the type of enemy killed. Finally, it prints a message that says that the enemy has been killed and that the player has gained a specific number of souls.

The purpose of having a general enemies class is so that we can add more enemies in the future if needed. **This helps to improve our code's maintainability and flexibility.**

We have also created 1 other new class which is the Skeleton class. This class is made to extend the Enemies class (Inheritance). The pre - existing Undead and LordOfCinder classes have also been modified to extend the Enemies class. The purpose of extending the general enemies class is so that we can access the methods and private attributes of the Enemies class.

The LordOfCinder class **overrides the playTurn() method of the Enemies clas**s. In this method, we first check if the hit points of the enemy is less than half of the maximum hit points and we also check that the ember form hasn't already been activated. If so, we call the enraged() method(explained below) and print out a message in the console which says that the ember form has been activated. This method also **checks if the Player is near Yhorm the Giant by calling the checkIsPlayerNear() method from the Enemies class**. If so, Yhorm the Giant will attack the player by calling the attackPlayer() method (explained below) to attack the Player.

We also **override the resetInstance() method from the Enemies class** in which **we remove the EnragedBossFollowBehaviour by calling the in-built removeIf() method**. We will then **move Yhorm the Giant back to its initial location by calling the MoveActor() method**.

Next, we will **override the attackPlayer() method from the Enemies class**. In this method, we loop through the array of actions and **find the target for which the Attack action is enabled using the getTarget() method**. Then we will add the EnragedBossFollowBehaviour to the array list of behaviours using the add() and finally **remove the WanderBehaviour using the removeIf() method**.

Lastly, we have the enraged() method. In this method, we will **activate the ember form of Yhorm the Giant by calling the getWeapon() and activateEmberForm() methods**. We will then loop through the list of behaviours of Yhorm the Giant. Then we find the target by checking if the Actor

has followBehaviour. When the target is found, we add the EnragedBossFollowBehaviour(target) to the list of behaviours so that Yhorm the Giant can follow the enemy. The Lord of Cinder is represented by the character 'Y'.

Apart from the classes mentioned above, **we have also created 2 other classes specifically for Lord of Cinder. These classes are EnragedBossFollowBehaviour and BurningGround.**

The EnragedBossFollowBehaviour class extends the FollowBehaviour class while the BurningGround class extends the Ground class.

In the EnrgaedBossFollowBehaviour class, we loop through the exits array list to find all the possible directions that the Actor can move to. If the surrounding ground is an instance of dirt or burningGround, we set the ground to burningGround by **creating a new instance of the burningGround and adding it to the map by calling setGround() method**. Then, we loop through the array list of exits once again and get the destination that the Actor needs to move to by calling the exit.getDestination() method. Then we check if the actor can enter that destination by calling the canActorEnter() method. Then, we check if the distance between the destination and the current location of the Actor is lesser if the Actor is moved to one of the exit points. If so the Actor will be moved to the exit point by calling the moveActorLocation() method.

The BurningGround method mentioned above has only one method called tick() which has been overridden from the Ground class. The purpose of this method is to ensure that the BurningGround lasts for only 3 turns. Each time this tick() method is executed, we increase the turnExisted instance variable(initially 1) by 1. **If the turnExisted instance variable is more than 3, the BurningGround around Yhorm the Giant will be reset to Dirt**. This is done by calling the setGround() method with a new instance of Dirt as the parameter. Lastly, this method checks if there is any actor stepping on the burning ground. All actors other than Yhorm the Giant stepping on the burning ground will lose 25 hit points. This is done by calling the location.getActor() method to get the Player at that specific location and the**n calling the hurt() method to reduce the hit points of the actor**.

The Skeleton holds either a Giant Axe or a BroadSword so we have created a random check in the constructor of the Skeleton class. This is done by **generating a random number in the range of 1-100 by using the in-built Random() method**. **If the random number is greater than 50, the Skeleton will be equipped with a BroadSword else the Skeleton will have a Giant Axe**.

The Skeleton class also overrides the **playTurn() method in which we check if the Player is near, by calling the isPlayerNear() method**. Then, we loop through the actions array list to check if it is possible to use any weapon's active skills by **generating a random number using the Random()**

method similar to what we have done in the constructor. **If the Player isn't near theSkeleton, we return the DoNothingAction()**.

Next, we have the die() method of the Skeleton class which has been overridden from the Enemies class. The purpose of this method is to revive the Skeleton if it hasn't already been received before. This is done by first checking that the Skeleton hasn't already been revived before by **checking that the revivedOnce variable is false** and then **using the in-built Random() method** as done above. A random number in the range of 1-100 is generated and if the number is below 50, the Skeleton will be revived. We then loop through the behaviours array and execute all the behaviours that the Skeleton has by using the Behavious.getAction method(). Else, the **Skeleton will be removed from the map by calling the removeActor() method**. A number of **souls will then be transferred to the Player by calling the transferSouls() method**. The Skeleton is represented by the character 'S'.

The Undead has a 10 % chance to die instantly so **we have created a new DieByChanceAction class which extends the Action class**. This class **overrides the execute() method in which the actor is removed from the map by calling the removeActor() method**.

