Design Rationale

Tong Jet Kit, Aaren Wong Cong Ming, Yew Yee Perng

Requirement 1

This UML class diagram is to showcase 2 new grounds and 2 new interactions with the Site Of Lost Grace that a Player can do other than resting. The new grounds are a new landscape which is a cliff that will kill a player if standing on it and a new door which acts as a doorway for players to travel to other parts of Limgrave which is the world of Elden Ring. For the 2 new interactions, the Player now needs to activate The Site Of Lost Grace before they are able to rest on it and the Player can fast travel now using The Site Of Lost Grace to other activated Site Of Lost Graces. The goal of this design is to increase the difficulty of the game by adding new things that can kill the player, new places to rest and new locations to discover and adventure on while providing modularity and extensibility.

Firstly, the cliff is a new ground so it would extend the Ground abstract class since it has all the common attributes of a ground. The cliff will kill any Player standing on it so it has a dependency on the Location and CapabilitySet class as it has to check if there is an actor standing on this location where the cliff is at and if the actor is a Player or not. Moreover, the CapabilitySet could also help to determine whether an actor can step on the cliff. The usage of inheritance will help reduce code duplication as the cliff shares similar attributes with the Ground class as it itself is a ground.

This design adheres to the Liskov Substitution Principle as the cliff should function as the same as a ground. For example, when the gameMap ticks a ground object it can expect that the ground object would be the Cliff or other new grounds we created. Moreover, the Do Not Repeat Yourself Principle is also followed as code duplication is reduced through the usage of inheritance. Therefore, this design will make it easier to add new grounds which will kill a player in the future through inheriting the Ground abstract class and using the location to kill the actor.

Next, there is a new interactable ground which is the Golden Fog Door which allows the player to travel to other parts of Limgrave. Since the Golden Fog Door is a type of ground, this class will inherit the Ground abstract class to inherit all the attributes of a ground object. Additionally, other than allowing the player to rest on The Site of Lost Grace, it also can allow players to travel too. However, the site needs be activated before the Player is able to rest on it or fast travel.Thus, an Activate Action is created that inherits the Action abstract class as it is an action so it shares similar attributes with other action classes. Since its an activatable ground, the Site of Lost Grace class will implement the activatable interface to allow this ground to be activated. The Activate Action will have an association with the Activatable class as it will activate the activatable object after executing the action.

Since the Golden Fog Door and an activated Site of Lost Grace can allow the player to travel, they will implement an interface called Travelable which contains methods to allow the Golden Fog Door or Site of Lost Grace to move the player from one place to another.

Additionally, since a player can travel through the travelable grounds, the classes will have a dependency on the map, location and the actor. This is because the travelable class will need to transport the player from this map to another location in another map. Therefore, they will depend on these 3 classes to be able to move the player which is an actor.

To allow the player an option to travel, an action called Travel Action is created. Since the Travel Action is an action, this class will inherit the Action class since this action is also a type of action and the action abstract class has all the common attributes of an action. Thus we can adhere to the Do Not Repeat Yourself Principle. The Travel Action has an association with the travelable class since the player will use the travelable object to travel after executing the travel action. Subsequently the travel action will have a dependency to the actor, location and map since when executing the action it will need to move the player from a location on the current map to another location on another map. Finally, to allow the player to know which map they are in, the FancyMessage class is used to output an ascii art that displays the map name. So the TravelAction will have a dependency to the FancyMessage class. Lastly, since there are different maps for the player to travel to, the GameMap class is needed to create the map.

This design adheres to the Liskov Substitution Principle, Do Not Repeat Yourself and Interface Segregation Principle. For example, the Golden Fog Door is a ground while the Travel Action is an action so it should work like its parent class and exhibit all behaviour of the Ground or Action class. The Interface Segregation Principle can be seen too through the usage of Travelable and Activatable interface since the ground that will be activatable by players or allow the player to travel will just need to implement the interface to have the methods to travel while other grounds which do not have that functionalities will not need to implement the methods. Finally, the Do Not Repeat Yourself Principle can be preserved through inheritance and implementing interfaces. Through this design, we can extend the system to have new activatable grounds or grounds that the player can interact with to travel. Since the new class just has to extend the Ground class and implement the Travelable or Activatable interface according to their functionality. If the new map has a name we can just add the ascii art of the map name to the FancyMessage class to print out the map name.

