Design Rationale

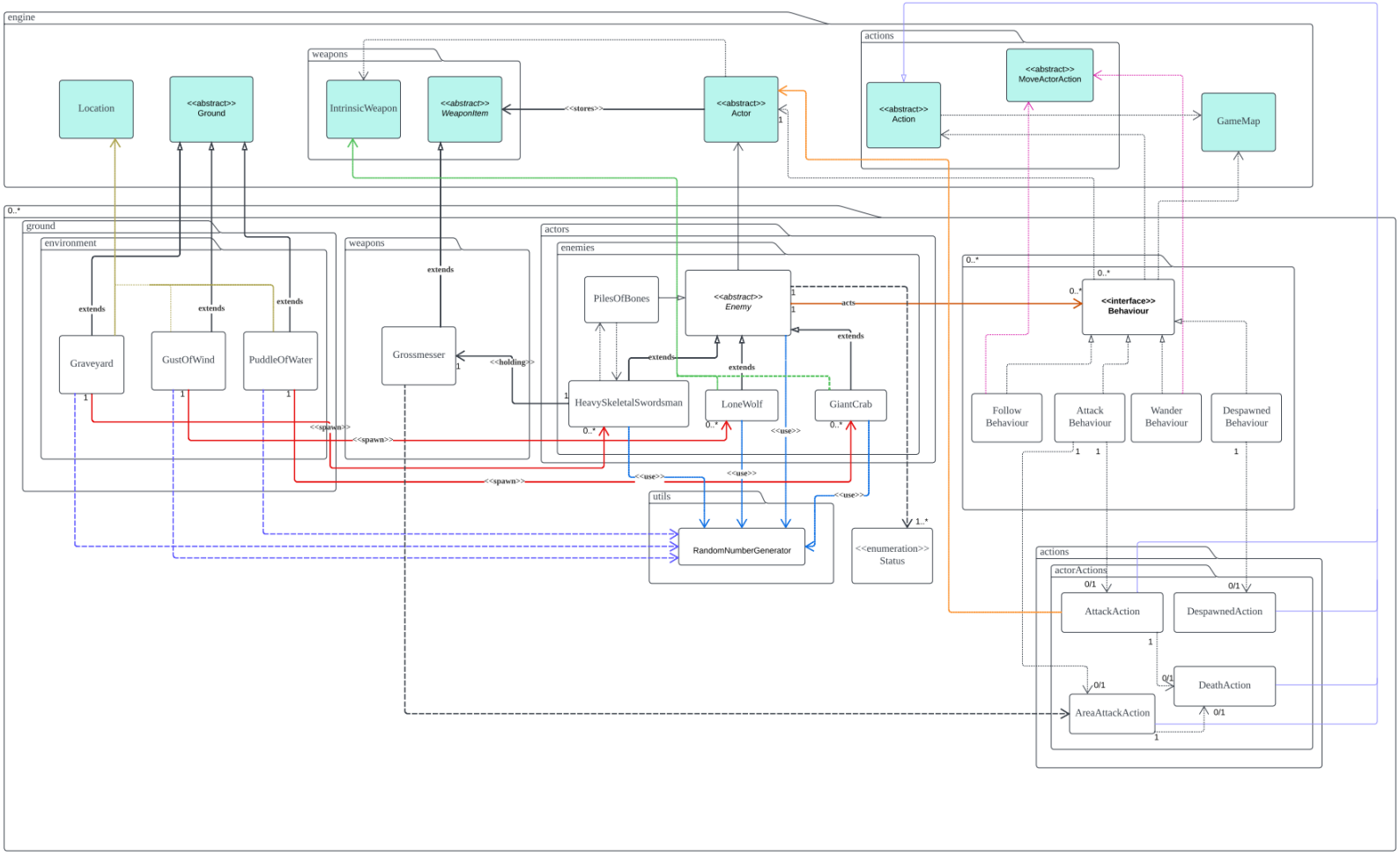
Group 6
Tan Chun Ling | Wan Jack Liang | King Jean Lynn

Design goals

Design rationale is an essential part of the design process, as it provides a clear and concise explanation of the reasoning and decision-making processes behind a particular design or solution. The goal of writing design rationale is to ensure that the design is well-informed, well-justified, and well-documented. To achieve this, our team use principles such as the SOLID principle to guide the design decisions. By documenting the design rationale, we explain why certain decisions were made, what factors were considered, and what pros and cons were weighed, while ensuring that the design adheres to SOLID principles. This can help to ensure that the design is consistent with the project's goals, requirements, and constraints, and can also provide a basis for future improvements or modifications to the design.

Req 1

UML Diagram



A. Environments

Package: game.ground.environment

New classes	Class Responsibility	Design Rationale
Graveyard extends Ground	A class that represents Graveyard which is occupied by "Heavy Skeletal Swordsman' creatures. Dependency: - **Ground**: Graveyard inherit ground class - **Location**: Graveyard spawn the Heavy Skeletal Swordsman by checking the current location - **RandomNumberGenerator:** The chance that an enemy spawn	The design rational of Graveyard class extended the abstract Ground class. Since they share some common attributes and methods. It is logical to abstract these identities to avoid repetition and achieve the **Don't Repeat Yourself** principle. As Graveyard is a ground that can spawn/respawn, we have chosen to give Ground objects that are spawnable the capability RESPAWNABLE in the enumeration Status. This allows us to follow the **Don't Repeat Yourself** design principle using public constants, so when future Ground may also be respawnable, we will have a unified identifier for all such ground. Moreover, Ground features a CapabilitySet which we can use to store the enumeration value. Random Number Generation for spawning also follows the **Single Responsibility Principle** by having dedicated classes responsible for generating random numbers.

GustOfWind extends Ground	A class that represents Gust of Wind which is occupied by “Lone Wolf’ creatures. Dependency: - **Ground**: GustOfWind inherit ground class - **Location**: GustOfWind spawn Lone Wolf by checking the current location - **RandomNumberGenerator:** The chance that an enemy spawn	The design rational of GustOfWind class extended the abstract Ground class as they share common attributes and methods. By extending from Ground class, it will help to achieve the **Don’t Repeat Yourself** principle. As GustOfWind is a ground that can spawn/respawn, we have chosen to give Ground objects that are spawnable the capability RESPAWNABLE in the enumeration Status. This allows us to follow the **Don’t Repeat Yourself** design principle through the use of public constants, so when future Ground may also be respawnable, we will have a unified identifier for all such ground. Moreover, Ground features a CapabilitySet which we can use to store the enumeration value. Random Number Generation for spawning also follows the **Single Responsibility Principle** by having dedicated classes responsible for generating random numbers
PuddleOfWater extends Ground	A class that represents Puddle of Water which is occupied by “Giant Crab”.	PuddleOfWater shares common methods and attributes with Ground. Following the **Don’t Repeat Yourself** principle, we should extend PuddleOfWater class to abstract Ground class as they share similar characteristics.