The Undead class overrides the **playTurn() method in which we check if the Player is near, by calling the isPlayerNear() method**. Then we loop through the list of Behaviours to check that the Undead doesn't have its follow behaviour activated. Then, we once again **generate a random number using the in-built Random() method to generate a random number in the range of 1-100**. If the number is less than 10, **we will call the DieByChanceAction() so that the Undead dies instantly**. Else, we will loop through the list of behaviours and execute all the Behaviour actions. **If the Player isn't near theSkeleton, we return the DoNothingAction()**.

The Undead class also **overrides getIntrinsicWeapon() method which generates a new intrinsic weapon which has a damage of 20**. Lastly, this class also overrides the isExist() method to return false. Undead is represented by the character 'U'.

The purpose of overriding and reusing the methods is so that we can reduce the amount of code needed to perform similar actions. This helps us to prevent repeating ourselves **(DRY Principle).**

Requirement 5: Terrains(Valley and Cemetery)

Design Choices:

We will create a new Cemetery class that extends the Ground class **(Inheritance)**. We need to extend the Ground class so that we can access its methods and private attributes. The Cemetery has a 25 % chance to spawn or create Undead, so we will **override the tick() method from the Ground class**. This method will generate a random number in the range of 1-100 by using the **Random() in-built method**. We will check all the possible locations where the Undead can be added by calling the **getGround()** method to check if the Ground is an instance of Dirt. If so, we will create a new Undead at that location.

We have **placed several cemeteries in the game map by modifying the map Array List in the Application class to have cemeteries instead of ground**. The cemeteries are represented by the character ‘C’.

The Player should get killed when it steps on the Valley, so we will **override the canActorEnter() method to return false**. Similar to what we have done in the Cemetery class, we will also **override the tick() method in the Valley class**. In this method, we will check if the Actor is a Player by calling the **instanceOf() method**. If the actor is a Player, we will reset the Player’s attributes by creating a new instance of the resetAction class and calling the **hurt() and execute() methods**. These methods will restore the hit points of the Player to maximum and reset its location to the Bonfire. The Valley is represented by the character ‘V’.

Making use of pre-existing methods such as tick(), hurt() and execute() will reduce the amount of code needed and prevent repeating ourselves **(DRY Principle).**

Requirement 6: Soft Reset/ Dying in the Game

Design Choices:

The player should be able to rest at the Bonfire, so we have modified the Player class to implement the Resettable interface. We have overridden the resetInstance(), isExist() and registerInstance() methods so that the attributes of the Player can be reset when it dies. The resetInstance() method first checks if the Player is alive by calling the isConscious() method. If the player is alive, it gets the location of the Player by calling the locationOf() method and sets the last location of the player to the current location by calling the setLastSavedLocation() method. If the Player has died, we will reset the number of souls the player has to 0 by calling the subtractSouls() and getSouls() methods. Then we will move the actor to the last saved location by calling the moveActor() method. Lastly, we will reset the charges of the Estus Flask to 3 by calling the getEstusFlask() and setChargesLeft() methods.

We have also created a **ResetAction class which extends the Action class with a ResetManager instance (Abstraction)**. This ResetManager instance instance will execute all the actions needed for a soft reset. Using the ResetManager class for soft resets will prevent any resets from happening accidentally and will help to achieve data hiding **(Encapsulation)**. An advantage of having a separate ResetAction class is that all other classes can perform soft resets by accessing the ResetManager instance. This will help to reduce the amount of code needed to perform the RESET **(DRY Principle)**.

We will also be using the same ResetManager instance to reset the map when the player dies. If the player dies, we will **retrieve the last location of the player using the locationOf() method**. Then, we will **place a new token of Souls at this location using the additem() method**. Lastly, the Player will be moved to the Bonfire and all its attributes will be reset by calling the ResetManager singleton instance.

By using the same ResetAction class to reset the Player’s attribute when the player rests and when the Player dies, we can reduce the amount of code needed and prevent repeating ourselves **(DRY Principle)**.

Requirement 7: Weapons

Design Choices:

We have created 4 new classes (one for each weapon) - BroadSword, GiantAxe, StormRuler and YhormsGiantMachete which all extend the MeleeWeapon class. The purpose of extending the MeleeWeapon class is so that we can add more weapons in the future if needed **(Open/Closed Principle)**. **This helps to improve the code's maintainability and flexibility.**

The YhormsGiantMachete has one method called emberForm() which is responsible for increasing the hit rate of the machete by 30. This method will be called when the Yhorm the Giant activates its Ember Form. Yhorm's Giant Machete has been represented by the character 'L'.

The BroadSword class will override the damage() method from the Weapon class. The purpose of this method is to provide the BroadSword with a 20% success rate to deal double damage with a normal attack. **This is done by generating a random number in the range of 1-100 using the Random() in-built method**. If the random number generated is less than or equal to 20, the damage caused will be double, else it stays the same. The BroadSword has been represented by the character '1'.