Requirement 2

This UML class diagram will showcase 2 new grounds and 2 new enemies added into the game. These new grounds are Cage and Barracks used to spawn the new enemies whereas the new enemies are Godrick Soldier and Dog who inhabit Stormveil Castle.

Firstly, the new enemies added are Godrick Soldier and Dog, which plays a role of foes in the StormVeil castle map in the game. They extend the Enemy abstract class as they are characters in the game that have similar properties such as they have a collection of behaviours, thus supporting the DRY principle to avoid repetitions for classes that share similar attributes and methods. Similarly, the Enemy abstract class extends the Actor class since they are actors in the game thus inheriting all its properties, supporting the DRY principle for classes that share similar attributes and methods. Besides that, the Enemy abstract class also has an association with two Enum types, which are Species and Status, which reflects the species and the status of the enemy itself. An advantage of creating an Enemy abstract class is it eases the addition of new enemies if needed in future implementation as they would just have to extend the Enemy abstract class to obtain all the common attributes that an enemy would share. However, if the child classes have a lot of unrelated information obtained from the parent abstract class, it is better to use an interface instead of making an abstract parent class.

Secondly, the added new grounds that spawns the enemies are Cage and Barracks which extends the Spawning Ground abstract class, where SpawningGrounds is a subclass of abstract class Ground as they are all a type of ground. SpawningGrounds uses location to get the coordinates in the game thus having a dependency to location. The reason for creating an abstract class for these spawns is because each ground that spawns enemies has a factory to spawn enemies depending on their location to spawn the relative enemy. So to reduce code duplication, the SpawningGrounds abstract class is created and all the grounds that can spawn an enemy will just extend it. Besides that, they require EnemyFactory, an interface that acts as a blueprint for EastEnemyFactory and WestEnemyFactory. These Factory classes take the role of Spawning Enemies depending on their location, thus having a dependency with location. WestEnemyFactory takes care of spawning enemies that have a location on the map which reflects to the west side, where EastEnemyFactory takes care of spawning enemies on the east side depending on their location. By having EnemyFactory, we just simply have to add some code into it when having any new enemies and can leave the classes unaltered during the process. Furthermore, the reason to make EnemyFactory to be an interface instead of abstract class is because there we only need the method provided in the interface and the implementation of the methods is unique to the east and west side. By doing so, this adheres to the Single Responsibility Principle as every class only takes care of one responsibility and Dependency Inversion principle as the class depends on abstract classes and interfaces instead of each other.t

Thirdly, Actions is an abstract class that enemies have a dependency on. It takes care of the actor's actions in the game such as DeathAction and AttackAction. DeathAction is used when an enemy or player dies, thus they have a dependency with DeathAction. Besides that, AttackAction is also used when an enemy or player attacks, therefore the enemies and player have a dependency with AttackAction. Similarly for DespawnAction , the player and enemies have a dependency with them as they rely on them to despawn. Despite being all Actions, these actions have only one role, which follows the Single Responsibility Principle. Moreover, it minimises the possibility of having a god class, which might prevent the whole code running if error is found.

Godrick soldiers carry a weapon called a club. It extends the WeaponItem class, and is Sellable and Purchasable by implementing the interface Sellable and Purchasable. Therefore it preserves the Interface Segregation Principle as each weapon implements the interfaces that it needs. On the other hand, Enemies such as Dog rely on Intrinsic Weapons to perform its attacks, thus having a dependency on it. Furthermore, actors regardless of roles would have either have a capability of being hostile to enemy or hostile to players, thus supporting the association. Hence, in future if there are enemies that have a weapon we can just extend the WeaponItem class for that weapon and associate the weapon with its corresponding enemy. If the weapon is sellable or purchasable they can just implement the respective interface. So the DRY principle and Interface Segregation Principle can be adhered to.

