Design Rationale

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

Requirement 1

This UML class diagram will showcase the game system in the types of classes in the game. The goal of the class system is to create a system that includes behaviours, actions, enemies, enemy weapons and players covered in the game.

To begin with, the enemies (Heavy Skeletal Swordsman, Lone Wolf and Giant Crab, Pile of bones) plays the role of foes in the game which extends the Enemy abstract class as the enemies 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 Enemy abstract class extends the Actor class 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 grounds that spawns the enemies are Graveyard, Gust of Wind and Puddle of Water which extends the Ground abstract class as they are a type of ground. These spawning grounds have a dependency with location as they need the coordinates of the map for spawning enemies. Similarly with the Enemy abstract class, it not only reduces code repetition by inheriting a parent class as the spawning grounds share similar attributes and methods, it also eases the addition of new spawning grounds that might be added in future implementations.

Thirdly, Actions is an abstract class that enemies have a dependency on. It takes care of the actor's action 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 AOE_AttackAction and DespawnAction , the player and enemies have a dependency with them as they rely on them to perform AOE attacks and despawns.The reason for the creation of AOE_AttackAction and DespawnAction was to accommodate actors with weapons or enemies with the intrinsic weapons to be able to perform an area of effect attack in the game if the actor has the capability to do so and able to despawn when their health points hit zero. 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.

Enemy weapons such as Grossmesser are weapons carried by Heavy Skeleton Swordsman. It extends the WeaponItem class and is sellable by implementing the interface Sellable. Therefore it preserves the Interface Segregation Principle as it implements the interfaces that it needs. In addition, the Grossmesser has an association with CapabilitySet, since the holder can perform a spinning attack onto the opponents. On the other hand, Enemies such as Lone Wolf and Giant Crab relies on Intrinsic Weapons to perform their attacks, thus having a dependency on them. Some actors play a role of an enemy having a capability to perform an area of effect by using their intrinsic weapons, explains the association of Actors to capabilitySet. 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 and AOE_AttackBehaviour has a dependency with AttackAction and AOE_AttackAction respectively as the behaviour depends on whether the enemy is going to perform an attack or AOE attack to their rivals. These behaviours implement 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.

The changes done in the design for Assignment 2 is that we removed EnemyDeathAction and PlayerDeathAction since we noticed that DeathAction takes care of the death of all actors and all actors will despawn when their hit points hit zero regardless of being a player or enemy. Besides that, we added Enemy abstract class to hold the similar properties shared by the enemies and also benefits us for future addition of enemies. The associations with the Enum status and species were also added to aid the Enemy abstract class to differentiate the species and status of different enemies. The interface Sellable was also added to make the enemy weapon, Grossmesser sellable to the trader. AOE_AttackAction and AOE_SkillBehaviour was also added to allow the actors to perform an area of effect attack when possible.

Requirement 2

This UML class diagram will showcase the trading system of the game using runes that the players will have which is the currency in this game and how a player can obtain runes. The goal of this design is to allow the players to trade weapons such as buy or sell weapons by using runes and to get runes from various sources while having modularity and reusability.

To begin with, although the rune that the player has should be an item, in this design we did not extend the item abstract class since we decided to make the runes a special kind of item that the system will take care of. Therefore, a RuneManager class has also been created to manage any operation that involves runes such as increasing or decreasing the rune amount. This is because it is not the player's responsibility to handle any rune operation but the system. Hence why there is also a need to have an association between the Rune class and the RuneManager Class. The RuneManager should associate itself to ensure that there is only one instance of it only because there should be only one manager handling the rune operations just like in real life where there is only one manager in a store. Additionally, the Player class has no association with the Rune class since the RuneManager handles the runes as it is a special type of item where it's a type of item that is handled by the system. Alternatively, we can add the runes into the player's inventory but if we want to obtain the runes we will increase the complexity of the function as we need to loop through the player's inventory to find the runes. We can simplify this by using a RuneManager to store the player's rune so we can access it quicker.

