Design Rationale

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

Requirement 1

This UML class diagram will showcase the relationship between the enemies and players as actors in the game. The goal of the class system is to show the characteristics, abilities and behaviours of the actors in the game in modularity.

To begin with, the enemies (Heavy Skeletal Swordsman, Lone Wolf and Giant Crab) that play the role of foes in the game which extends the Actor class as the enemies are characters in the game thus inheriting all its properties, supporting the DRY principle to avoid repetitions for classes that share similar attributes and methods. Graveyard, Gust of Wind and Puddle of Water extends abstract class Ground as they are a type of ground. Location has an association with GameMap to get the coordinates in the game, where GameMap has an association with ActorLocationIterator to provide the coordinates for the actors. All enemies also have a dependency with the abstract class Actions, that takes care of the actor's action in the game such as DeathAction. DeathAction is made abstract and has two subclasses, EnemyDeathAction and PlayerDeathAction. The creation of these two subclasses are used to differentiate the death action in the game of the enemies and the player. Both of them handle the death action of the enemies and the player, so the player and the enemies depend on them respectively. The enemies have an association with the EnemyWeapon as it handles the source of damage dealt to the players or actors in the game. However, not all have an association with EnemyWeapons as they do not equip a weapon. For those enemies without any weapons, they depend on IntrinsicWeapon to deal damage to their opponents in the game. Moreover, EnemyWeapon extends the abstract class WeaponItem as they are a weapon in the game and they also extend from the Item abstract class since weapons exist as Items in the game as they are portable hence greatly reducing code repetition. Intrinsic Weapon implements the Weapon interface since they are also a type of weapon. It does not extend WeaponItem since the Intrinsic Weapon such as fists are not portable.

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.

CapabilitySet has a dependency on the Capable interface. Actors and Weapons that have an association with the capabilitySet are used to take account of abilities that the Classes have. It stores all the information for all the weapons including intrinsic weapons that contain any special ability. Furthermore, Capable ensures that when more associations are linked

with CapabilitySet, we do not have to fix CapabilitySet to accommodate for new enemies or weapons added into the game by the Open Closed Principle.

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 enemies while having modularity and reusability.

To begin with, a rune is an item so it should extend the item abstract class to inherit all attributes of an item. A Rune Manager class has also been created to manage any operation that involves runes such as increase, decrease rune amount due to trading. 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 Rune Manager Class. The Rune Manager 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.

Since a player can sell weapons to gain runes, buy weapons with runes and store runes, they need a rune manager and an inventory of items. So the Player class has an association with the Item and Rune Manager class. It associates with Item rather than Rune since a player can store other items. If we let the Player class associate with every item then we will have multiple dependencies between the Player class and the items which extends the Item class and it is not recommended. So to reduce coupling we associate the Player class with the Item class instead. This design will obey the Dependency Inversion Principle since it breaks down the dependencies between Player and the items. Moreover, it prevents any violation of the Open-Closed Principle as we do not need to use an if-else statement to check the type of item. Furthermore, they will also have a dependency to a BuyWeaponAction and SellWeapon class since they can buy/sell weapons that 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.

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. 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.

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 a EnemyDeathAction class. Furthermore, Heavy Skeleton Swordsman will drop its weapon that can be sold to a trader after death so they should have a dependency on the DropWeaponAction class. But other enemies should not have it since they don't have any weapons to drop. Subsequently, players are able to pick up said drop weapon to sell so players will have a dependency to PickUpWeaponAction class. Lastly, the EnemyDeathAction, BuyItemAction, SellItemAction will be dependent on Rune Manager since these actions involve runes.

This UML class diagram provides modularity, extensibility and reusability by using inheritance and polymorphism as stated above. However, the disadvantage of this design is that there are too many dependencies which is a weak relationship. We can solve this problem by creating an association between the entities and the action class through a manager class which will handle the actions. Consequently, this design follows 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 player. 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.

Starting with FlaskOfCrimsonTears, it is a starting consumable item of the player that will never be dropped. 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, 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.

Next we have the SiteOfLostGrace. This is a unique ground and as such will extend the Ground class to have all the attributes of a ground. The player is also allowed to rest on this and with that, the SiteOfLostGrace and Player classes will be having a dependency on RestAction which extends the Action abstract class.

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 all the enemies and Player class which extend the abstract class Actors due to each of them being an actor, will implement the Resettable interface. Having all the enemies and the Player class extend Actors 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 any operation involving the reset. The ResetManager also associates itself to ensure that there is only one instance of it because changes will be made to any classes that will be influenced by a reset which leads to the need of only one reset action. 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 the 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.

Next, a reset will occur when the player "dies". Hence, the Player class will have a dependency on the PlayerDeathAction class which extends the DeathAction class that extends the Action class due to it being an action itself. As the death of the player causes a reset, the PlayerDeathAction will also have a dependency on the ResetManager. The reason for having a PlayerDeathAction class instead of just a DeathAction class is because the effects of a player's death is different from when an

enemy is killed. This will adhere to the Single Responsibility Principle to avoid the DeathAction class being a god class. 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.