Besides that, the enemies have an association with behaviour Interface as they have a collection of behaviours that appears in the game. It acts like a console menu for the enemies. For example, when an actor gets within the range of an enemy, the enemy would react by showing a behaviour resulting in an attack to the actor. This allows them to access the dependencies of Behaviours such as Wander Behaviour and Follow Behaviour. With that, they are able to wander around the map or follow the player when in range. Specifically, the AttackBehaviour has a dependency with AttackAction as the behaviour depends on whether the enemy is going to perform an attack on their rivals. This behaviour implements Behaviour interface as they use the interface as a blueprint to implement the methods needed. The reason for not making Behaviour interface as an abstract class is because all behaviours might have different attributes. For example, AttackBehaviour has a target, but WanderBehaviour does not. By doing so, it adheres to the Do Not Repeat Yourself (DRY) principle as it eliminates redundancies of code by reusing applicable codes.

Requirement 3

This UML class diagram will showcase 2 new weapons, 2 new items, and a new trader which were added into the game. The goal of the design was to make a new trader which could exchange weapons and items as well as sell them along with having golden runes be randomly spawned around the map while proving modularity, extensibility and maintainability.

Firstly, the 2 new weapons and 1 new item which can be exchanged and sold. The 2 new weapons are called the AxeOfGodric and GraftedDragon which will extend the WeaponItem abstract class, following the Do Not Repeat Yourself (DRY) Principle. The mentioned item would be the RemembranceOfTheGrafted which extends the Item abstract class. As the RemembranceOfTheGrafted is used to exchange for the one of the 2 new weapons, it will also have an association to the CapabilitySet to check if it can be exchanged and implements the Exchangeable interface. As all 3 of these new additions are also sellable, they will have a dependency to the CapabilitySet to check if they are capable of being sold and also implement the Sellable interface as well. Having these capabilities and interfaces allows for extensibility in the case of future additions of items or weapons such as a Wand that can be used to exchange for other stuff while also being able to be sold to a trader. With these items and weapons having some implementations, it adheres to the Interface Segregation Principle as each item or weapon will implement the necessary interface depending on whether they are sellable or exchangeable or both .

Next, the new trader which is FingerReaderEnia that exchanges and sells both weapons and items, was implemented by modifying the Trader class that extends the Actor abstract class to inherit all the attributes of an actor. As the trader is capable of exchanging and selling, it will have a dependency to the SellWeaponAction, SellItemAction, and ExchangeAction class that all extend the Action abstract class since these actions are also a type of action and the action abstract class has all the common attributes of an action. Thus, we can follow the Do Not Repeat Yourself (DRY) Principle and also Single Responsibility Principle as a result . Another option is to combine the two trading actions into a single action called tradeAction, but in order to uphold the Single Responsibility Principle, it is preferable to segregate the two trading activities into different classes. The Trader class will also have an association with the CapabilitySet to check if it is capable of buying items sold by the player or to exchange weapons and items along with 2 associations to the Sellable and Exchangeable interface which follows the Dependency Inversion Principle as it has a list of exchangeable and sellable items and weapons. The reason it is a list of exchangeables and sellables instead of a list of items and a list of weapons is to avoid overgeneralization as certain things that can be sold cannot be exchanged and vice versa, hence having a list of specifics allows the problem to be avoided. Last but not least, the trader in this system is unable to move, attack, or die. The merchant class is therefore dependent on DoNothingAction but not on AttackAction or DeathAction. As an alternative, we can distinguish between those who can and cannot die by creating an interface called Mortality.

The second new item that was created was the GoldenRunes which is a consumable item that will be randomly dropped around the map. Being a consumable item, the GoldenRunes will implement the Consumables interface, providing abstraction, allowing the item to have the functionalities it requires as a consumable item, and also adhering to the

