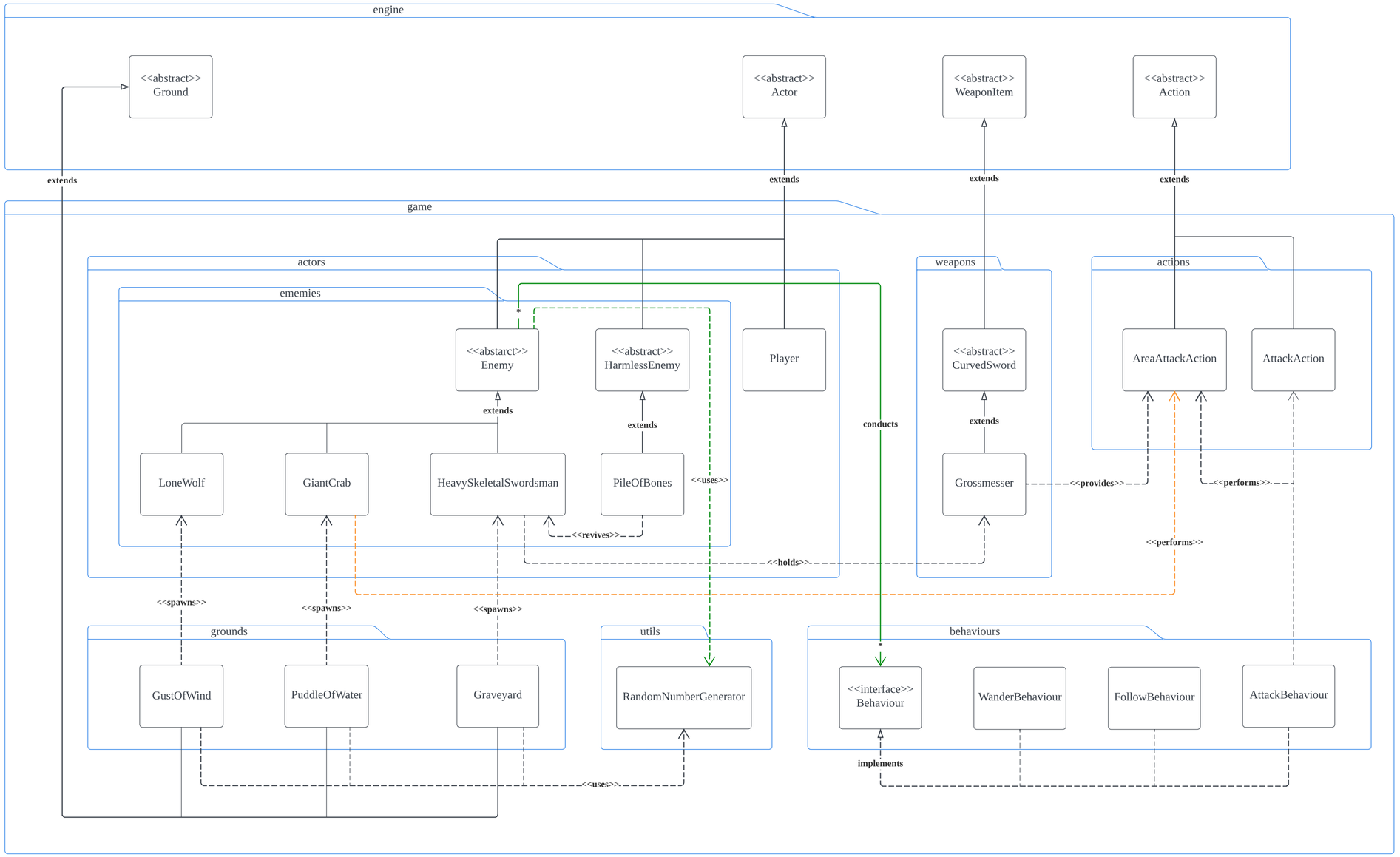