	Dependency: - **Ground**: PuddleOfWater inherit ground class - **Location**: PuddleOfWater spawn Giant Crab by checking the current location - **RandomNumberGenerator:** The chance that an enemy spawn	As PuddleOfWater is a ground that can spawn/respawn, we have chosen to give Ground objects that are spawnable the capability RESPAWNABLE in the enumeration Status. This allows us to follow the **Don't Repeat Yourself** design principle through the use of public constants, so when future Ground may also be respawnable, we will have a unified identifier for all such ground. Moreover, Ground features a CapabilitySet which we can use to store the enumeration value. Random Number Generation for spawning also follows the **Single Responsibility Principle** by having dedicated classes responsible for generating random numbers

B. <u>Enemies</u>

Package: game.enemies

New classes	Class Responsibility	Design Rationale
Enemy (abstract) extends Actor implement Resettable	An abstract class which acts as the base class for Actors in the Enemies World Dependency: - **Behaviour**: enemies contain a haspMap	To achieve **Open-Closed Principle**, we create an abstract class that include all the characteristics of Enemy, allowing for easier addition of new enemy classes in the future.

	of behaviours - **RandomNumberGenerator**: enemies have a 10% chance of being despawned	By implementing Resettable interface, we can ensure that all enemies can reset. We try to use interface to avoid having God Class in our system, this is also achieving **Single Responsibility Principle**.
HeavySkeletalSwordsman extends Enemy	A class represents Heavy Skeletal Swordsman. Dependency: - **WeaponItem:** the Grossmesser weapon that will be carried by HeavySkeletalSwordsman - **PilesOfBones**: HeavySkeletalSwordsman can become a pile of bones for 3 turns if killed by other enemies or player. - **RandomNumberGenerator**: if killed, the creatues could drop any amount of runes within the range of 35 and 892.	The design rationale of HeavySkeletalSwordsman class is similar to the LoneWolf class. By making this class a subclass of Enemy, we can reduce the number of repeated codes, thus fulfilling the principle of **Don't repeat yourself**. Besides, HeavySkeletalSwordsman will turn itself into Piles Of Bones if killed by enemies or player, therefore we create dependency between Piles of Bones and HeavySkeletalSwordsman.
LoneWolf extends Enemy	A class represents Lone Wolf Dependency: - **IntrinsticWeapons**: bite other creatues - **RandomNumberGenerator**: if killed, the creatues could drop any amount of runes within the range of 55 and 1470.	This class is given in the original system; however, we modify Lone Wolf inherit the Enemyclass to fulfil **Single Responsibility Principle.** The LoneWolf's bite will be implementeed by overriding the getIntrinsicWeapon() method of Actor. Then we can perform the attack by setting the verb and damage of IntrinsicWeapon (hitRate is the default 50%).

GaintCrab extends Enemy	A class represents GaintCrab Dependency: - **IntrinsticWeapons**: slam other creatues - **RandomNumberGenerator**: if killed, the creatues could drop any amount of runes within the range of 318 and 4961.	The design rationale of GaintCrab class is similar to the LoneWolf class. By making this class a subclass of Enemy, we can reduce the number of repeated codes, thus fulfilling the principle of **Don't repeat yourself**. The GiantCrab's slam will be implementeed by overriding the getIntrinsicWeapon() method of Actor. Then we can perform the attack by setting the verb and damage of IntrinsicWeapon (hitRate is the default 50%).
Piles of Bones	A class represents Piles of Bones	Piles of Bones are only created when Heavy Skeletal Swordsman dies. Following the **Single Responsibility Principle**, we create Piles of Bones class to prevent too many responsibilities in Heavy Skeletal Swordsman class.

Package: game.bahaviour

New classes	Class Responsibility	Design Rationale
AttackBehaviour implements Behaviour	A class which implements Behaviour interface and generates AttackAction or AreaAttactAction. Dependency: - **Actor**: the target attack actor	This AttackBehaviour class was created as it represents a behaviour which was not originally modeled in the system. Besides, following the **Single Responsibility Principle**, all enemy behaviors are not merged into one single Behaviour class. If we need to further

	- **GameMap**: the map the actor is on as passed as parameter	modify attack behaviour in the future, we will only need to modify this class. The implementation of the 'getAction()' method in this behavior class also demonstrates the concept of polymorphism, as it relies on the 'Behaviour' interface that is implemented by multiple classes. The playTurn() method in the Actor class can loop through an array of behaviors and call the 'getActioin()' method of each subclass of the 'Behaviour' interface to determine if the specific action (attack behavior for this case) represented by the subclass is possible for the 'Actor' to perform in its current turn. AttackBehaviour implements Behaviour, so can act this behaviour without the player's input
DespawnedBehaviour implements Behaviour	A class that represent the enemy despawn behaviour. Where for each enemy, they have despawn behaviour. Since at each turn, GiantCrabs, LoneWolf and HeavySkeletalSwordsman have 10% chance of being despawned (removed from the map) unless they are following the player. Dependency:	This DespawnedBehaviour class was created as it represents a behaviour which was not originally modeled in the system. This class is similar to AttackBehaviour class where it follows the **Single Responsibility Principle**, so all behaviours are not merged into one single behaviour class. When the enemy is being despawned, it will implement

	- **Actor**: the despawn actor - **GameMap**: the game map containing the actor as passed as parameter	'getAction()' method in the behavior class (**Polymorphism**) and return the DespawnedAction. Moreover, the class will have one private attribute despawnChance (10 to represent 10%), which can reduce **implicit dependency on literals** and better than hardcoding the value. Our game design also makes sure that the enemy can only perform one action at each turn. If it despawn, it will not perform attack action, follow or wander at the same time. DespawnedBehaviour implements Behaviour, so can act this behaviour without the player's input

Package: game.actions.actorActions

New classes	Class Responsibility	Design Rationale
AttackAction extends Action	A class that represents the action of attacking a target. Dependency: - **Actor**: The actor that was targeted, a parameter passed in for the constructor. - **Weapon**: The weapon being used to attack the targeted actor, passed in as parameter for the constructor.	This AttackAction class was created because it represents an action which was not originally modeled in the system. This attack action is not merged into other action class as we are following the **Single Responsibility Principle**. We make the AttackAction class inherit from Action class as both have similar methods that can be reused or overridden. Thus, we can reduce the number of

	- **DeathAction**: death action is called if the enemy is not conscious after attacked.	duplicated code and achieve the principle of **Don't Repeat Yourself**. Enemies would have the AttackAction applied to its action list by default for the Player to attack it, this can be overriden for different behaviour such as LoneWolf. E.g., LoneWolf will be given the capability HOSTILE_TO_ENEMY if they can be attacked by enemy. We can choose to add the AttackAction to the LoneWolf action list if it is hostile to enemy, so other Actors can see the options.
AreaAttackAction extends Action	This class represents the actor attack action, particularly the area surrounding to attack. Dependency: - **GameMap:** the game map containing the actor as passed as parameter - **Weapon**: The weapon being used to attack the actor around the surrounding, passed in as parameter for the constructor. - **DeathAction**: death action is called if the enemy is not conscious after attacked.	This AreaAttackAction was created because it represents an action that was not originally present in the system. Following the **Single Responsibility Principle**, this area attack action is not merged into other action classes. We make the AreaAttackAction class inherits from Action as execute() method and menuDescription() method can be reused but area attack targets all creatures around the surrounding. Similar to AttackAction which targets a single actor to attack, AreaAttackAction targets the surrounding actor to attack.