Interface Segregation Principle as the GoldenRunes will only use methods that a consumable would use that serve no purpose to non-consumable items. Having a Consumables interface also allows extensibility as any future addition of consumable items that might be added such as a mana potion can just implement the interface. Due to it being an item, it will extend the Item abstract class to inherit all the attributes of an item. As it is also a consumable which will be providing a ConsumeAction to the player, it will also have a dependency to the ConsumeAction class which extends the Action abstract class, to have all the attributes an action should have. As the GoldenRunes will provide the player with a random number of runes when consumed, it will also have a dependency to the RandomNumberGenerator class. Due to the player’s rune count being updated from consuming it, there is also a dependency to the RuneManager class which handles the updating of the player’s rune count. As the GoldenRunes will be randomly scattered around the map during the start of the game at locations that can be dropped such as Dirt and Floor that are empty, the Dirt and Floor class will have a dependency to the GoldenRunes while also having an association to the CapabilitySet to ensure that GoldenRunes can be dropped on them. The reason we only allow the GoldenRunes to be dropped on the Dirt and Floor is because it is the general ground of a map that is always accessible to the player compared to being able to be dropped at other grounds such as the SpawningGrounds which already has its own unique feature which is having enemies be spawned which could cause trouble for the player to navigate to the rune without the risk of dying. As such the player will always be able to navigate to it, allowing confirmed access to any GoldenRunes that can be found. This also allows the GoldenRunes feature to be displayed much better to the player without much trouble. If added difficulty is desired in retrieving GoldenRunes, future changes can be made to the game such as allowing the runes to be dropped at the SpawningGrounds while the player has an assistant NPC that can be tasked to focus on certain tasks such as retrieving the runes while the player fights the enemies.

This UML class diagram uses inheritance and polymorphism, as mentioned above, to offer modularity, extensibility, and reuse. The drawback of this design is that it has an excessive number of weak relationships due to a high amount of dependencies. This issue can be resolved by connecting the entities and the action class via a manager class that will manage the actions. As a result, this architecture adheres to the Do Not Repeat Yourself, Interface Segregation Principle, Dependency Inversion Principle, and Single Responsibility Principles.

Requirement 4

This UML class diagram will showcase a new role we introduced called Astrologer, a new weapon brought along by an Astrologer named Astrologer staff, two mysterious characters named Ally and Invader who are summoned upon a new ground named Summon Sign.

First of all, An astrologer is a new role to be picked in addition to the 3 roles that exist in the game (Samurai, Wretch and Bandit) by the player, who is an actor in the game. The Player class will extend the Actor class to inherit all its properties and avoid repetitions. Astrologer extends the Player class since it is a player too. This design will follow the Single Responsibility Principle as each class (Player class) is only responsible for one type of class. For example, the Astrologer class handles everything that relates to Astrologer while other role classes take care of their own role rather than grouping them up together into a single class. The Player class is an abstract class since it is logical that we do not instantiate a player but either a Samurai, a Bandit, an Astrologer or a Wretch player. By doing so, the game may extend the type of classes that a player can be with ease due to the abstraction. This will adhere to the Liskov-Substitution Principle as each combat class is players. Alternatively, we could make the Player as a concrete class while having an association with the Class (referring to the combat classes) class. However, this design is pointless since we should be controlling a Samurai which is a Player not a Player which is a Samurai. Moreover, to ensure the consistency in the name of the class, the enum class ClassType is there to ensure it. Alternatively, we can just use a String to represent the class type but to prevent runtime error, an enum class is recommended.

Secondly, An Astrologer Staff is a unique signature weapon for Astrologers. It extends the WeaponItem abstract class since it is a weapon item and shares common attributes with other WeaponItems. It is also dependent on Astrologer since it starts with its own unique signature weapon. Additionally, each weapon may or may not have a skill which is a capability, so it has an association with the CapabilitySet class. If the weapon has a skill then it needs to be able to execute it, however Astrologer Staff does not have any. Besides that, all player weapons in the game implements Sellable and Purchasable since they are purchasable and sellable. This adheres to the Interface Segregation Principle as the weapons only implements the interfaces they need. For example, if a weapon is only sellable but not purchasable, it only needs to implement a sellable interface instead of implementing both sellable and purchasable. By extending WeaponItem abstract class, it adheres to the DRY principle, since it shares similar attributes and methods. An advantage of having an abstract class is it eases the addition of new weapons if needed in future implementation as they would just have to extend the WeaponItem abstract class to obtain all the common attributes that a weapon would share. However, if the child classes have a lot of unrelated information obtained from the parent abstract class, it is better to use an interface instead of making an abstract parent class.