This design on the runes adheres to the Single Responsibility Principle as we created a manager class to handle the runes which is a system item rather than the Player class handling the rune operations. Therefore we can prevent Player class from being a God class by handling a system item. The advantage of this design is that we can separate the responsibility of the player to just play the game and let any underlying responsibility be delegated to the system. However, if there are more players the RuneManager might need to manage the runes of all the different players. To solve this issue, we can create a hashmap to store the runes of the players by using a key. Thus, we can extend the game to allow multiple players through a hashmap implementation.

Next, the trader that trades weapons are actors too so they should extend the actor class to inherit all attributes of an actor. A trader is also able to trade weapons which allow players to buy or sell weapons so they will have a dependency to a BuyWeaponAction and SellWeaponAction class that extends 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 adhere to the Do Not Repeat principle. Alternatively we can create a single action called tradeAction that does both the trading procedure but to adhere to the Single Responsibility Principle, it is best to separate the 2 trading actions into separate classes. Subsequently, the Player class will have a dependency to a BuyWeaponAction and SellWeapon class since they can buy/sell weapons from the traders. Lastly, the trader in this system also cannot move nor be able to be attacked, attack or die. Therefore, the trader class has a dependency on DoNothingAction but not to AttackAction or DeathAction. Alternatively, we can create an interface called Mortality so we can differentiate who can die or not. Subsequently, since some weapons are sellable or purchasable, the weapons should implement two interfaces, Sellable and Purchasable if they are sellable or purchasable by the Trader. This adheres to the Interface Segregation Principle since only the weapons will not need to implement a purchase/sell method if they are not purchasable or sellable. The Trader class will also have to associate with the Purchasable and Sellable Interface since the traders would need to know what weapons can be sold to the player or what weapons can be sold to itself.

This design on the trader adheres to the Do Not Repeat Yourself Principle, Single Responsibility Principle and the Interface Segregation Principle through the usage of inheritance and Interfaces. For example, the trading action is separated into 2 actions so that the buying and selling action implementations are independent from each other in their own class. Therefore, we can adhere to the Single Responsibility Principle as the buy/sell action has its own class that implements the execution. Therefore, we can prevent any God classes from appearing. Furthermore though using interfaces, we can let classes to only implement the methods that they need and not be forced upon methods that are irrelevant to them. Through this design, we can extend the system to accommodate more weapons. Since we just need to implement the right interface for the weapon and add it to the trader purchasable and sellable inventory. Hence, the trader can create a selling/buying action for the newly created weapons in the future. However, the trader class is not closed to modification as it needs to add code to add the newly created Sellable/Purchasable weapon.

Players can sell their weapons in their inventory so it has an association with the WeaponItem Class. The player has a collection of WeaponItems instead of the weapon class to reduce coupling between the classes and prevent violating the Dependency Inversion Principle. Additionally, if there are additional weapons we do not need to fix the Player class to accommodate the new weapons

Next, when enemies die they will give runes to players, so they should have a dependency to the DeathAction class. Furthermore, Heavy Skeleton Swordsman will drop its weapon that can be sold to a trader after death, the DeathAction class should have a dependency on the DropWeaponAction class. Subsequently, players are able to pick up said drop weapon to sell so players will have a dependency to PickUpWeaponAction class. Lastly, the DeathAction, BuyWeaponAction, SellWeaponAction will be dependent on Rune Manager since these actions involve runes.

This design on how the player is able to obtain runes from enemies adheres to the Dependency Inversion Principle. This allows to reduce coupling between classes that rely on each other. So when the low level class changes, the high level class will not need to be modified much. For example, the Player and the Weapon class is abstracted through an abstract class called WeaponItem so we can just use this class type to reference any weapons. This will inadvertently help preserve the Liskov Substitution Principle as the Weapon class extends the WeaponItem class so anything the WeaponItem class can do the Weapon class can do. Therefore the player just needs to have a collection of WeaponItem type objects rather than having each weapon as a separate attribute in the Player class which will lead to code smell. So in the future if there are more weapons and the players can obtain it then the weapons just need to extend WeaponItem class and the player can add it to the weapon inventory. Then, the player just needs to use the WeaponItem class to reference the new weapon to execute some action so we can reduce the coupling between the player and the newly created Weapon.