DespawnedAction extends Action	A class that represents the action when the actor is despawn. Dependency: - **Actor**: The despawn action is applied on an actor. - **GameMap**: the map the actor is on as passed as parameter	This DespawnedAction was created because it represents an action that was not originally present in the system. It is also following **Single Responsibility Principle** where it inherits from Action as both have similar methods. Reduce repeated code Don't Repeat Yourself. When an actor is being despawned, it will implement 'execute()' method and remove the actor from the map.
DeathAction extends Action	A class that represents the action when an actor is dead. Dependency: - **AttackAction**: when attack is executed, check if target is conscious. If not, then DeathAction is executed. - **Actor**: The death action is applied on an actor. - **GameMap**: the map the actor is on as passed as parameter	The DeathAction class is also following Single Responsibility Principle where it inherits from Action as both have similar methods that can be reused or overridden, reduce duplicated code achieving Don't Repeat Yourself. When a death action is called, the implementation of 'execute ()' will remove the actor from the map as it already died.

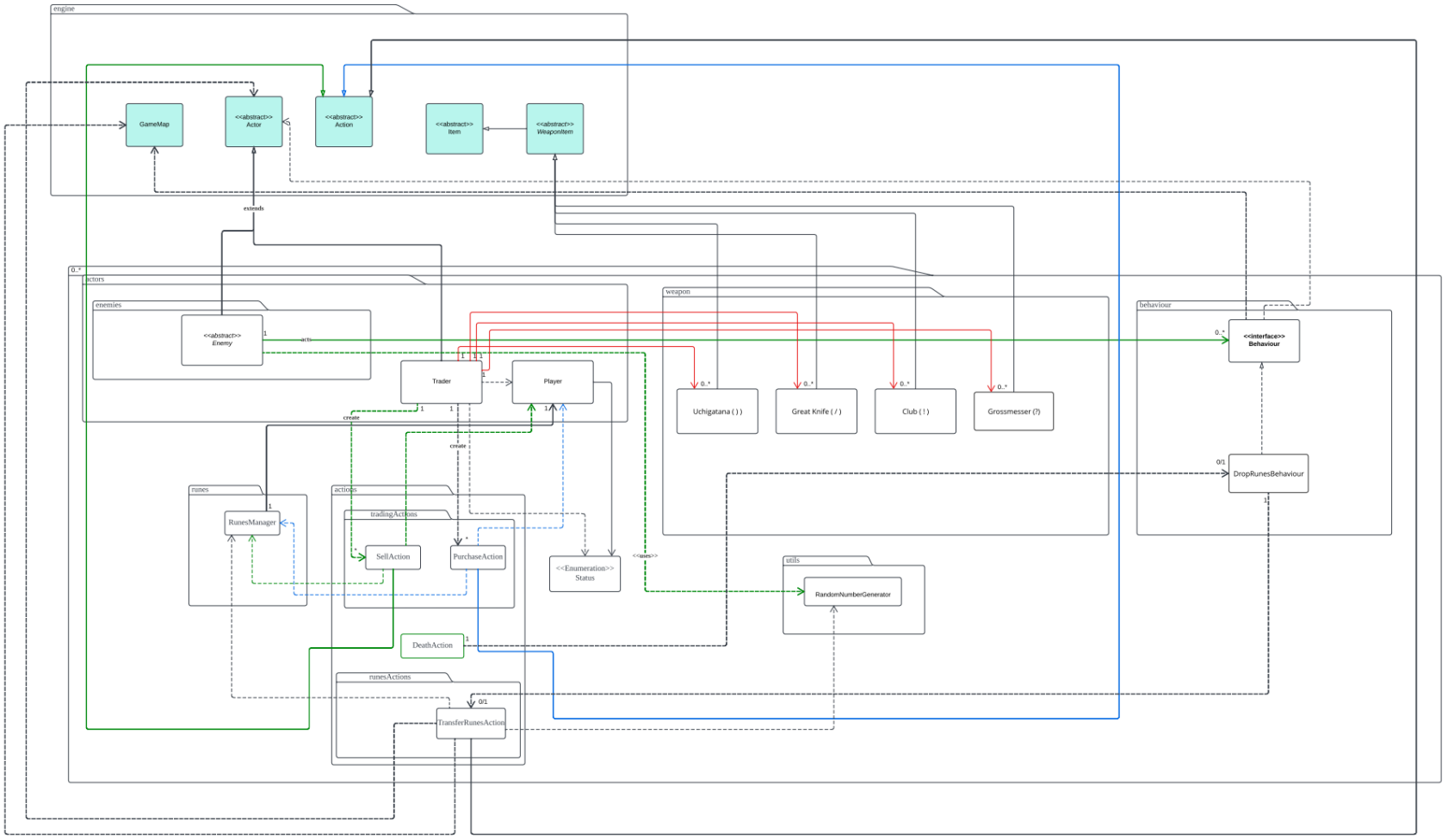
C. Weapons

Package: game.weapons

New classes	Class Responsibility	Design Rationale
Grossmesser extends WeaponItem	A class that represents a weapon called Grossmesser, which is carried around by Heavy Skeletal Swordsman.	This Grossmesser class is created because it represents a weaponItem that does not exist in the original system. Use of WeaponItem (abstract) from engine, **ease of implementation of new weapons** in the future. In addition, they share the common attributes and methods, which can reduce the repeated code achieving the **Don't Repeat Yourself principle.**

Req 2

UML Diagram



A. Enemies

Package: game.actions

New classes	Class Responsibility	Design Rationale
DeathAction extends Action	A class that represents the action when an actor is dead. - **DropRunesBehaviour:** if the attacker is player the actor will perform dropRunesBehaviour(which directly add runes into player)	The DeathAction class is also following **Single Responsibility Principle** where it inherits from Action as both have similar methods that can be reused or overridden, reduce duplicated code achieving Don't Repeat Yourself. When a death action is called, the implementation of 'execute ()' will create the corresponding action/behaviour to perform.

Package: game.behaviours

New classes	Class Responsibility	Design Rationale
DropRunesBehaviour implements Behaviour	A behavior represents when the enemy die. Dependency: - **TransferRunesAction**: the player will only get runes from the hostile creatures if they defeat them directly. - **Behaviour**: DropRunesBehaviour implement Behaviour interface - **DeathAction**: If the attacker is Player and target is enemy, DropRunesBehaviour is called and the	This class is created because it represents a behaviour that is not modelled in the original system. Following the **Single Responsibility Principle**, the dropRunesBehaviour is not merged into other behaviour classes. Besides, this implementation is based on the concept of **Polymorphism**, which allows the playTurn() method in Enemies class to loop through the behaviours array and call the getAction() method of each subclass of

	hostile creatures perform DropRunesBahaviour.	behaviour interface in the array to check if the specific action represented by the subclass is possible to be performed.

Package: game.actions.runesActions

New classes	Class Responsibility	Design Rationale
TransferRunesAction extends Action	An action that perform if the player defeat hostile creatures directly. (The runes transfer directly into player's runes) Dependency: - **Action**: TransferRunesAction inherit Action - **RunesManager**: Call the method inside Runes to add runes into player's wallet. - **DropRunesBehaviour**: TransferRunesAction is created by DropRunesBehaviour. - **RandomNumberGenerator**: A random number that hostile creature dropped when they died - **Actor**: The target actor that used to pick up the runes. - **Gamemap**: The map that current actor at.	This class is created because it represents an Action that is not modelled in the original system. Since the player will directly get runes (without dropping the runes and pick up from player), we manage to create TransferRunesAction, which directly transfer the Runes into player's runesWallet, this fulfils the **Single Responsibility Principle**.

