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: New Map & Fog Door

Design Choices:

For this requirement, we have created 4 new classes namely - Worldmap, ProfaneCapital, AnorLondo and MapsManager. **The worldmap class is an abstract class which the ProfaneCapital and AnorLondo classes will extend from.** The worldmap class has methods such as spawnEnemies, getName, getGameMap, spawnFogDoor, spawnBonfire and getFogDoorLocation methods. These methods will be overridden in the ProfaneCapital and AnorLondo classes.

The MapsManager class contains helper methods which can be used to modify the Profane Capital and Anor Lando maps**(Dependency Injection)**. This class **creates a new hashmap object called mapList <String, Worldmap> pair**. The first method in this class is the a**ddMap() method, which retrieves a specific map along with its name and adds it to the mapList hashmap**. The next method is the getMap method which is an accessor method to retrieve the name of the map. Lately, we have the spawnFogDoor method which loops through the mapList hashmap and calls the spawnFogDoor() method from the Worldmap class to add a fog door to the map.

The ProfaneCapital class creates an instance of the FancyGroundFactory class with Dirt, Wall, Floor, Valley, Vendor and Cemetery as Parameters. It also has an **arraylist of strings called map which has the map for the Profane Capital**. The first method we have in this class is the addPlayer() class. This method first creates a Player by creating an instance of the Player class. Then we add the Player to the map by **calling the addActor() method** from the World class. The next method is the **spawnEnemies() method which adds the Lord of Cinder and Storm Ruler to the map by calling the addPlayer() and addItem() methods**. Then, we add Skeletons to the map at random locations by using the **rand() in-built method and the getXRange() and getYRange() methods from the GameMap class**. The next method that we have is the spawnFogDoor() method which adds the fog door to the map by **calling the setGround() method from the Location class** and **creating an instance of the FogDoor class**. The last method in this class is the spawnBonfire() method which adds the Bonfire to the map by creating a new instance of the Bonfire class. Then, we **light the Bonfire by calling the litBonfire() method from the Bonfire class**. Then we a**dd the Bonfire to the map by calling the setBonFireManager() method** from the setBonFireManager class.

The AnorLondo class creates an instance of the FancyGroundFactory class with Dirt, Wall, Floor, Valley, Vendor and Cemetery as Parameters. It also has an **arraylist of strings called map which has the map for the Anor Londo.** The first method is the **spawnEnemies() method which adds the Aldrich the Devourer to the map by calling the addActor() method.**The next method that we have is the **spawnFogDoor() method which adds the fog door to the map** by **calling the setGround() method from the Location class** and **creating an instance of the FogDoor class**. The last method in this class is the **spawnBonfire() method which adds the Bonfire to the map by creating a new instance of the Bonfire class**. Then, we **light the Bonfire by calling the litBonfire() method from the Bonfire class**. Then we a**dd the Bonfire to the map by calling the setBonFireManager() method** from the setBonFireManager class.

Requirement 2: Updated Bonfire

Design Choices:

We have created a new class called **LitBonfire(Single Responsibility principle) which extends the Action class(Inheritance)**. The purpose of this class is so that the Player can light the Bonfire. This class has an execute() method in which we first call the **litBonfire() method** which we have written in the Bonfire class(explained below) and then we display a message in the console to the Player that the specific Bonfire is lit by calling the **getName() method** which we have also written in the Bonfire class. The second method we have in this class is the **menuDescription() method** which writtens a descriptive string that says that the bonfire has been lit.

We have also created another **new class called BonfireManager.** This class creates a new hashmap called bonfireList. This class has 2 methods - registerBonfire() method and getTeleportable() method. The **registerBonfire() method creates a new Bonfire and adds it to the bonfireList hashmap along with its location**. The **getTeleportable() method returns the hashmap of bonfires in the game along with their locations**. This class also has 2 methods called setLastInteractedBonfire() and getLastInteractedBonfire(). These methods are responsible for accessing and retrieving the last bonfire that the Player has interacted with.

We have also created another class called TeleportToBonfireAction which extends the MoveActorAction**(Inheritance)**. In this class, we create an instance of the Bonfire and BonfireManager classes. We override the execute() method from the moveActorAction class and then we **call the execute() method from the moveActorAction class**. Then we call the **setLastInteractedBonfire() method to set the last bonfire that the Player has interacted with**. Then we call the menuDescription() method to return a message in the console. The last method in this class is the **menuDescription() method which returns a descriptive string to the Player**.