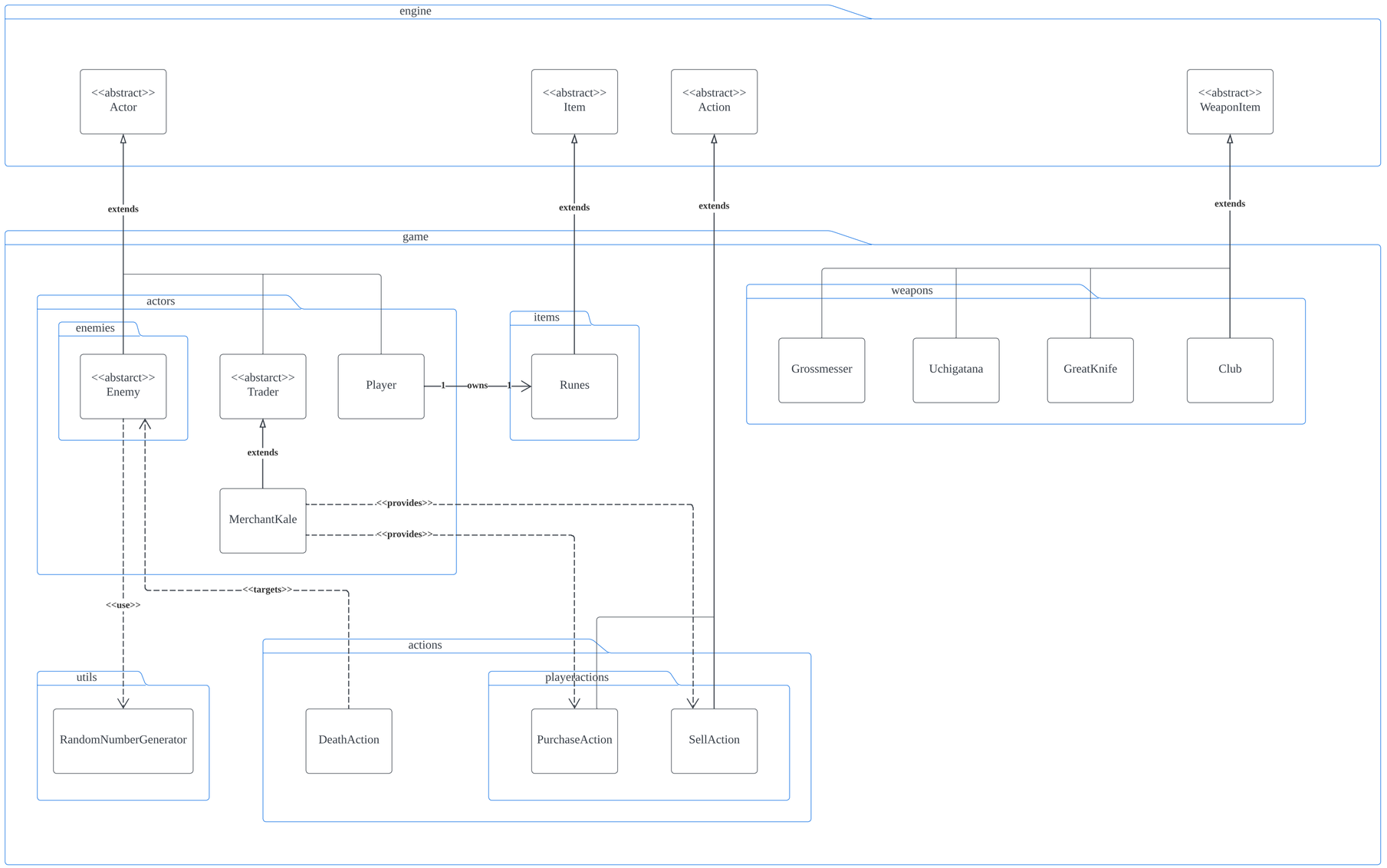