B. <u>Trader</u>

Package: game.actors

New classes	Class Responsibility	Design Rationale
Trader extends Actor	A class represents an ordinary trader, allows the player to purchase or the weapons (or items) Dependency: - **SellAction**: Create different available SellAction into allowableAction (ActionList) - **PurchaseAction**: Create different available PurchaseAction into allowableAction (ActionList) - **Actor:** the actor approach to the trader - **Status:** only the actor has the capability to purchase and sell item can have the actions	In order to allow player to purchase and sell item inside their inventory, the trader class is created, this is because we are trying to fulfil **Single Responsibility Principle** and reducing the repeating code. Besides, we create a sellItem list therefore player can purchase the weapon inside the list. We design in this way so that if there is more than one trader in the map, we can sell different weapon. Such implementation can achieve **Open-Closed Principle**. Similarly, we also create acceptedWeapon list such that player can sell their weapon inside their Weapon inventory to the trader.

Package: game.actions.tradingActions

New classes	Class Responsibility	Design Rationale
PurchaseAction extends Action	A class that represents an action for purchasing a weapon Dependency: - **Action**: PurchaseAction inherit Action class	This class is created because it represents an action that is not modelled in the original system. Based on the **Single Responsibility Principle**, we created purchaseAction that only allow player to purchase the weapon

	- **RunesManager**: Call RunesManager to subtract the value of Runes after purchasing the weapon - **Player**: add weapon item into Player's weapon inventory	since we are avoiding having God class inside our system. Similar to other action class, the implementation of PurchaseAction is also based on the concept of **Polymorphism**, which allows different type of actions store into ActionList that can be use in various ways.
SellAction extends Action	A class that represents an action for selling a sellable item Dependency: - **Action(abstract)**: SellAction inherit Action class - **RunesManager**: Call RunesManager to add the value of Runes after purchasing the weapon - **Player**: remove weapon item into Player's weapon inventory	This class is created because it represents an action that is not modelled in the original system. Based on the **Single Responsibility Principle**, we created sellAction that only allow player to sell the weapon inside their weaponInventory since we are avoiding having God class inside our system. Similar to other action class, the implementation of SellAction is also based on the concept of **Polymorphism**, which allows different type of actions store into ActionList that can be use in various ways

Package: game.utils

New classes	Class Responsibility	Design Rationale
Status (enumeration)	The enum class to give `buff` or `debuff`	We add 'BUYING' status to check if the actor has the capability to have tradingActions. The use of enumration will avoid the excessive use of literals, and therefore will

		improve maintainability and extensibility of the code in the long-term.

C. Weapons

Package: game.weapons

New classes	Class Responsibility	Design Rationale
Uchigatana extends WeaponItem	A class that represents a weapon called Uchigatana Dependency: - **Trader:** Uchigatana will be added into sellItem and acceptedWeapon list	This class is created because it represents a weaponItem that is not existed in the original system. We inherited WeaponItem class because they share the common attributes and methods, which can reduce the repeated code (**Don't Repeat Yourself**) Moreover, we add this weapon into trader's 'sellItem' list and 'acceptedWeapon' list such that this Weapon item can be purchased by player and sold to trader.
GreatKnife extends WeaponItem	A class that represents a weapon called Great Knife Dependency: - **Trader:** GreatKnife will be added into sellItem and acceptedWeapon list	This class is created because it represents a weaponItem that is not existed in the original system. Similar to Uchigatana class, we inherited WeaponItem class since they share the common attributes and methods, which can reduce the repeated code (**Don't Repeat Yourself**)

		Moreover, we add this weapon into trader's 'sellItem' list and 'acceptedWeapon' list such that this Weapon item can be purchased by player and sold to trader.
Club extends WeaponItem	A class that represents a weapon called Club Dependency: - **Trader:** Club will be added into sellItem and acceptedWeapon list	This class is created because it represents a weaponItem that is not existed in the original system. We inherited WeaponItem class because they share the common attributes and methods, which can reduce the repeated code (**Don't Repeat Yourself**) Moreover, we add this weapon into trader's 'sellItem' list and 'acceptedWeapon' list such that this Weapon item can be purchased by player and sold to trader.
Grossmesser extends WeaponItem	A class that represents a weapon called Grossmesser, which is carried around by Heavy Skeletal Swordsman. Dependency: - **Trader:** Grossmesser will be added into acceptedWeapon list	Moreover, we add this weapon into trader's 'acceptedWeapon' list such that this Weapon item can be sold to trader (but this weapon cannot be purchased).

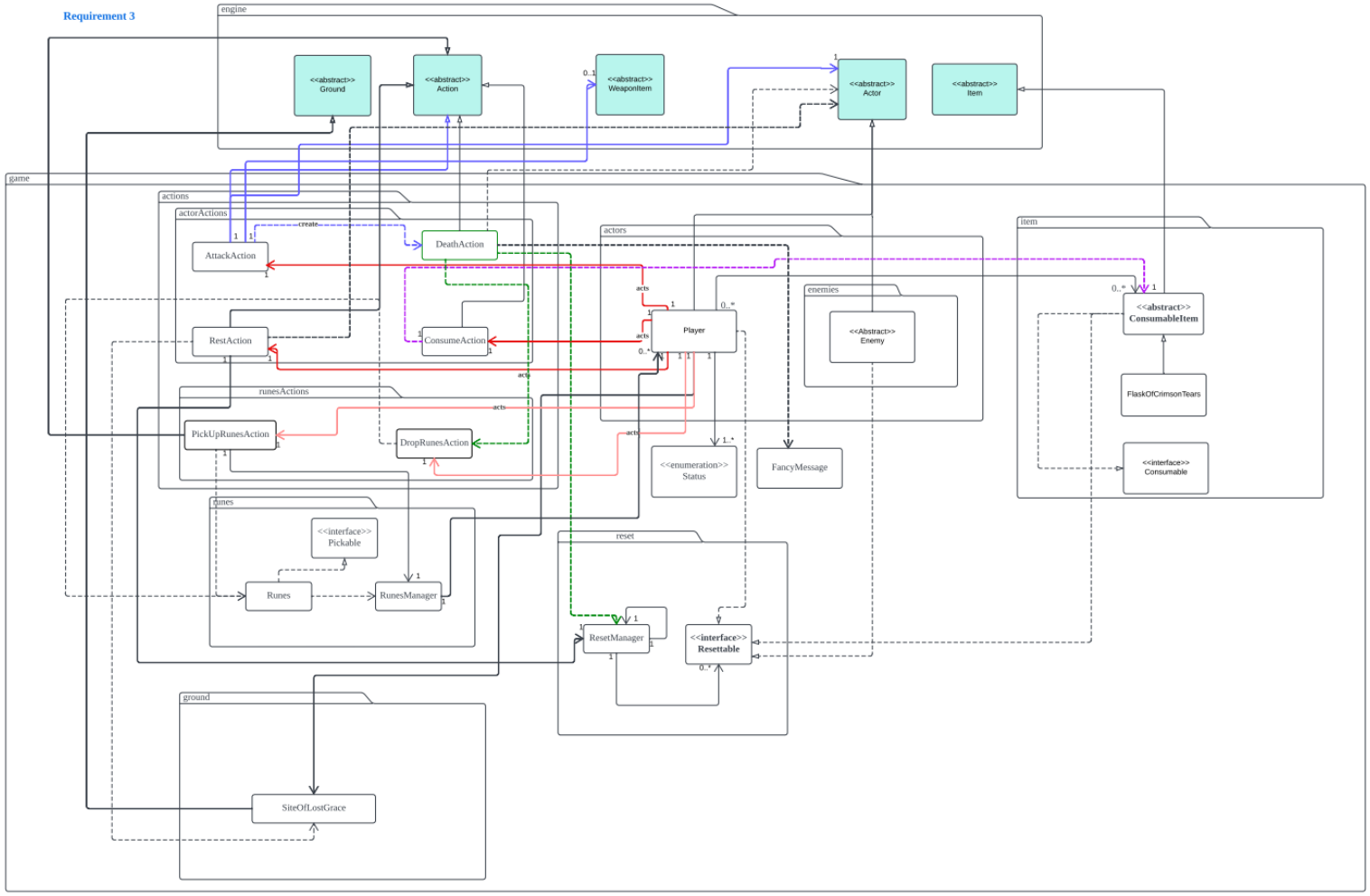
Package: game.runes

New classes	Class Responsibility	Design Rationale
RunesManager	A class that manages the Runes Dependency: - **Player**: The role of RunesManager is to manage the player's runesValue	To avoid having God class of Runes, we created a RunesManager to manage the Runes (**Single Responsibility Principle**)