Lastly, we have **made some changes to the pre-existing Bonfire class**. We have created **3 new instance variables- An instance of the BonfireManager class called bonFireManager, a boolean variable called lit to check if the bonfire has been lit and a string containing the name of the Bonfire**. We have also created **4 new methods(mentioned above) - getAllowableAction(), setBonfireManager(), litBonfire() and getName()**. The getAllowableActions() method creates a new ArrayList of Actions by creating an instance of the Actions class. We will then check if the Bonfire has already been lit by **calling the lit() method**. If the bonfire has already been lit, we will add the resetAction to the arrayList of allowable actions. Then we create a new hashmap that contains the bonfires by **calling the getTeleportable() method**. Then, we loop through the hashmap and add the moveActorAction to the allowable actions arraylist so that the actor can teleport from one bonfire to another. If the bonfire hasn't been lit yet, we add the litBonfire action to the list of allowable actions. The **setBonfireManager() method** first retrieves the bonFireManager instance variable created above and then stores the location of the Bonfire by **calling the registerBonfire() method** written in the BonfireManager class. The **litBonfire() method** sets the value of the lit instance variable to true so that it can be called when the Player lights the Bonfire. Lastly, the **getName() method is an accessor method** which returns the name of the Bonfire.

BonfireManager will be added to Player class through constructor. If the player dies, the player will spawn at the location of the bonfire that he last interacted with (either interacted through teleportation or lighting of bonfire). The last interacted bonfire and its location can be obtained from the BonfireManager class.

Requirement 3: Aldrich the Devourer (Lord of Cinder)

Design Choices:

For this requirement, we have **created a new AldrichTheDevourer class which extends the Enemies class which was created in assignment 2**. This class first overrides the playTurn() method. In this method, we **save the initial location of Aldrich the Devourer by calling the setInitialLocation() method**. Then we loop through the Inventory of Aldrich the Devourer and **check if any of the items have the LONG_RANGED_WEAPON capability by calling the hasCapability() method from the Item class**. If so , we **add the capability to the item by calling the addCapability() method from the Actor class**. Then we loop through all the possible exits that Aldrich the Devourer can move to and check if the Player exists at any of the exits by calling the containsAnActor() method. If this method returns true, we **retrieve the Actor at this location by calling the getActor() method** from the Location class. Then we check if Aldrich the Devourer has the capability to be hostile to the enemy. If so, we **add the FollowBehaviour to the enemy by calling the in-built add() method** and **remove the WanderBehaviour from the enemy by using the in-built removeIf() method**. Then we **return a new RangeAttackAction(explained below) so that Aldrich follows the Player over a range of 3 x 3**. Then we loop through the list of behaviours and get the allowable actions for each behaviour and return the action. We also **override the resetInstance() method from the Enemies class** in which **we remove the FollowBehaviour by calling the in-built removeIf() method**. We will then **move Aldrich the Devourer back to its initial location by calling the MoveActor() method**.

We have also created another **new class called RangeAttackAction(used in the AdlrichTheDevourer class) which extends the Action class**. The first method in this class is the execute() method which loops through the possible locations that the actor can move to by comparing the location of the actor and the target. **The method also checks if an attack will be blocked by calling the getGround() and blocksThrownObjects() methods to check if the Ground blocks any objects that are thrown at it**. If so, the attack will be automatically blocked. If the attack is successful, we **call the getWeapon() and damage() method to retrieve the amount of damage that the weapon inflicts on the target**. Then we **call the hurt() method to reduce the hit points of the target** depending on the damage that the weapon inflicts. This class also has a menuDescription() method which returns null.

Aldrich the Devourer initially holds the Darkmoon Longbow weapon so we have **created a new class called DarkmoonLongBow which extends the weaponItem class. The DarkmoonLongBow class will override the damage() method from the Weapon class**. The purpose of this method is to provide the BroadSword with a 15% 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 15, the damage caused will be double, else it stays the same. The **next method we have is the toString() method, this method returns a string containing the name of the weapon.** The last method in this class is the **GetDropAction() method which has been overridden from the Item class**. This **method returns null so that the Darkmoon Longbow can't be dropped**.

Requirement 4: C4 Bomb

Design Choices:

Souls can be traded by the Player to buy a new item called C4 Bomb through the Vendor. The price of the C4 Bomb will be 200 souls. The player can place the bomb anywhere on the map(as long as the Player can enter that area), but the bomb will only activate if it is dropped on Dirt ground, otherwise it will just deactivate (do nothing).