The changes done in the design for Assignment 2 is that we have removed the need to separate the player death and enemy death, the usage of purchasable and sellable interface and the addition of the weapon classes into the design. We remove the PlayerDeathAction and EnemyDeathAction class since during the implementation we realise that the DeathAction class is not a God class as it governs over the death of an actor not the player or enemy. Thus there is not a need to create a separate class for the death of a player or enemy. This change will not violate the Single Responsibility Principle as the DeathAction class still handles the death but the subject is now the actor rather than the player or enemy. Additionally, we added the usage of purchasable and sellable interface as some weapons are not purchasable or sellable therefore we created these interfaces to adhere to the Interface Segregation Principle so only Sellable weapons can be sold and only Purchasable weapons can be bought as they have the method needed. Finally we add the weapon classes into the design to better demonstrate the trading system by showing the relationship of the sellable or purchasable weapons to the Trader, Player, BuyWeaponAction and the SellWeaponAction.

This UML class diagram provides modularity, extensibility and reusability by using inheritance and polymorphism as stated above through following the Single Responsibility Principle, Dependency Inversion Principle and Do Not Repeat Yourself Principle.

Requirement 3

This UML class diagram mainly revolves on showcasing the reset system of the game using items, a unique ground, and the actors. The goal of this design is to mainly implement a reset with specific effects that is dependent on the way a reset of the game is to be called while proving modularity, extensibility and maintainability.

First of all, the Player is an actor in the game. Therefore the Player class will extend the Actor class to inherit all its properties which follows the DRY principle. Each player will have a unique item that cannot be dropped called FlaskOfCrimsonTears. This item is a class that extends the Item abstract class as it should have the basic attributes of an item. As this item is also consumable by the player, it will implement the Consumable interface and also have a dependency to the ConsumeAction class which extends the Action abstract class, to have all the attributes an action should have. It also has a dependency on the actor class as the consumption of the item will have an effect on the actor such as healing or buffing them. Having a Consumable interface also follows the Interface Segregation Principle as some items might not be consumable so they would not need to implement any method for consuming it. The advantage of this design is that having a Consumable interface allows for extensibility as there might be additional items in the future that can be consumed such as a ManaPotion to replenish the player’s mana if magic was added to the game.

In the game, there will be a unique ground called the SiteOfLostGrace, with the first site of grace being called "The First Step". Since this class is a unique ground, it will extend the Ground class. The player is also allowed to rest on this and with that, both the SiteOfLostGrace and Player class will be having a dependency on RestAction which extends the Action abstract class. Having TheSiteOfLostGrace and Rest action extends the Ground and Action class respectively follows the DRY principle. Therefore, there will be less duplicated codes. Moreover, if there are more grounds that we can rest upon we just need to let that class extend grounds and return the RestAction to the player.

Now onto the game reset. First off, as all enemies will despawn and the player's hit points along with the number of consumptions of the FlaskOfCrimsonTears will be set to its maximum value again when a reset occurs, the FlaskOfCrimsonTears, and the Enemy and Player class will implement the Resettable interface. Having a Resettable interface to allow all the classes that should be able to be reset implement it follows the Interface Segregation Principle. As the Enemy class consists of all the enemies, each enemy class extends the Enemy abstract class, in which the Enemy class will extend the Actor abstract class, to have its properties and avoid repetitions. Having the Enemy and the FlaskOfCrimsonTears class extend the Actors and Items class respectively also follows the Don't Repeat Yourself (DRY) Principle as certain attributes will be present in every actor of the game. The reason Resettable is an interface is because not everything in the game will be influenced by a game reset. A ResetManager class was also created that has an association to the Resettable interface with the purpose to manage the resettable entities and any operation involving the reset. 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. Having the Resettable interface also 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. This design is good as we have created a manager class to manage the reset rather than the resettable entities itself so we can adhere to the Single Responsibility Principle as well. Furthermore, for future extension, like more resettable entities we can just let them implement resettable and register them into the resetManager so they can be resetted too if a reset is needed.