	(runesValue is like a wallet attribute inside player class) - **SellAction**: SellAction called RunesManager to add player's runes value. - **PurchaseAction**: PurchaseAction called RunesManager to substract player's runes value. - **TransferRunesAction**: TransferRunesAction called RunesManger to add the runes value that enemies drop into player's runes value	Since the methods inside RunesManager don't require any object state manipulation, we design our methods and attributes as static so we can **reuse standard behaviour across instances of different classes.**

Req 3

UML Diagram



A. Flask of Crimson Tears

Package: game.items

New classes	Class Responsibility	Design Rationale
Consumable (Interface)	A class that represents the interface of Consumable, contains methods for consumable items. Dependency: - **ConsumableItem**: ConsumableItem will implement this interface to implement the Consumable methods.	Consumable has the method of Consume, all the classes that implement this interface should also have the method being implemented in it, which suits the design principle of **Don't Repeat Yourself,** making the design more extensible and reusable.
ConsumableItem (abstract) extends Item implements Consumable	A class that represents an item that can be consumed by player. Dependency: - **Consumable**: Implements methods from Consumable for ConsumableItem	We have implemented an abstract class called ConsumableItem that extends from Item class, which will include all the attributes for a consumable item. This implementation can achieve the **Open-Closed Principle**, and this makes introducing new item that is consumable easier. We have our ConsumableItem class implementing the Consumable interface so that all items under the ConsumableItem can be consumed with the method 'consume' being implemented. This design follows the **Single Responsibility Principle**, preventing the presence of God Class in our design.
FlaskOfCrimsonTears extends ConsumableItem	A class that represents a type of item that is of ConsumableItem.	The FlaskOfCrimsonTears class inherits the ConsumableItem abstract class. From this

		inheritance, we reduce repetition of codes, and so makes the program more efficient and easier to extend. The common attributes are stated in the abstract class. This makes it open for extension and close for modification, which fulfills the **Open-Closed Principle**.

Package: game.actions.actorAction

New classes	Class Responsibility	Design Rationale
ConsumeAction extends Action	A class that is used to represent the action of consuming item, for player. Dependency: - **Player**: Player can choose to consume the ConsumableItem. - **ConsumableItem**: The items that are being consumed when ConsumeAction is performed by the Player.	ConsumeAction class is created to give the player an action to Consume a ConsumableItem. We follow the **Single Responsibility Principle** when creating the ConsumeAction class so that God Class is avoided, as the ConsumeAction only allows player to consume an item. Similar to other action class, the implementation of ConsumeAction is also based on the concept of **Polymorphism**, which allows different type of actions store into ActionList that can be use in various ways

B. Site of Lost Grace

Package: game.ground

New classes	Class Responsibility	Design Rationale

SiteOfLostGrace extends Ground	A class that represents a type of Ground, where the player can rest at this place. Dependency: - **Player**: Player can only rest when they are at the Site of Lost Grace.	The SiteOfLostGrace class inherits the Ground abstract class. From this inheritance, we reduce the repetition of codes, and so makes the program more efficient and easier to extend. The common attributes are all stated in the abstract class. And from this inheritance, it follows the **Open-Closed Principle** since it is open for extension by adding new classes inheriting the Ground and close for modification since no modification is needed for the Ground abstract class.

Package: game.actions.actorActions

New classes	Class Responsibility	Design Rationale
RestAction extends Action	A class that is used to represent the action of resting, for player. Dependency: - **SiteOfLostGrace**: Player can only rest when they are at SiteOfLostGrace. - **Player**: Player can choose to 'rest' at the SiteOfLostGrace.	RestAction class is created to give the player an action of Rest when they are at the SiteOfLostGrace. We follow the **Single Responsibility Principle** when creating the RestAction class so that God Class is avoided, the RestAction only allows player to rest. Similar to other action class, the implementation of RestAction is also based on the concept of **Polymorphism**, which allows different type of actions store into ActionList that can be use in various ways

C. Game Reset

Package: game.reset

New classes	Class Responsibility	Design Rationale
Resettable (interface)	A class that represents an interface of Resettable, contains methods for factors that can be reset. Dependency: - **Enemy**: All enemies should be despawned from the ground when reset is executed - **Player**: Player's attributes should be reset to default when reset is executed, depends on whether it is executed when player is resting or is dead. - **FlaskOfCrimsonTears**: The number of times this item can be consumed will be set back to default during reset.	**Resettable** will have a method of reset, and every class that implements the Resettable interface should implement this method too, which suits the design principle of **Don't Repeat Yourself**, making the design more extensible and reusable.
ResetManager implement Resettable	A class that is used to manage all the items that can be reset, contains list of resettable items. Dependency: - **RestAction**: When the player rests at the Site of Lost Grace, reset is called to set everything to default except for runes.	To avoid having a God Class, we implemented a ResetManager to manage all the items that and attraibutes that can be reset in the game (**Single Responsibility Principle**). This class implements the Resettable interface to reduce repetition of codes and gives a flexibility on modifying the game. It will be easier to make extensions. (**Don't Repeat Yourself**)

	- **DeathAction**: When the player dies, everything in the World will be reset.	

Package: game.actions.actorActions

New classes	Class Responsibility	Design Rationale
DeathAction extends Action	A class that represents the action when an actor is dead. Dependency: - **AttackAction**: when attack is executed, check if target is conscious. If not, then DeathAction is executed. - **Actor**: The death action is applied on an actor. - **DropRunesAction**: When player dies, runes will drop on the ground. - **FancyMessage**: A fancy message of "YOU DIED" is displayed when the player dies.	An extra dependency is added to DropRunesAction class so that when DeathAction is executed on Player, DropRunesAction is executed too. A DropRunesAction class was created so that it reduced repetition of codes following the **Don't Repeat Yourself Principle.** FancyMessage class is depended on, and certain messages will be displayed according to the current action.

D. Runes

Package: game.runes

New classes	Class Responsibility	Design Rationale
Pickable (interface)	A class that represents an interface of Pickable, which has the method of pick, for items or objects that can be picked up.	**Pickable** will have a method of 'pick', every class that implements the Pickable interface should also implement this method. With the implementation of this interface, we avoid

		repetition of codes, and this fulfills the **Don't Repeat Yourself** principle.
Runes implement Pickable	A class that represents the Runes object, which is the currency in the game. Dependency: - **Pickable (interface)**: When implementing the pickable interface, the 'pick' method will be implemented too.	**Runes** simply represent the Runes object throughout the game and are used within the game for trading. Runes can be earned from killing enemies. This object will be dropped when the player dies and can be picked up when the player goes near to runes on the ground, therefore, it implements the interface of pickable.