Thirdly, Summon Sign is a new ground added to the game used to Spawn allies or Invaders. It implements Summonable, an interface for it which adheres to the Interface Segregation Principle as it implements the interfaces that it needs, since only SummonSign can summon an ally or invader. Besides that, it also extends Ground as it is a type of ground. In addition, it has an association with location as they need the coordinates of the map to obtain its location. Furthermore, Summon Sign also has a dependency with exits as it depends on exits to check if the player is in the vicinity of it, which then uses Summon Action to either summon an Ally or Invader. Similarly with the WeaponItem abstract class, it not only reduces code repetition by inheriting a parent class as grounds share similar attributes and methods which adheres to the DRY principles, it also eases the addition of new grounds or summonable grounds that might be added in future implementations.

Ally and Invader play the role of allies to player and enemy to player as characters in the game respectively.They are randomly spawned upon summoned and also have the hit-points and unique signature weapon of the random chosen character. The Ally is hostile to enemy and the Invader is hostile to player. They both extend the Creep abstract class as they are characters in the game that have similar properties such as they all have a collection of behaviours, thus supporting the DRY principle to avoid repetitions for classes that share similar attributes and methods. Similarly, the Creep abstract class extends the Actor class since they are actors in the game thus inheriting all its properties, supporting the DRY principle for classes that share similar attributes and methods. Besides that, the Creep abstract class also has an association with two Enum types, which are Species and Status, which reflects the species and the status of the enemy itself. They also have an association with CapabilitySet, since every creep has a status which could be hostile to enemy or hostile to player. Although an Invader is hostile to enemy, it does not prioritise its attacks on a Player since it can attack any existing opponent other than its own species. An advantage of creating an Creep abstract class is it eases the addition of new creeps if needed in future implementation as they would just have to extend the creep abstract class to obtain all the common attributes that an enemy would share. However, if the child classes have a lot of unrelated information obtained from the parent abstract class, it is better to use an interface instead of making an abstract parent class.

Actions is an abstract class that Creeps have a dependency on. It takes care of the actor's action in the game such as DeathAction and AttackAction. AttackAction is also used when an enemy or player attacks, therefore the enemies and player have a dependency with AttackAction. DeathAction is used when an enemy or player dies, thus they have a dependency with DeathAction. DeathAction uses ResetAction to perform a reset when the player dies. The ResetAction would then call ResetManager. The ResetManager also associates itself to ensure that there is only one instance of it because all the resettable entities will reset together as the reset is being called. Hence is it enough to have just one instance of the ResetManager. ResetManager implements Resettable, which helps in the reduction of the amount of dependencies between both ResetManager and resettable entities. By doing so the design also adheres to the Dependency Inversion Principle and Open-Closed Principle since we do not need to fix the ResetManager to accommodate new entities that are resettable while putting a layer of abstraction between the ResetManager and the ressettable entities.

Similarly, RestAction will have a dependency with ResetManager as enemies are despawned when the player rests. SummonAction uses Summonable interface to get the location of Summon Sign since Summonale stores the location of it. Therefore, it adheres to the dependency inversion principle as SummonAction relates to summonable instead of SummonSign. This allows any future actions to depend on summable instead of their grounds, which also supports the Open Closed principle. For DespawnAction , the actors have a dependency with them as they rely on them to despawn. Despite being all Actions, these actions have only one role, which follows the Single Responsibility Principle. Moreover, it minimises the possibility of having a god class, which might prevent the whole code running if error is found.

Besides that, the Creeps have an association with behaviour Interface as they have a collection of behaviours that appears in the game. It acts like a console menu for the Creeps. For example, when an actor gets within the range of an ally or invader, it would react by showing a behaviour resulting in an attack to the actor of the opposite status. This allows them to access the dependencies of Behaviours such as Wander Behaviour and Follow Behaviour for Invader and Wander Behaviour for Ally. With that, Invaders are able to wander around the map or follow the player when in range. Specifically, the AttackBehaviour has a dependency with AttackAction as the behaviour depends on whether the enemy is going to perform an attack on their rivals. This behaviour implements Behaviour interface as they use the interface as a blueprint to implement the methods needed. The reason for not making Behaviour interface as an abstract class is because all behaviours might have different attributes. For example, AttackBehaviour has a target, but WanderBehaviour does not. By doing so, it adheres to the Do Not Repeat Yourself (DRY) principle as it eliminates redundancies of code by reusing applicable codes.