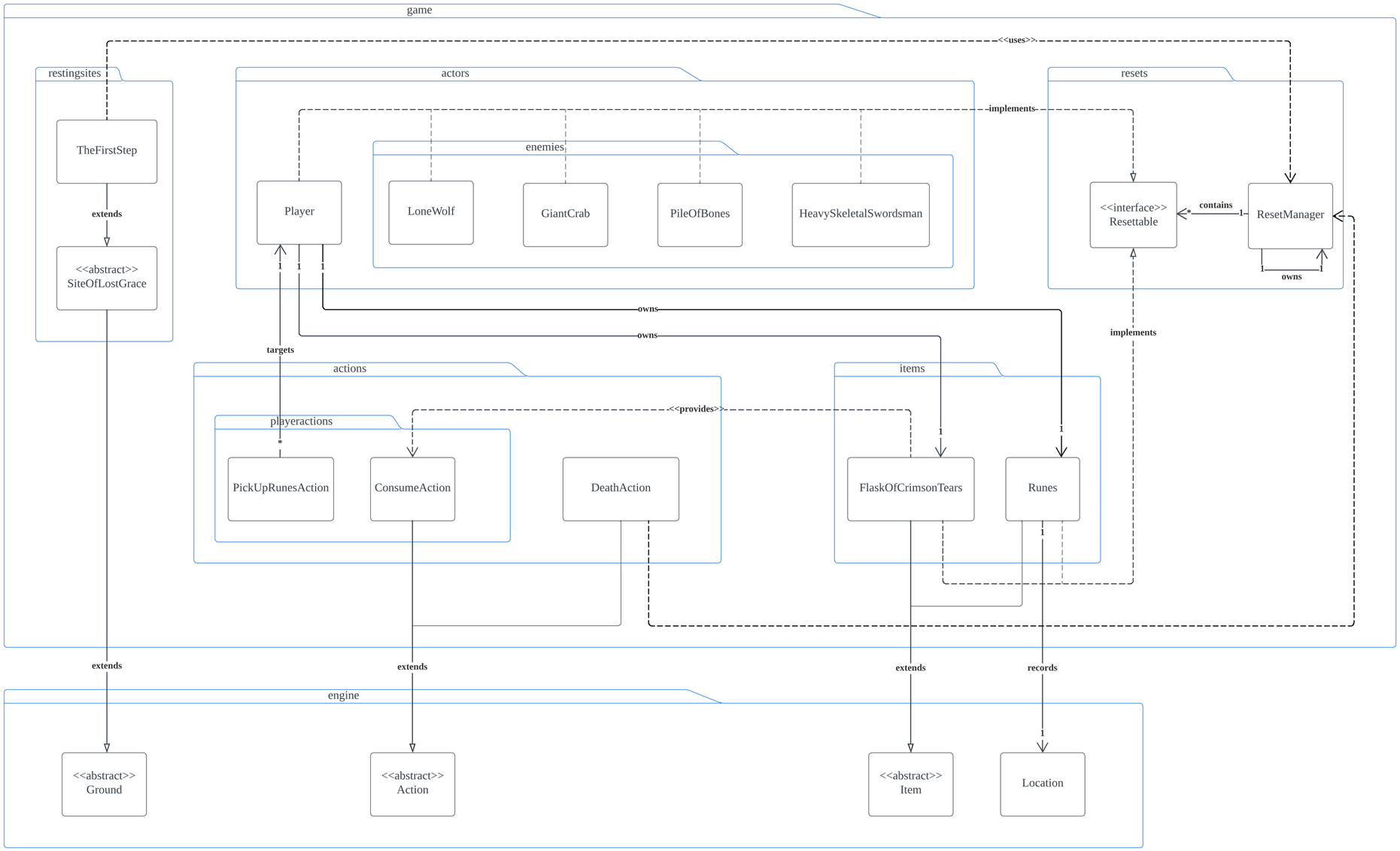
Requirement 1