Once the Player drops the bomb, it will take 2 turns before the bomb explodes and causes damage to actors that are in the surrounding area, including the player himself. The damage caused by the bomb will be 50 hit points. If an actor is still conscious after receiving damage, he will be stunned for a turn.

The bomb can only be activated once, then it will be removed from the map after it has been used.

Requirements	**Features (HOW) / Your Approach / Your Answer**
Must use at least two (2) classes from the engine package.	Classes like Location and Exit from the engine package are being used for this requirement. (there are more classes from engine package being used)
Must use/re-use at least one (1) existing feature (either from A2 and/or A3 - fixed requirements)	The C4 bomb can be **purchased from the Vendor** which we have implemented in assignment 2.
Must use existing or create new abstraction/interfaces (apart from the engine code).	Purchase of this C4Bomb will be done using PurchaseC4BombAction which extends from the **PurchaseAction class which is an abstract class.** We have created a new class called C4Bomb which extends the **Item class which is an abstract class** from the engine package. When the bomb is placed on the Ground, it explodes after 2 turns. A new class called Bombed Ground which extends the **Ground class** from the engine package is created that changes the ground to a bombed ground and it will reduce the hit points of the actor who is standing on it.

Must use existing or create new capabilities.	We have made use of the **STUNNED capability**. When an actor receives damage by stepping on the bombed ground and he is still conscious, this actor will be added a capability of STUNNED status.
Must explain why it adheres to the SOLID principles.	We have created a new class for the C4 bomb. This abides by the **Single Responsibility principle** as the class has only one responsibility. This one class contains all the functionality needed to support that responsibility. We have also created a PurchaseC4BombAction class which extends the PurchaseAction class so that the Player can purchase the bomb through the Vendor. This abides by **Liskov Substitution Principle** as the PurchaseC4BombAction, C4Bomb and BombedGround classes inherit their functions and abilities respectively from their parent class

We have created a **new C4Bomb class(Single Responsibility Principle) which extends the Item class(Inheritance)**. The purpose of extending the Item class is so that we can use it as an item. In this class, we have **overridden the getPickUpAction() method to return null** so that the Player can't pick up the bomb. We have also **overridden the tick() method from the Item class**. In this method, we **first check if the location where the bomb is dropped on dirt by using the in-built instanceOf method**. If the location is not dirt, **the bomb will be removed from the Ground by calling the removeItem() method** and a message will be displayed to the user that the Bomb has been deactivated. Then we check if the number of turns after dropping the bomb has been more than 2, and if so, we first find which map the Player is in by calling the currentLocation.map() method from the Location class. Then, we set the current location of the Player to Bombed Ground by calling the **setGround() method from the Location class** and creating a **new instance of the BombedGround class**(explained below). We will then loop through the list of possible exits from the current location by calling the **getExits() method** and then get the final destination by calling the **getDestination() method**. If the surrounding ground is an instance of dirt, we set the ground to bombedGround by **creating a new instance of the bombedGround and adding it to the map by calling setGround() method**

We have also created a **new BombedGround class which extends the Ground class (Inheritance)**. The purpose of this class is to ensure that the bombed ground lasts for only 3 turns. **This class has a method called tick() which has been overridden from the Ground class.** Each time this tick() method is executed, we increase the tickCount instance variable(initially 0) by 1. **We check if the tickCount instance variable is more than 3 by calling the getTickCount() method written in the same class and**

if so, the BombedGround around the Player will be reset to Dirt by calling the setGround() method with an instance of the Dirt class as parameter. This is done by calling the setGround() method with a new instance of Dirt as the parameter. Then, the method checks if there is any actor stepping on the burning ground. All actors including the PLayer will lose 50 hit points if they step on the Bombed Ground. This is done by calling the location.getActor() method to get the Player at that specific location and then **calling the hurt() method to reduce the hit points of the actor**. A message will be displayed that the actor has lost 50 hit points. Then, we call the **cleanBattle() method from the CleanBattleField class(explained below) to remove the actor from the map if it has died** and display a message in the console that the actor has died.If the actor is an enemy, we will **add the STUNNED capability by calling the addCapability() method** from the Actor class. The second method in this class is the getTickCount() method which is an accessor to retrieve the value of the tickCount instance variable which we have used in the tick() method.