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. 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 or decreasing the amount of 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.

When the player dies, their runes will be left on the ground and to do so, the Player class will have a dependency on the DropItemAction class which extends DropAction class that extends the abstract class Action as it itself is an action. The reason for also having a dependency to DropItemAction is because the rune will be dropped and runes are an item.

When the player stands on the dropped runes, the player is given the option to retrieve the runes hence the Player class also having a dependency to the PickUpItemAction class which extends PickUpAction class that also extends the abstract class Action due to being an action as well. The reason for also having a dependency to PickUpItemAction is because the rune will be picked up and runes are an item.

When the player drops or retrieves runes, the total amount of runes the player will have either decreases or increases respectively. As such, both the DropItemAction and PickUpItemAction class have a dependency to the RuneManager class.

If the player dies once more before retrieving the dropped runes, the runes will despawn hence the Runes class implementing the Resettable interface as a reset will have an effect on the state of the runes.

This UML diagram provides extensibility, modularity, and reusability through polymorphism and inheritance as stated above. For example, rather than creating a class that is responsible for death of all the enemies and players, a PlayerDeathAction class was created that inherits the abstract class DeathAction attributes as the effects of a player's death is different from when an enemy is killed which achieves modularity. 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 Responsible 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. 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 as a Samurai is also a player. 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 choose their classes.

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. Furthermore, since a weapon is also an item and a weapon, the WeaponItem abstract class also extends from the Item abstract class and implements the Weapon Interface. Additionally, each weapon may or may not have a skill which is a capability, so it has an association with the CapabilitySet class. 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 which relates closely to the attack action and Capability Set this will violate the Do Not Repeat Yourself Principle. Furthermore, since a player can choose to use their fist to fight, it has a dependency with the Intrinsic Weapon class which extends the Weapon interface since a fist is also a weapon. Alternatively, we can associate a player with the Intrinsic Weapon class since dependency is a weak relationship and any entity has an intrinsic weapon.

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 UML diagram provides modularity, extensibility and reusability through polymorphism and inheritance. For example, rather than creating a class that is responsible for all of the classes, we created 3 different classes that are responsible for each class. This 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. Consequently, the design will adhere to the Open-Closed Principle, Single Responsibility Principle, Dependency Inversion Principle and the Do Not Repeat Yourself Principle.

Requirement 5

This UML class diagram will showcase the relationship between the enemies and players as actors in the game. The goal of the class system is to show the characteristics, abilities and behaviours of the actors in the game in modularity.

To begin with, the enemies (Heavy Skeletal Swordsman, Lone Wolf, Giant Crab, Skeletal Bandit, Giant Dog, Giant Crayfish) that play the role of foes in the game which extends the Bone, Canine and Crustaceans class. By creating these abstract Species classes, it reduces code repetition for enemies of the same kind but different territories such as Lone Wolf and Giant Dog who share the same attributes and methods. Furthermore, these abstract classes extend the Actor class thus inheriting all its properties, supporting the DRY principle for classes that share similar attributes and methods. Graveyard, Gust of Wind and Puddle of Water extends abstract class Ground as they are a type of ground. Location has an association with GameMap to get the coordinates in the game to differentiate the east and west side of the map, where GameMap has an association with ActorLocationIterator to provide the coordinates for the actors. All species also have a dependency with the abstract class Actions, that takes care of the actor's action in the game such as DeathAction. DeathAction is made abstract and has two subclasses, EnemyDeathAction and PlayerDeathAction. The creation of these two subclasses are used to differentiate the death action in the game of the enemies and the player. Hence, the Single Responsibility Principle is adhered. Both of them handle the death action of the enemies and the player, so the player and the enemies depend on them respectively. The enemies have an association with the EnemyWeapon as it handles the source of damage dealt to the players or actors in the game. However, not all have an association with EnemyWeapons as they do not equip a weapon. For those enemies without any weapons, they depend on IntrinsicWeapon to deal damage to their opponents in the game. Moreover, EnemyWeapon extends the abstract class WeaponItem as they are a weapon in the game and they also extend from the Item abstract class since weapons exist as Items in the game as they are portable hence greatly reducing code repetition. Intrinsic Weapon implements the Weapon interface since they are also a type of weapon. It does not extend WeaponItem since the Intrinsic Weapon such as fists are not portable.

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.

CapabilitySet has a dependency on the Capable interface. Actors and Weapons that have an association with the capabilitySet are used to take account of abilities that the Classes have. It stores all the information for all the weapons including intrinsic weapons that contain any special ability. Furthermore, Capable ensures that when more associations are linked

with CapabilitySet, we do not have to fix CapabilitySet to accommodate for new enemies or weapons added into the game by the Open Closed Principle.