Requirement 5

This UML class diagram will showcase a new type of mechanic in the game where the player is able to obtain a blessing or a curse through praying in the Church of Irith. However, to get to the Church of Irith, the Player has to defeat a boss to unlock the new Map. The goal of this design is to enhance the game experience by providing the player with an engaging and immersive gameplay element that is locked behind a boss fight that the player can overcome to be able to obtain advantages, or challenges based on the outcome of their prayers.

Firstly, the Church of Irith which is the building that the Player will interact with to obtain the blessing or curse is a new ground. Therefore it should extend the Ground abstract class as it shares similar attributes with all the other grounds. Thus, it will reduce code duplication. To allow the player to pray, a Pray Action has been created that inherits the Action abstract class as it is an action so it shares similar attributes with other action classes. Hence, it is also evident that the Church of Irith has a dependency on the Pray Action class. This Pray Action has a dependency on the RandomNumberGenerator, Actor, CapabilitySet and the RuneManager class. This is because the blessing and curse are random so it needs the RandomNumberGenerator to select the outcome of the praying action. Furthermore, the runes given or taken are within a random value so it needs the RandomNumberGenerator class to select. Next, since the blessing and curse will affect the player's hp and runes it will have a dependency on the Actor, CapabilitySet and RuneManager class as the RuneManager manages any operation that involves runes. Alternatively, we can create a blessing and curse action separately that does the blessing and the cursing respectively. However this design contradicts with the design goal since the blessing and cursing should not be of the player's choice but random.

This design adheres to the Liskov Substitution Principle, Single Responsibility Principle and the Do Not Repeat Yourself Principle. For example, the Church of Irith and Pray Action inherits its respective parent class therefore they should function as its parent class which allows us to reference them by using the parent class without altering the correctness of the program. Moreover, through inheritance, we can reduce a lot of code duplication which will be hard to maintain if there is a lot of it. The single Responsibility Principle can be seen from the Pray Action, rather than fine grating the blessing and cursing into 2 different action classes, this design merges them into one as a prayer should handle either getting a blessing response or a curse. This design also provides extensibility for future extension of more grounds to pray upon since they can allow the player to pray by providing the Pray Action and extend the Ground abstract class as there will be common attributes. Hence this design will be easier to maintain through the modularity and reusability of the design.

Next, the Player needs to defeat a boss to be able to access the Church of Irith. Therefore a boss needs to be created. This boss is the Regal Ancestor Spirit which will inherit the Enemy abstract class as itself is an enemy so they share similar attributes with other enemies like its behaviour thus supporting the DRY principle to avoid repetitions for classes that share similar attributes and methods. Similarly, the Enemy abstract class extends the Actor class since they are actors in the game thus inheriting all its properties, supporting the DRY principle for classes that share similar attributes and methods. Besides that, the Enemy abstract class also has an association with 2 Enum types, which is Species Status, which reflects the species and status of the enemy itself to allow differentiation of the normal enemies and the boss. An advantage of using an Enemy abstract class is it eases the addition of new enemies or bosses if needed in future implementation as they would just have to extend the Enemy abstract class to obtain all the common attributes that an enemy would share.. However, if the child classes have a lot of unrelated information obtained from the parent abstract class, it is better to use an interface instead of making an abstract parent class. This boss has a dependency on the Intrinsic Weapon as it uses its horns which is an intrinsic weapon to attack.

Other than attacking normally, this new boss has a skill which allows it to heal itself after attacking. Hence, a LifestealAction class is created. Since this is an action, it will inherit the Action abstract class as it is an action so it shares similar attributes with other action classes. This LifestealAction class has a dependency on the Attack action class as it uses this action class to attack the target. Both of these Action classes have an association with the actor, IntrinsicWeapon and the weapon interface class. This is because they need to know who to attack and with what weapon whether it's intrinsic or not. This action also has a dependency on the DeathAction class as well since this attack may kill the target. The AttackAction and the DeathAction class will also extend the Action abstract class as well as they are also an action. The DeathAction however just only has an association with the actor class as it only needs to know who died. Subsequently, since actions is an abstract class that takes care of the actor's action in the game such as DeathAction and AttackAction. The boss which is also an enemy has a dependency on.