Next, a reset will occur when the player "dies". As the death of the player causes a reset, the DeathAction will have a dependency on the ResetAction. The reason for having a DeathAction class is to handle when the player or enemies die. Besides that, when a player rests at the SiteOfLostGrace, a reset occurs and because of that the RestAction class will have a dependency with the ResetManager class. The DeathAction, ResetAction and RestAction will all be extending the Action class, thus following the DRY principle.

Now for the runes. A rune is an item and as such extends the abstract class Item to have the attributes an item should be having which follows the DRY principle. A RuneManager class was also created that has an association to the Runes class. The purpose of the RuneManager is to manage any operation involving runes, such as increasing and decreasing the amount of runes, or dropping the runes. This is due to the fact that the system is in charge of managing any rune operations. The Rune Manager also associates itself to ensure that there is only one instance of it. From this, the design on the runes adheres to the Single Responsibility Principle as a manager class was created which handles the runes which is a system item rather than the Player class handling the rune operations. Therefore, the Player class can be prevented from being a God class by handling a system item. The design advantage is that we can separate the responsibility of the player to just play the game and let any underlying responsibility be delegated to the system. However, if there are more players, the RuneManager might need to manage the runes of all the different players. To solve this issue, we can create a hashmap to store the runes of the players by using a key. Thus, we can extend the game to allow multiple players through a hashmap implementation.

When the player dies, the runes will be dropped at the location of the player at the previous turn. This is managed by the RuneManager which saves the current location of the player at each turn which will be saved as the rune location where the rune will be dropped along with the rune count be set to 0 once the runes are dropped. Thus, the Player class will have a dependency on the RuneManager class as it manages the dropping of runes as well and also reduces the rune count of the player. The disadvantage for this design is it is not extensible for multiplayer since the rune location and player rune is only one in the RuneManager but can be extended later through hashmap to store the location and runes which is linked to the different players.

When the player stands on the dropped runes, the player is given the option to retrieve the runes, hence the Player class will have a dependency to the RecoverRuneAction which extends the Action class, following the DRY principle. The RecoverRuneAction class will be responsible in handling the pickup of the rune from the ground by the player and calling the

RuneManager class to update the rune count of the player with the amount that was picked up, thus also having a dependency to the RuneManager class. The reason RecoverRuneAction does not extend PickUpAction instead is because while runes are an item, it will not be in the player's inventory which makes the rune pick up to be different from the other items. The runes will not be in the player's inventory as the characteristics of a rune is different from a normal item such as weapons as the runes will be dropped upon death, and it also has its own manager to manage any changes that will be done on it such as being increased or decreased. This design to create a RecoverRuneAction is better since in our design the runes are a special type of item which is part of the system rather than the player. Therefore we created the RecoverRuneAction that should extend Action and not PickUpAction as rune is part of the system and pickUpAction is for the items that truly belong to the player.

If the player dies once more before retrieving the dropped runes, the runes will despawn hence the DeathAction will have a dependency to the RuneManager as the RuneManager class manages any operation involving runes.

Multiple changes were made during Assignment 2 which improved upon Assignment 1 based on the feedback we were given. One such change was the addition of a ResetAction which will be called upon the death of the player. The ResetAction will spawn the player at the last SiteOfLostGrace that they had rested on by getting the spawn point location from the ResetManager and reset the other entity such as enemies by despawning them and refilling the Flask Of Crimson Tears. We added this ResetAction as we realise that the ResetManager is unable to implement these reset actions as they lack information on the resettable entity. Another change that was implemented is the addition of the Consumables interface. As there is an item that can be consumed by the player which is the FlaskOfCrimsonTears, a Consumable interface was made to handle any consume action that is to be called. Besides that, a RecoverRuneAction was also created to allow the player to pick up the dropped runes on the map as while the rune is an item, a rune is a special kind of item which does not get added to the player's inventory hence it not using the the PickUpItemAction and instead uses the RecoverRuneAction and RuneManager to update the rune count.

This UML diagram provides extensibility, modularity, and reusability through polymorphism and inheritance as stated above. By using the parent class as the type, we can also allow extensibility through polymorphism by allowing the addition of new subclasses without modification on the code since polymorphism allows a code to perform a single action in different ways. Hence, the design adheres to the Open-Closed Principle, Dependency Inversion Principle, Do Not Repeat Yourself (DRY) Principle, and the Single Responsibility Principle.