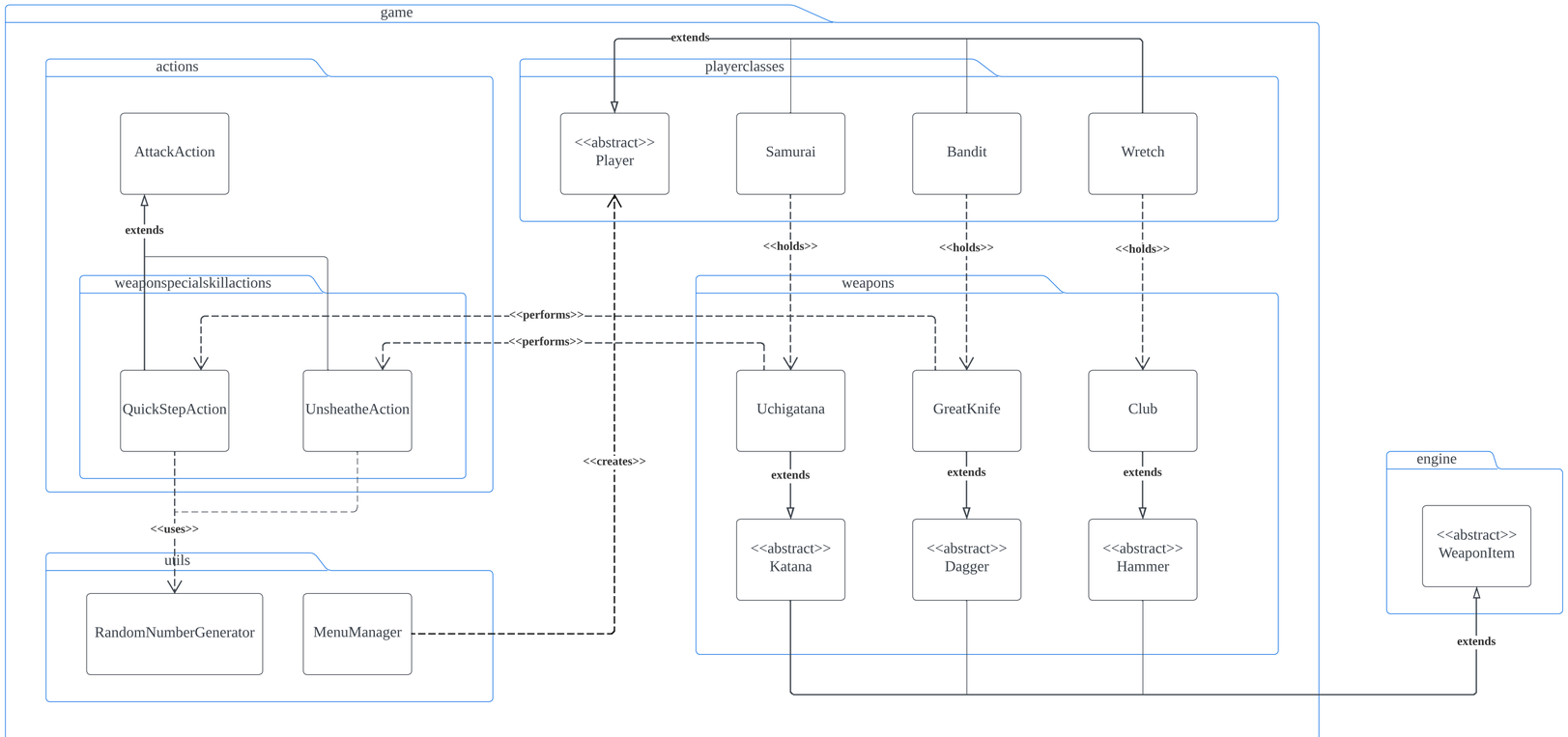
Requirement 2



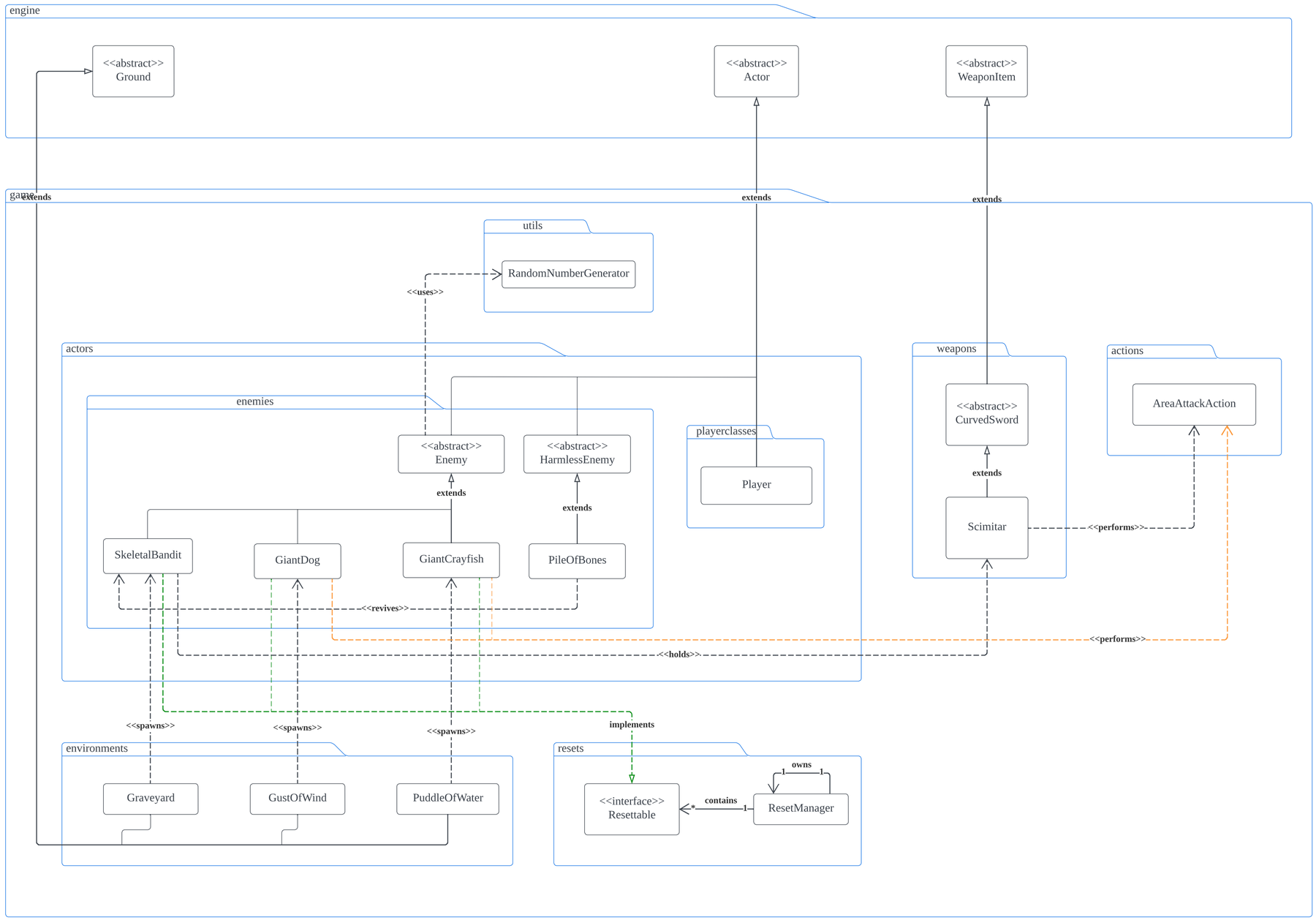
Requirement 3



Requirement 4



Requirement 5



DESIGN RATIONALE

Requirement 1

The UML diagram above represents an object-oriented system for Game Requirement 1 in which there are three types of environments where each has a different chance to spawn its respective enemy at every game turn. The enemy can only attack another type of enemy unless an area attack is performed. Besides, the enemy will start to follow the player once the player comes closer to the enemy, else they will just wander around the map. In addition, there is a unique enemy known as "Pile Of Bones," which appears only when a Heavy Skeletal Swordsman is defeated and would not cause any harm to the player. If the Pile Of Bones is not defeated within three turns, the Heavy Skeletal Swordsman will be revived with full health. Besides, “Grossmesser” a weapon that is carried around by a Heavy Skeletal Swordsman can perform a single attack or spinning attack (area attack) .

To fulfil Requirement 1 part A, three new classes called “Graveyard”, “GustOfWind” and “PuddleOfWater” are created where each is the environment for their respective enemy. Each environment class has a different chance of spawning its corresponding enemy, therefore each environment class can use the “RandomNumberGenerator” to decide when to spawn their corresponding enemies on the location, based on specific spawning chance provided. To follow the DRY (Don't Repeat Yourself) principle, the classes for Graveyard, Gust of Wind, and Puddle of Water inherit the “Ground” abstract class to prevent repeating code as the “Ground” class defines all the attributes and methods that are common to all types of environments. Besides, this approach also follows the Single Responsibility Principle (SRP) as each subclass is responsible for defining its unique chance of spawning and instantiating its corresponding enemy.

In requirement 1 part B, we create three new classes for hostile creatures: “HeavySkeletalSwordsman”,”LoneWolf”, and “GiantCrab”. All three classes inherit from a parent abstract class called “Enemy”**,** which itself is derived from the engine code’s abstract class “Actor”. This approach follows the DRY principle which avoids repetition of code since all the enemy classes share common attributes and methods. Rather than defining these attributes and methods in each enemy class, they are defined once in the "Enemy" abstract class and inherited by all its subclasses. To elaborate, instead of coding all similar implementations in every different enemy class, we can just code a generic one in the “Enemy” parent class and then call *super()* in the subclasses and add some unique functionality easily. Besides, creating an “Enemy” abstract class can also reduce the dependency on the actor class (between specific enemy classes with the “Actor” abstract class”), as well as separating the functionalities of an NPC from that of players.