We have created a PurchaseC4BombAction class which extends the PurchaseActionClass (Inheritance). The purpose of extending the purchaseAction method is so that we can access the methods and private attributes of the class. The PurchaseC4BombAction class has only 2 methods- **menuDescription() and doAction()** which have both been overridden from the PurchaseAction class. The menuDescription() method returns a descriptive string which says that the Player has purchased the C4 Bomb for 200 Souls. **The doAction() method adds the item to the inventory by calling the addItemToInventory() method from the Actor class**.

Lastly, we have created a new class called CleanBattleField. This class has only one method called cleanBattle(). The cleanBattle() method checks if the actor is alive by calling the isConscious() method and if the actor is dead , it loops through the inventory and drops all the items that the actor has. Then we **check if the actor is the Player by calling the in-built instanceOf() method,** and if so, we reset the attributes of the Player by creating an instance of the ResetAction class and then calling the execute() method from the Action class. If the actor is an enemy, we call the **die() method** which we have previously written in the enemies class to remove the actor from the map and transfer the reward amount(Souls) to the player.

Requirement 5: Invisibility

Design Choices:

Souls can be traded by the Player to buy a new item called invisibility cloak which makes the Player invisible to enemies. The price of the invisibility cloak will be 500 Souls. The Player can equip the cloak anytime. If the invisible cloak is used, the player will be invisible in the map, enemies will stop attacking the player if they are already attacking, and continue their wandering behaviour. If no enemy is attacking player, enemies will ignore the presence of the player even if player is nearby.

Requirements	**Features (HOW) / Your Approach / Your Answer**
Must use at least two (2) classes from the engine package.	We have created 3 new classes- InvisibleCloak, InvisibleAction and PurchaseInvisibleCloak. The Invisible cloak extends the **Item class** and the InvisibleAction extends the **Action class**
Must use/re-use at least one (1) existing feature (either from A2 and/or A3 - fixed requirements)	The invisible cloak can be purchased from the Vendor which we have implemented in assignment 2
Must use existing or create new abstraction/interfaces (apart from the engine code).	Purchase of this C4Bomb will be done using PurchaseC4BombAction which extends from the **PurchaseAction class which is an abstract class.**
Must use existing or create new capabilities.	We have created a **new capability called INVISIBLE**. When the player executes InvisibleAction, this INVISIBLE status will be added to the player as a new capability.

Must explain why it adheres to the SOLID principles.	**We have created a new class for the invisible cloak, this abides by the single responsibility principle as t**he class has only one responsibility. This one class contains all the functionality needed to support that responsibility. We have also created a PurchaseInvisibleCloak class which extends the PurchaseAction class so that the Player can purchase the Invisible Cloak through the Vendor. This abides by **Liskov Substitution Principle** as the PurchaseInvisibleCloak class inherits its functions and abilities from its parent class

We have created a **new InvisibleCloak class which extends the Item class(Single Responsibility Principle)**. The first method in this class is the **getAllowableActions() method**. This method loops through the arraylist of allowable actions and adds the InvisibleAction(explained below) as an allowable action if it is not present by calling the **add() method from the Actions class**. We have also overridden the tick method. In this method, we first check if the cloak is currently being used by the player by calling the **isUsing() method** written in the same class. If true, we increase the tickCount instance variable(which represents the number of times the tick method has been executed) by 1. Then we check if the tickCount variable is equal to 3 by calling the **getTickCount() method**. If so, we remove the INVISIBLE capability from the Player by calling the **removeCapability() method** from the Player class. Then we reset the tick count to 0 by calling the **resetTickCount() method** and lastly reset the isUsing instance variable to false. The next method is the getTickCount() method which returns the value of the tickCount instance variable. We also have the resetTickCount() method which resets the value of tickCount to 0. Lastly, we have the isUsing() method which checks if the Player has equipped the invisible cloak.

We have **created a new class called InvisibleAction which extends the Action class(Inheritance)**. In this class, we have **overridden the execute() method**. In this method, we **add the INVISIBLE capability to the Player by calling the addCapability() method** from the Actor class. Then we call the actorUsing() method so that the actor equips the cloak. Lastly we call the menuDescription() method to return a descriptive message of the action that has happened.

The last new class which we have created is the **PurchaseInvisibleCloak class which extends the PurchaseAction class (Liskov Substitution Principle)**. The purpose of this class is so that the Player can purchase the Invisible Cloak from the Vendor. The first method in this class is the **doAction() method** which has been overridden from the PurchaseAction class. This method adds the item to the inventory by calling the **addItemToInventory() method** from the Actor class. Lastly, we have the menuDescription() method which returns a descriptive string after the purchase action has been completed.