Requirement 4

This UML class diagram will showcase the game system in the types of classes in the game. The goal of the class system is to create a system that allows the player to choose a combat class that has different weapons and skills while allowing the possibility of extending it and be easier to maintain.

First of all the player is an actor in the game. Therefore the Player class will extend the Actor class to inherit all its properties and avoid repetitions. Each player can also choose between 3 different classes, so 3 concrete classes called Samurai, Bandit and Wretch are created that will extend the Player class since they are also players. This design will follow the Single Responsibility Principle as each class (Player class) is only responsible for one type of class. For example, the Bandit class handles everything that relates to Bandit while Samurai class handles everything that relates to Samurai 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 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. Moreover, the Player class has a dependency to the Menu class since they are able to use the menu to select their actions.

Besides that, the design allows the Open-Closed Principle and Liskov Substitution Principle to be preserved since by making the Player class abstract and letting the combat classes extend it, we can prevent any modification to the Player class when allowing more combat classes to exist. This is because rather than letting the Player class be responsible for all of the combat classes, we created 3 different classes that are responsible for each class and extends the Player class. This is because the 3 combat roles are also players too, Therefore, if there is a need for a new type of player like a Mage then we can just create another combat class that extends the Player class. This helps reduce any duplicate codes through inheritance and allow the combat class system to be extended without any modification needed to the existing classes. Furthermore, the extension will also adhere to the aforementioned Single Responsibility Principle as the Mage class just has to handle the logic of a mage player in a game. Hence, this design provides modularity, extensibility and reusability through polymorphism and inheritance.

In the game, each class has their own unique signature weapon. Therefore I have created 3 concrete classes to handle the weapons attributes and methods called Uchigatana, GreatKnife and Club class that extends from WeaponItem abstract class since all weapons are a weapon item so they share common attributes. These 3 weapon classes are also dependent on each of their related combat classes since each combat class starts with their 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, hence that is why the UnsheatheAction and

QuickStepAction class are created. Both of these classes extend the Action class as they are an action of themselves so we can reduce code duplication. Alternatively, we can create a new class called skill action that each weapon has to provide the player the ability to do a skill if the conditions have been met. However, by creating a skill action that does all the possible skills will make this class into a God class that sequentially will violate the Single Responsibility Principle. Therefore we separated the two skills into their own classes that are responsible for defining the logic specific to each skill. For QuickStepAction, it is an action for the skill where the actor will move away after attacking so it has a dependency with the location class and move actor class.

Furthermore, since a player can choose to use their fist to fight, it has a dependency with the Intrinsic Weapon class. Alternatively, we can associate a player with the Intrinsic Weapon class since dependency is a weak relationship and any entity has an intrinsic weapon. However to avoid high coupling between the player and the intrinsic weapon class we should make them dependent on each other instead so the Dependency Inversion Principle can be preserved.

The Player class is associated with the WeaponItem class instead of the 3 weapon class to reduce coupling. This is due to the fact of polymorphism which allows us to provide different implementation according to the type of object without the usage of if-else statement which is not recommended since it will make the Player class and the 3 weapons class to be tightly coupled. Additionally, if there are additional weapons we do not need to fix the Player class to accommodate the new weapons Therefore, we can obey the Dependency Inversion Principle and the Open-Closed Principle since the classes are hidden behind abstraction. Less effort is also required when modifying the system since the classes are independent from each other.

This design follows the Single Responsibility Principle, Dependency Inversion Principle and Open-Closed Principle. Thus it can provide modularity, extensibility and reusability through polymorphism and inheritance. For example, rather than creating a single skill action for the weapons, the unsheathe skill and the quick step skill are created in their own classes where their implementations are done in the execute method which can be overridden as they both extend the action class. Hence, any new skills added to the game will just have to implement their own execute method without modifying any of the existing classes. Furthermore, the players and the weapons are linked together through a layer of abstraction which is the weaponItem class. So the player class and the weapon classes are not tightly coupled so that if any changes are made to the weapon class there won't be a need to modify the player class. Moreover, the implementation of each of the weapons can be independent of each other. Hence this design allows modularity as each class handles a single responsibility that can be used together to perform a larger application. This will also reduce duplicated code as there will be common attributes and be easier to maintain and test as we just only have to focus on a single module of a larger application. By using the parent class as the type, we can also allow extensibility through polymorphism by allowing the addition of new subclasses without modification on the code since polymorphism allows a code to perform a single action in different ways.