Alternative 1 (initial thought): Since no attack/hitpoints & attack accuracy, so we treat it as ground

Limitations:

1. Ground has no name attribute - cannot set it “PileOfBones”

2. Pile of bones logically must have same precedence level as typical enemies
3. Hard to keep track of the number of rounds during the game (when to respawn into HSS and when to remove itself from map - once attacked by player) - is possible but will probably introduce more dependency

Alternative 2: subclass of enemy

Limitations:

1. E.g. No hitpoints & accuracy & despawn chance & cannot drop runes once died & supposingly no inventory etc.
2. Have certain differences from typical enemy
3. No harm to players as well

Alternative 3 (decision): abstract class “Harmless Enemy”

Advantages:

1. Differentiate between typical enemies and zero-attack enemy - e.g. cause no harm to players
2. Since extends from “Actor” so have playTurn method - can keep track of the number of rounds here more easily and reduce certain dependency
3. Same precedence level as enemy
4. Unique constructor that have a little difference with “Enemy” - no parameters to initialise instance vars like despawn chance, hitpoints, accuracy etc.
5. More robust and extensible in future (and maybe relates abit to OCP) - if more types of harmless enemies introduced to game, then easy to implement by just extending this abstract class

In our initial design, we treat Pile Of Bones as a ground, since they have no hit points, no item and weapon inventory, no attack accuracy and no despawn chance. However, there are a few limitations in this approach. Since player can perform attack action to pile of bones and they will remove and revives Heavy Skeletal Swordsman after being hit within three turns, logically they must have same precedence level as typical enemies because player can only attack an enemy and after a ground is added it cannot be removed. Besides, Ground has the lowest display level in this game, in the scenario where there is an actor or item in the same location as pile of bones, pile of bones will not be display if it is treated as a “Ground”. Also, if the pile of bones is implemented as a type of ground, it would also be difficult to keep track the number of rounds that the pile of bones has taken, revive the Heavy Skeletal Swordsman and remove itself from the map after being hit, more dependencies will be introduce for the implementation. Hence, “PileOfBones” should not be implemented as a type of “Ground”.

This led to a second alternative where we treat “PileOfBones” as an actor and is a subclass of “Enemy” abstract class. However, Pile of Bones does not have the ability to attack, follow or wander which distinguishes it from typical enemies and can be considered harmless. The

limitations of this alternative is that we have to manually initialise the irrelevant instance variables to the Pile of Bones such as despawn chance and attack accuracy via calling the *super()* in the constructor which we used to inherit the other harmful enemies like HeavySkeletalSwordsman, LoneWolf and GiantCrab. This makes the system not robust because theoretically based on specification, “PileOfBones” is not having any of the attributes mentioned above as they will not despawn by chance and they are harmless to the others where no attack will be executed by them.

Therefore, this led to our third approach where in order to differentiate between typical enemies and harmless enemy we create an abstract class called “HarmlessEnemy” which has the same level of inheritance as “Enemy” abstract class. This approach avoids turning the "Enemy" abstract class into a "god class" that knows all aspects of harmful and harmless enemies, which would go against the SRP principle and by overriding the *playTurn* method from “Actor”, the number of rounds that Pile of Bones has taken can be kept track easily without introducing unnecessary dependency. Moreover, this approach is also more robust and extensible in future because if there are new types of harmless enemies introduced to the game, only a new subclass need to be created without making any changes to the “HarmlessEnemy” parent class, this follow OCP principle.

To achieve this requirement, our team decided to add capability of *RESPAWNABLE* (enumeration) to every Heavy Skeletal Swordsman, and then in the “DeathAction” class’s *execute* method, we would check whether or not the currently dying enemy has the *RESPWANABLE* capability. If the statement returns true, then the enemy is a Heavy Skeletal Swordsman and therefore we can directly remove the dying enemy and create an instance of “PileOfBones” on that location.

There is another alternative for this requirement which is creating a new interface “Respawnable” and to be implemented by “HeavySkeletalSwordsman”. However, the only limitation of this alternative is that, there is no any required promise method to be implemented by the respawnable class, in which we will not actually make full use of a typical Java’s interface’s main objective. Also, this alternative will result in more unnecessary dependency relationships between classes, making the system much more complicated. Therefore, we chose to utilize Java's enumeration for this requirement out of the two valid alternatives. The only tradeoff of using enumeration is that we will have to manually add *RESPAWNABLE* capability to every enemy that has this unique ability.

Furthermore, instead of performing a single attack, Giant Crab can decide to attack (slam) all creatures within their surroundings, so a class called “AreaAttackAction” is created that takes on the responsibility of handling any area attacks which follows the SRP principle (separating responsibility of normal attack to this unique (area) attack) where each class has a clear and distinct responsibility, and only one reason to change.