This new boss also has an association with the Behaviour interface as they have a collection of behaviours that appears in the game. The interface acts like a console menu for the enemies. The new boss has the behaviour of wandering, following players, attacking and using a lifestealAttack. Therefore to allow the Regal Ancestor Spirit to do the lifesteal attack, we created a LifestealBehaviour class which will allow the boss to perform the Lifesteal Action. This class will implement the Behaviour interface as it is a behaviour so it will implement it to obtain all the methods that a behaviour has. The reason for not making Behaviour interface as an abstract class is because all behaviours might have different attributes. For example, AttackBehaviour has a target, but WanderBehaviour does not.

This design adheres to the Liskov Substitution Principle, Dependency Inversion Principle and the Do Not Repeat Yourself Principle. For example, the new boss and action both inherit its respective parent class which has all the common attributes and methods. Therefore, through inheritance, we are able to reduce the amount of repeated codes in the RegalAncestorSpirit and LifestealAction class. Since, the boss and action inherits the Action and Enemy class. It is also expected of them to function as their parent class. Therefore, its reference type should be able to be replaced by its parent class without altering the correctness of the code. Finally the Dependency Inversion Principle can be seen by the Enemy having a collection of the Behaviour Interface rather than having an association with each of the Behaviour independently. This can reduce coupling between the concrete Behaviour class so that if any changes are made to the Behaviour concrete class there won't be a need to modify the enemy class. Moreover, classes that are tightly coupled are not recommended as it will lead to violation of the Open/Closed Principle as if-else statements are needed and they are not recommended. Lastly, despite being all Actions, these actions have only one role, which follows the Single Responsibility Principle. Moreover, it minimises the possibility of having a god class, which might prevent the whole code running if error is found.

Finally, after defeating the boss a Golden Fog Door will appear that allows the player to travel, they will implement an interface called Travelable which contains methods to allow the Golden Fog Door to move the player from one place to another. Additionally, since a player can travel through the travelable grounds, the classes will have a dependency on the map, location and the actor. This is because the travelable class will need to transport the player from this map to another location in another map. Therefore, they will depend on these 3 classes.

To allow the player an option to travel, an action called Travel Action is created. Since the Travel Action is an action, this class will inherit the Action class since this action is also a type of action and the action abstract class has all the common attributes of an action. Thus we can adhere to the Do Not Repeat Yourself Principle. The Travel Action has an association with the travelable class since the player will use the travelable object to travel after executing the travel action. Subsequently the travel action will have a dependency to the actor and map since when executing the action it will need to move the player from a map to another map and the Golden Fog Door. Finally, to allow the player to know which map they are in, the FancyMessage class is used to output an ascii art that displays the map name. So the TravelAction will have a dependency to the FancyMessage class.

This design adheres to the Liskov Substitution Principle, Do Not Repeat Yourself, Dependency Inversion Principle and Interface Segregation Principle. For example, the Golden Fog Door is a ground while the Travel Action is an action so it should work like its parent class and exhibit all behaviour of the Ground or Action class. The Interface Segregation Principle can be seen too through the usage of Travelable interface since the ground that allows the player to travel will just need to implement the interface to have the methods to travel while other grounds which do not have that functionalities will not need to implement the methods. Through the TravelAction class, we can see that it associates with the Travelable interface rather than the Golden Fog Door, this will help reduce the coupling between the 2 concrete classes that might result in a large fix needed if there is any code changes. Finally, the Do Not Repeat Yourself Principle can be preserved through inheritance and implementing interfaces. Through this design, we can extend the system to have or travelable grounds that the player can interact with to travel. Since the new class just has to extend the Ground class and implement the Travelable interface according to their functionality. If the new map has a name we can just add the ascii art of the map name to the FancyMessage class to print out the map name.