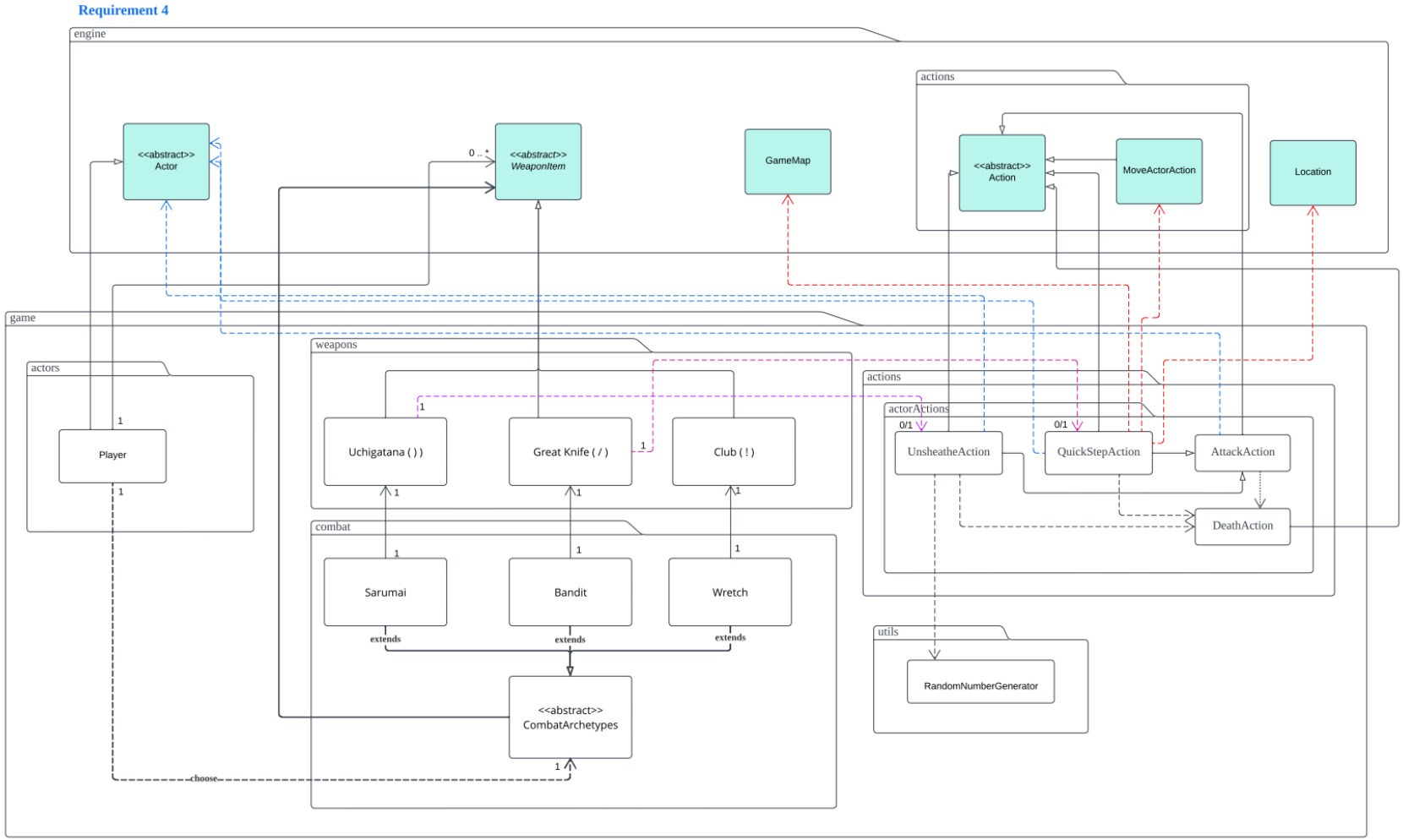
Package: game.actions.runesActions

DropRunesAction extends Action	A class that is used to represent the action of dropping runes, when player dies, or when enemies die attacked by player. Dependency: - **DeathAction**: When player is dead, DropRunesAction will be created and executed. - **Player**: Player will drop runes when they died. - **Runes**: Runes object that is dropped onto the ground.	DropRunesAction class inherits from Action, and has a dependency on DeathAction, where when Player dies, Player will drop runes on the ground before reviving at the Site of Lost Grace. This follows the **Single Responsibility Principle** as DropRunesAction is only used to drop runes, avoiding creation of God Class in our implementation.
PickUpRunesAction extends Action	A class that is used to represent the action of picking up runes from the ground. Dependency:	PickUpRunesAction class inherits from Action, giving the Player an action to pick up runes from the ground. This implementation follows the **Single Responsibility Principle** as this action is used to pick up runes only,

	- **Player**: Player can choose to pick up the runes that is on the ground when they are near to the runes. - **RunesManager**: The 'add' method in the RunesManager is used when player pick up runes. The runes will be added to the player's current runes and be updated. - **Runes:** The player can pick up the runes object that is on the ground which is of Runes class. The value is stated.	avoiding the creation of a God Class in our implementation. Additionally, this action has a dependency to RunesManager, where it utilizes RunesManager to manage the addition of Runes to the Player, avoiding repetition of codes, following the **Don't Repeat Yourself** Principle.

Req 4

UML Diagram



A. Classes/Combat Archetypes

Package: game.combat

New classes	Class Responsibility	Design Rationale
CombatArchetypes (abstract)	An abstract class which acts as the base class of the three different modes in the game. Dependency: - **WeaponItem**: the starting weapon for each mode	An abstract class, CombatArchetypes, was implemented, to allow the concrete classes, Sarumai, Bandit and Wretch to extend from, adhering to the **Dependency Inversion Principle.** Additionally, the CombatArchetypes class is an abstract class because we knew that we would not instantiate the CombatArchetypes class in any of our code. Such implementation helped us to achieve abstraction.
Sarumai extends CombatArchetypes	This class represents one of the starting classes/modes in the game. Dependency: - **WeaponItem**: the starting weapon	Sarumai class is used to represent one of the modes in the game and therefore inherit CombatArchetypes. In Sarumai class, it will have its own starting weapon Uchigatana and its own starting hit point. So that this design adheres to **Single responsibility principle**.
Bandit extends CombatArchetypes	This class represents one of the starting classes/modes in the game. Dependency: - **WeaponItem**: the starting weapon	Banditclass is used to represent one of the modes in the game and therefore inherit CombatArchetypes. In Bandit class, it will have its own starting weapon Great Knife and its own starting hit point. So that this design adheres to **Single responsibility principle**.
Wretch extends CombatArchetypes	This class represents one of the starting classes/modes in the game.	Wretch class is used to represent one of the modes in the game and therefore inherit

	Dependency: - **WeaponItem**: the starting weapon	CombatArchetypes and its own starting hit point. So that this design adheres to **Single responsibility principle**.