The changes done in the design for Assignment 2 is that we have removed the ClassType enum class and added the UnsheatheAction and QuickStepAction classes. We have removed the ClassType enum class as during implementation we realised that the class does not provide any usage in developing the system. Therefore we removed the class since it's a useless class and will accumulate code smell if left unattended. Next we added the UnsheatheAction and QuickStepAction as during implementation we found that the AttackAction is unable to accommodate the details of these skills. Hence, there is a need to create their own class to implement the skill.

Requirement 5

This UML class diagram will showcase the game system in the types of classes in the game. The goal of the class system is to create a system that includes behaviours, actions, enemies, enemy weapons and players covered in the game.

To begin with, the enemies (Heavy Skeletal Swordsman, Lone Wolf and Giant Crab, Pile of bones, Giant Dog, Skeleton Bandit, Giant Crayfish) plays the role of foes in the game which extends the Enemy abstract class as the enemies 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 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 grounds that spawns the enemies are Graveyard, Gust of Wind and Puddle of Water which extends the abstract class SpawningGrounds, 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. EnemyFactory, an abstract class that takes the role of spawning enemies and has a dependency with location to know which side to spawn the enemies and has an association with actor to know which actors (enemies) to spawn, and has 2 subclasses WestEnemyFactory and EastEnemyFactory which is used to spawn enemies on their grounds. 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 need not modify information on our grounds if we have to take account of new enemies in future implementations. Furthermore, EnemyFactory being abstract would reduce the time needed for more conditions on spawning new enemies on specific parts of the map by creating a child class to take account for future requirements. Therefore this design will adhere to the Do Not Repeat Yourself Principle. For future extension, if there is another ground that can spawn enemies, the classes then just have to extend the spawning ground and its enemy be added into the respective factory.

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 AOE_AttackAction and DespawnAction , the player and enemies have a dependency with them as they rely on them to perform AOE attacks and despawns. 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.

Enemy weapons such as Grossmesser and Scimitar are weapons carried by Heavy Skeleton Swordsman and Skeleton Bandit respectively. It extends the WeaponItem class, and Grossmesser is sellable whereas Scimitar 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. In addition, both Grossmesser and Scimitar have an association with CapabilitySet, since the holder can perform a spinning attack onto the opponents. On the other hand, Enemies such as Giant Dog and Lone Wolf rely on Intrinsic Weapons to perform their attacks, thus having a dependency on them. Some actors play a role of an enemy having a capability to perform an area of effect by using their intrinsic weapons, explains the association of Actors to capabilitySet. 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 and AOE_SkillBehaviour has a dependency with AttackAction and AOE_AttackAction respectively as the behaviour depends on whether the enemy is going to perform an attack or AOE attack to their rivals. These behaviours implement 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.

The changes done in the design for Assignment 2 is that we removed EnemyDeathAction and PlayerDeathAction since we noticed that DeathAction takes care of the death of all actors and all actors will despawn when their hit points hit zero regardless of being a player or enemy. Besides that, the associations of the Enemy abstract class with the Enum status and species were also added to aid the Enemy abstract class to differentiate the species and status of different enemies. The interface Sellable and purchasable was also added to make the enemy weapon, Grossmesser sellable to the trader, Scimitar sellable and purchasable to the trader. AOE_AttackAction and AOE_SkillBehaviour was also added to allow the actors to perform an area of effect attack when possible. The EnemyFactory abstract class was added to aid in spawning enemies by its subclasses, WestEnemyFactory and EastEnemyFactory. Lastly, SpawningGrounds abstract class was added under the consideration of similar properties shared by each spawning ground and also benefits us for any future addition of spawning grounds.