Here can also touch a bit on the main reason why we create this “AreaAA” class - because player will be given option to perform this action via console (if they hold this weapon) - so to make the code consistent we decided to make a new class for every user selectable action (as well as in the following Game requirements) - better clarity - e.g. if player has this weapon held, we add this areaattackaction to allowabbleactions method instead of attackaction - easy to understand & debug & trace errors

Heavy Skeletal Bandit carries around a weapon called Grossmesser, so a new class “Grossmesser” is created which extends “CurvedSword” abstract class which itself is a subclass

of the “WeaponItem" abstract class in the engine's code. This inheritance structure is designed to account for other weapons that belong to the curved sword type. By having this approach, if another curved sword type weapon is added in the future, it can be easily maintained and implemented by creating a new subclass of the “CurvedSword” abstract class, making the system more extensible in future. This follows the Open-Close Principle (OCP) and Single Responsibility Principle (SRP) of the SOLID principles by allowing for easy extension of the code without modifying existing parent classes and ensuring that each subclass has a single responsibility (or their own special functionalities without letting the “CurvedSword” class to handle/manage their uniqueness). Not only that, if more weapons of type “CurvedSword” are added into the game in future, repetition of code can be avoided as calling *super()* will help subclasses to inherit the commonalities from parent class. Since Grossmesser can allow the user and Heavy Skeletal Swordsman to perform a single attack or spinning attack thus, it also has a dependency to “AreaAttackAction” and “AttackAction” class. (... user including HSS to perform spinning attack instead of normal single attack, therefore in the “Grossmesser” class we can directly add an instance of “AreaAttackAction” into the list of user’s executable actions),

Also, since the “GiantCrab” has chance to execute slam area attack, therefore in its *playTurn* method we can simply add an instance of “AreaAttackAction” into the list of executable actions of “GiantCrab”. For example, we will be using “RandomNumberGenerator” to decide when the “GiantCrab” will perform “AreaAttackAction” instead of normal attack. Here our team assumed the probability of slam area attack being executed by “GiantCrab” is 50% since it is not mentioned in the assignment specification.

Here our group has decided to treat the spinning attack and slam area attack the same by just providing “AreaAttackAction” for both cases, as from the specification we realised that there is apparently no difference between these two attacks, which is just hitting anything in the surroundings with the actor’s original damage and attack accuracy.

To achieve the precedence of different behaviours among enemies, our team decided to utilise Java's data structure which is HashMap to implement this requirement. Every enemy will have their own instance variable called *behaviours* with HashMap as its variable type. Since every enemy will wander around the map by default, therefore our group planned to just add an instance of “WanderBehaviour” into their *behaviours* in the “Enemy” class’s constructor, by putting a large key value (as this behaviour has the lowest precedence among the three). Once the player comes closer to an enemy, we will then add an instance of “FollowBehaviour” by putting a key value smaller than that of “WanderBehaviour” into the *behaviours* of this particular enemy (as this behaviour has the intermediate precedence). Lastly, when an enemy can perform an attack, we will add an instance of “AttackBehaviour” into the particular enemy’s *behaviours* with a small key value (as this behaviour has the highest precedence among all). From the initial code fragments provided in the “LoneWolf” class’s *playTurn* method, the first found behaviour of enemies from their *behaviours* HashMap will be executed, which fulfil the precedence requirement mentioned in the specification.

To check if two enemies are of the same type, we can simply access the name of enemies via the enemy instance’s implicit *toString* method, and then check if they are of the same type via *actor.equals(otherActor)*. This is because from the engine code namely “Actor” class, we can see that the overridden *toString* method will return the name of the actor. This implementation prevents the use of *instanceof* to check the class of enemy instances as it would somewhat break the object-oriented design principle.

Requirement 2

The key concept to be implemented in Requirement 2 is that the player should own runes by defeating enemies directly. By holding some amount of runes, the player can perform purchase and sell actions with the trader. New classes that are introduced in Requirement 2 are actors include **Trader** and **MerchantKale**; items include **Runes**; all weapons include **Grossmesser**, **Uchigatana**, **GreatKnife** and **Club**; lastly, player actions include **PurchaseAction** and **SellAction**.