The GiantAxe has an active skill called spin attack action. For this, we have created a **new class called SpinAttackAction which extends the WeaponAction class (Inheritance)**. This class has an **execute() method which checks if the actor is Player or enemy and also checks if the target is a Player**. If the target is the Player, the enemy will attack and the Player's **hit points will be reduced by calling the hurt() method**. If the Player has died, the **enemy will drop its weapon by calling the getDropWeaponAction()** and the Player's attributes will be reset by calling the ResetManager instance. If the target that has been killed is an enemy, its **location will be reset by calling the die() method.**

The GiantAxe class will override the getActiveSkills() method in which we will return the SpinAttackAction. This class also has another method called getAllowableActions in which we add an instance of the PurchaseBroadSword action to the allowable actions ArrayList so that the Player can purchase a Giant Axe through the Vendor. The Giant Axe has been represented by the character 'T'.

The StormRuler has 2 active skills called charge action and wind slash action. For this, we have **created 2 new classes called ChargeAction and WindSlashAction which both extend the WeaponAction class (Inheritance)**. The Storm Ruler has been represented by the character '7'.

The ChargeAction class has an **execute() method which increases the number of charges of the weapon by 1** and it returns a message that says that the storm ruler has been charged. This class also has an accessor to retrieve the number of charges of the weapon as well as a **resetNumOfCharge() method to reset the number of charges to 0 (Single Responsibility Principle)**.

The **WindSlashAction class also has an execute() method. This method will first check if Yhorm the Giant is an adjacent square by calling the in-built Character.compare() method as this weapon skill can only be executed when the holder stands next to Yhorm the Giant**. If Yhorm the Giant is next to the Player the damage caused by the weapon will be doubled and the enemy will be attacked. We will then check if Yhorm the Giant has died, and if so the Player will drop the weapon by calling the getDropAction() method and the enemy will be removed from the map by calling the map.removeActor() method. Lastly, the number of charges of the Storm Ruler will be restored to 0 by calling the resetNumOfCharge() method. If the Player is not standing next to Yhorm the giant, the method will return a message which says that the Player has missed the target **(Single Responsibility Principle)**.

The purpose of extending the WindSlashAction, ChargeAction and SpinAttackAction classes from the WeaponAction class is so that we can easily add more weapon skills in the future if needed **(Open/Closed Principle)**. **It increases the code's maintainability and flexibility.**

Requirement 8: Vendor

Design Choices:

For this feature, we will **create a new Vendor class that extends the Ground class**. We need to extend the Ground class so that we can access its methods and private attributes.

The Vendor has been added to the map by modifying the map ArrayList in the Application class and replacing the Ground on the left of the Bonfire with the Vendor. The Vendor has been represented by the character ‘V’.

The Player can purchase BroadSword, Giant Axe or increase maximum HP through the Vendor. For this, we have created 4 new classes called PurchaseWeaponAction, PurchaseBroadSwordAction, PurchaseGiantAxeAction and PurchaseStatAction. The PurchaseWeaponAction class extends the SwapWeaponAction class **(Inheritance).** The purpose of extending this class is so that we override the execute() method of the SwapWeaponAction class. The PurchaseBroadSwordAction, PurchaseGiantAxeAction and PurchaseStatAction all extend the PurchaseWeaponAction class **(Inheritance)**.

The PurchaseWeaponAction class has the execute() method which has been overridden from the SwapWeaponAction class. The purpose of this method is to loop through the Player’s current inventory and remove any pre-existing weapons.This is done by calling the removeItemFromInventory() method. Once the weapon has been removed the method then adds the new weapon that has been purchased to the Player’s inventory by calling the addItemToInventory() method. Finally, the method displays a message in the console that the purchase has been successful **(Open/Closed Principle)**.

The PurchaseBroadSwordAction class has an execute() method which retrieves the number of Souls that the Player has by calling the getSouls() method and checks if the number of souls is greater than the cost of the BroadSword. If the Player has enough souls, the purchase will be completed by calling the execute() method from the PurchaseWeaponAction class. If the Player doesn’t have enough souls, a message will be displayed to the Player that the purchase was unsuccessful due to lack of souls **(Single Responsibility Principle)**. The hotKey for this action is ‘c’.

The PurchaseGiantAxeAction class has an execute() method which retrieves the number of Souls that the Player has by calling the getSouls() method and checks if the number of souls is greater than the cost of theGiant Axe. If the Player has enough souls, the purchase will be completed by calling the execute() method from the PurchaseWeaponAction class. If the Player doesn’t have enough souls, a message will be displayed to the Player that the purchase was unsuccessful due to lack of souls **(Single Responsibility Principle)**. The hotKey for this action is ‘d’.

The PurchaseStatAction class has an execute() method which retrieves the number of Souls that the Player has by calling the getSouls() method and checks if the number of souls is greater than the cost of increasing the maximum hit points. If the Player has enough souls, the purchase will be completed by calling the execute() method from the PurchaseWeaponAction class. If the Player doesn’t have enough souls, a message will be displayed to the Player that the purchase was unsuccessful due to lack of souls **(Single Responsibility Principle)**. The hotKey for this action is ‘f’.

The purpose of extending the PurchaseWeapon class is so that we can purchase more weapons or items from the Vendor in the future if needed. **It helps to improve the code’s maintainability and flexibility.**