B. <u>Weapons</u>

Package: game.weapons

New classes	Class Responsibility	Design Rationale
Uchigatana extends WeaponItem	A class that represents a weapon called Uchigatana. Starting weapon of Samurai class.	This class was created because it represents a weaponItem that does not exist in the original system. We inherited WeaponItem class because they share the common attributes and methods, which can reduce the repeated code (**Don't Repeat Yourself**) Moreover, Uchigatana allows the user to perform Unsheathe special action.
GreatKnife extends WeaponItem	A class that represents a weapon called Great Knife. Starting weapon of Bandit class.	This class was created because it represents a weaponItem that does not exist in the original system. Similar to Uchigatana class, we inherited WeaponItem class since they share the common attributes and methods, which can reduce the repeated code (**Don't Repeat Yourself**) Besides, GreatKnife allows the user to perform QuickStep special action.
Club extends WeaponItem	A class that represents a weapon called Club. Starting weapon of Wretch class.	This class was created because it represents a weaponItem that does not exist in the

		original system. We inherited WeaponItem class because they share the common attributes and methods, which can reduce the repeated code (**Don't Repeat Yourself**)

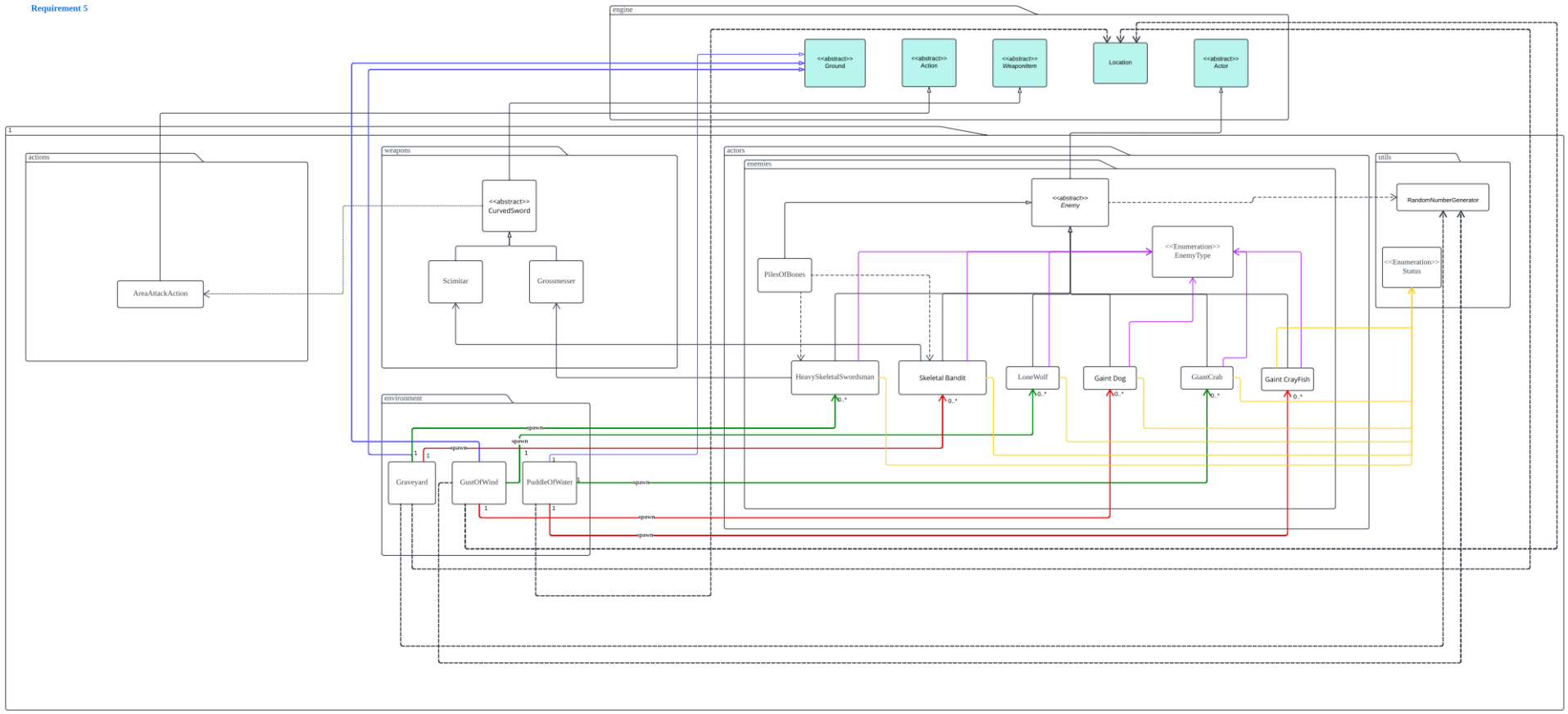
Package: game.actions.actorActions

New classes	Class Responsibility	Design Rationale
UnsheatheAction extends AttackAction	A unique attack action class that represents UnsheatheAction Dependency: - **Uchigatana**: UnsheatheAction is a special skill that an Uchigatana can have. When user execute getSkill() method will return this action. - **RandomNumberGenerator**: User have 60% chance to hit the enemy. - **Actor**: The user and target are actors. - **DeathAction**: If the target is not conscious then execute DeathAction	To achieve **Single Responsibility Principle**, we create an action class that perform 'Unsheathe' if the user holds Uchigatana. Besides, we design UnsheatheAction inherit AttackAction, which reduce the repetition of codes, and so makes the program more efficient and easier to extend. **(Don't Repeat Yourself** and **Open-Closed Principle**)
QuickStepAction extends AttackAction	A unique attack action class that represents QuickStepAction Dependency: - **GreatKnife**: QuickStepAction is a special skill that a GreatKnife can have. When user execute getSkill() method will return this action.	To achieve **Single Responsibility Principle**, we create an action class that perform 'QuickStep' if the user holds GreatKnife. Besides, we design UnsheatheAction inherit AttackAction, which reduce the repetition of codes, and so makes the program more efficient and easier to extend. **(Don't Repeat Yourself** and **Open-Closed Principle**)

	- **AttackAction**: QuickStepAction inherit AttackAction - **MoveActorAction**: User perform move action after performing attack action. - **GameMap:** the current map - **Location**: The location that user moves after attacking. - **DeathAction**: If the target is not conscious then execute DeathAction	

Req 5

UML Diagram



Scenario

Package: game.ground.environment

New classes	Class Responsibility	Design Rationale
Graveyard extends Ground	A class that represents Graveyard which is occupied by SkeletalTypeEnemy creatures. Dependency: - **Ground**: Graveyard inherit ground class - **Location**: Graveyard spawn the different type of SkeletalTypeEnemies by checking the current location - **RandomNumberGenerator:** The chance that an enemy spawn	We modify the existing method (tick (Location location)) inside **Graveyard** to spawn different enemy based on their map location. Therefore, on the **West side** of the map, we spawn **Heavy Skeletal Swordsman** and spawn **Skeletal Bandit** on the **East side** of the map.
GustOfWind extends Ground	A class that represents GustOfWind which is occupied by DogTypeEnemy creatures. Dependency: - **Ground**: GustOfWind inherit ground class - **Location**: GustOfWind spawn different type of DogTypeEnemy by checking the current location - **RandomNumberGenerator:** The chance that an enemy spawn	Same as the concept of Graveyard, we modify the existing method (tick (Location location)) inside **Gust of Wind** based on their map location. Hence, **Lone Wolf** is spawned in **West side** and **Giant dog** is spawned in **East side**.

PuddleOfWater extends Ground	A class that represents PuddleOfWater which is occupied by WaterTypeEnemy creatures. Dependency: - **Ground**: PuddleOfWater inherit ground class - **Location**: PuddleOfWater spawn different type of WaterTypeEnemy by checking the current location - **RandomNumberGenerator:** The chance that an enemy spawn	We modify the existing method (tick (Location location)) inside **Puddle of Water** to spawn different enemy based on their map location. Therefore, on the **West side** of the map, we spawn **Giant crab** and spawn **Giant Crabfish** on the **East side** of the map.

A. <u>Enemies</u>

Package: game.enemies

New classes	Class Responsibility	Design Rationale
HeavySkeletalSwordsman extends Enemy	A class represents Heavy Skeletal Swordsman Dependency: - Enemy: HeavySkeletalSwordsman inherit Enemies class - **Status**: Add 'HOSTILE_TO_DOGTYPE' and 'HOSTILE_TO_WATERTYPE' into capabilities - **EnemyType:** SkeletalTypeEnemy	- Add 'HOSTILE_TO_DOGTYPE' and 'HOSTILE_TO_WATERTYPE' into the status such that DogTypeEnemy and WaterTypeEnemy can attack HeavySkeletalSwordsman

	- **Graveyard**: The Graveyard will spawn this type of enemy if the location is on the West Side of the map.	
LoneWolf extends Enemy	A class represents Lone Wolf Dependency: - **Status**: Add 'HOSTILE_TO_SKELETALTYPE' and 'HOSTILE_TO_WATERTYPE' into capabilities - **EnemyType:** DogTypeEnemy - **GustOfWind**: The GustOfWind will spawn this type of enemy if the location is on the West Side of the map.	- Add 'HOSTILE_TO_SKELETALTYPE' and 'HOSTILE_TO_WATERTYPE' into the status such that SkeletalTypeEnemy and WaterTypeEnemy can attack LoneWolf
GiantCrab extends Enemy	A class represents Giant Crab Dependency: - **Status**: Add 'HOSTILE_TO_SKELETALTYPE' and 'HOSTILE_TO_DOGTYPE' into capabilities - **EnemyType:** WaterTypeEnemy - **PustOfWater**: The PustOfWater will spawn this type of enemy if the location is on the West Side of the map.	- Add 'HOSTILE_TO_SKELETALTYPE' and 'HOSTILE_TO_DOGTYPE' into the status such that SkeletalTypeEnemy and WaterTypeEnemy can attack **GiantCrab**
SkeletalBandit extends Enemy	A class represents Skeletal Bandit Dependency:	- Add 'HOSTILE_TO_DOGTYPE' and 'HOSTILE_TO_FISHTYPE' into the status such that DogTypeEnemy and

	- **Status**: Add 'HOSTILE_TO_DOGTYPE' and 'HOSTILE_TO_WATERTYPE' into capabilities - **EnemyType:** SkeletalTypeEnemy - **Graveyard**: The Graveyard will spawn this type of enemy if the location is on the East Side of the map.	WaterTypeEnemy can attack **SkeletalBandit**
GiantDog extends Enemy	A class represents Giant Dog Dependency: - **Status**: Add 'HOSTILE_TO_SKELETALTYPE' and 'HOSTILE_TO_WATERTYPE' into capabilities - **EnemyType:** DogTypeEnemy - **GustOfWind**: The GustOfWind will spawn this type of enemy if the location is on the East Side of the map.	- Add 'HOSTILE_TO_SKELETALTYPE' and 'HOSTILE_TO_WATERTYPE' into the status such that SkeletalTypeEnemy and WaterTypeEnemy can attack **GiantDog**
GiantCrayfish extends Enemy	A class represents Giant Cray Fish Dependency: - **Status**: Add 'HOSTILE_TO_SKELETALTYPE' and 'HOSTILE_TO_DOGTYPE' into capabilities - **EnemyType:** WaterTypeEnemy - **PustOfWater**: The PustOfWater will spawn this type of enemy if the	- Add 'HOSTILE_TO_SKELETALTYPE' and 'HOSTILE_TO_DOGTYPE' into the status such that SkeletalTypeEnemy and WaterTypeEnemy can attack **GiantCrayfish**

	location is on the West Side of the map.	
PileOfBones extends Enemy	A class represents Piles of Bones Dependency: - **Status**: Add 'HOSTILE_TO_DOGTYPE' and 'HOSTILE_TO_WATERTYPE' into capabilities - **EnemyType:** SkeletalTypeEnemy - **SkeletalTypeEnemy:** SkeletalTypeEnemy will turn into PilesOfBones if killed	- Add 'HOSTILE_TO_DOGTYPE' and 'HOSTILE_TO_WATERTYPE' into the status such that SkeletalTypeEnemy and WaterTypeEnemy can attack **PileOfBones**
EnemyType (Enumeration)	The enum defines enemy type	Three types of Enemies: - SkeletalTypeEnemy - DogTypeEnemy - WaterTypeEnemy The use of enumration will avoid the excessive use of literals, and therefore will improve maintainability and extensibility of the code in the long-term.

Package: game.utils

New classes	Class Responsibility	Design Rationale
Status (Enumeration)	The enum class to give `buff` or `debuff`	Add '**HOSTILE_TO_SKELETALTYPE**' Add '**HOSTILE_TO_DOGTYPE**' Add '**HOSTILE_TO_WATERTYPE**'

		To prevent violating **Open-Closed Principle**, we create corresponding status so makes the program more efficient and easier to extend in the future.

B. <u>Weapons</u>

Package: game.weapons

New classes	Class Responsibility	Design Rationale
CurvedSword (abstract) extends WeaponItem	An abstract class which acts as the base class of Grossmesser and Scimitar Dependency: - **WeaponItem**: CurvedSword inherit WeaponItem class. - **AreaAttackAction**: A special skill that a CurvedSword can perform.	To achieve **Open-Closed Principle**, we create an abstract class that include all the characteristics of CurvedSword weapon item, allowing for easier addition of new enemy classes in the future. We also inherited curved sword from weapon item class so we can reuse the existing method from parents class to **reduce the repeating code**. Also, we knew that we would not instantiate the curvedSword class in any of our code. Such implementation helped us to achieve **abstraction**.
Grossmesser extends CurvedSword	A class that represents a weapon called Grossmesser, which is carried around by Heavy Skeletal Swordsman. Dependency: - **CurcedSword**: Grossmesser inherit CurvedSword class.	Initially, Grossmesser inherited from weapon item but now we make this as the subclass of CurvedSword because they share the similar characteristic with Scimitar class.

Scimitar extends CurvedSword	A class that represents a weapon called Scimitar, which is carried around by Skeletal Bandit. Dependency: - **CurcedSword**: Scimitar inherit CurvedSword class.	Since Scimitar share the similar characteristic with Grossmesser class, we inherit the scimitar class from curvedSword class, this achieves the principle of **Don't Repeat Yourself**.

Summary – pros and cons

Overall, our design rationale follows the SOLID principle, aiming to reduce code repetition and maintainability. For instance, we replaced the use of 'instanceOf' with a status enumeration class to fulfil the Open-Closed principle. While we have made significant progress, there is still a room of improvement in our code. In our design rationale, we have identified areas where our design violates SOLID principles, such as the handling of actor death actions. While we have tried to avoid violating the Single Responsibility Principle, we have had to design the death action to manage multiple responsibilities since we cannot modify the actor class. Therefore, we acknowledge that there are opportunities to refine our design further in the future.