To start the design, (association of player → runes) the player should have runes as an item in the inventory, where one player only owns one runes, while one runes only belongs to one player. We thought of only having an instance of int amount to keep track of the total amount of runes that the player currently has instead of having an instance of runes. This however violates the SRP principle because runes should have a specific responsibility as a unique item to execute relevant operations such as addition/substraction of runes amount, keeping track of total amount etc.

Our team had made the decision to let every player have their single instance of “Runes” so that it is easier to manage. For example, when the player picks up or drops runes, we just have to simply modify the total amount of runes by basic addition and subtraction in the “Runes” class itself, reducing any unnecessary dependencies between classes and ensuring SRP. Also, this is where another new class called “PickUpRunesAction” comes in handy and SRP applies here as well such that this newly created class can have its own unique functionalities (other than those from its “PickUpItemAction” parent class) such as incrementing the targeted player’s runes amount, printing specific descriptive string to the console, and perhaps other relevant methods in the future, making it much more extensible. These implementations could result in high cohesion and low coupling between relevant classes.

An alternative to the implementation above could be creating multiple instances of “Runes” whenever the player drops their runes (remove from player’s item inventory) or picks up the runes (add into player’s item inventory). However, this implementation is not so preferrable as it would probably result in lengthy code or complex implementation. For instance, when the player drops runes, we have to do a *for* loop statement iterating through all items in the player’s item inventory, checking if the particular item is an instance of “Runes”, and lastly remove these runes instances from the inventory. Also, considering the last task (Game Requirement 2) where a player can buy weapons from the trader, it would not be practical if we want to subtract and remove a particular amount of runes from the multiple “Runes” instances in their inventory.

By defeating enemies, the player will be rewarded a certain amount of runes within a specific range number. Since the minimum and maximum runes that each enemy will drop is fixed, we design the system by initialising final static variables for minimum and maximum runes. This achieves the principle and makes the maintenance of code easier. Our team will be using The final static variables will be passed into the Random Number Generator to generate a number which is the exact amount of runes that should be rewarded to the player.

Trader is designed as an abstract class because it is assumed that different versions of trader can exist to interact with actors in the game. Hence, if there’s any new trader added to the game with different behaviours or provide different services, our existing system can be extended easily by creating a new concrete class and extending Trader class which achieves the OCP principle

and makes our system extensible in future. In Requirement 2, only one trader can exist in the existing system which is named Merchant Kale.

The player can purchase or sell weapons to the trader. Since the functionality of purchase and sell is slightly different, a PurchaseAction class and SellAction class is created to implement respective functionality when the action is executed. This achieves the SRP principle as we planned to not combine every trade-related implementation in one single class.

Weapons such as Uchigatana and GreatKnife and Club can be purchased/sold by the player from/to the trader; and Grossmesser can only be sold by the player to the trader. To ensure correct capability of weapons to be purchased and sold, different implementations were discussed such as interface, initialization of instance variable and enumeration:

First alternative is to create an interface, called Tradable that contains two methods that return boolean value which are sellable() and purchasable(). This interface should be implemented by all weapon items and allow the trader to check if a weapon can be purchased or sold by the player. However, this defeats the purpose of interface class where only class has the capability should implement the interface class. Returning a false should means the class cannot have the capability, hence logically this particular class should not implement the particular interface. Weiyo very concise

This gives us an idea of a second alternative which is the initialization of instance variables in each weapon class to return a boolean value whether the weapon is purchasable and sellable. This however breaks the OCP principle since we are not allowed to modify the engine code of WeaponItems to have the instance variables, hence the need to declare the variables in each and every weapon subclass. This can cause repetitive codes. amazing

Another implementation is to have enumeration checking if a weapon has the capability of PURCHASABLE and SELLABLE. This can prevent checking via *instanceof* which are important to achieve OCP principle. (prevent … *instanceof* as this would break the object-oriented design principle)

However, the only tradeoff of using enumeration we discovered so far is that we must make sure to always remember to manually add the SELLABLE and PURCHASABLE capabilities to every weapon item that can be purchased/sold, in their own constructors. Meaning to say, forgetting to manually add either of these two capabilities to a weapon will cause the player to be unable to buy/sell the particular weapon item, which might be prone to logic error in Assignment 2.

→ Kale cannot move around, and player cannot attack them

- Enumeration *HOSTILE_TO_ENEMY* in constructor
- Dont add WanderBehaviour()
- Only player can enter Floor, enemy cannot
- Bao wei trader with floors

Requirement 3

This UML diagram represents the object-oriented system for Game Requirement 3 in which the player is given an option to reset the game if the player is resting on the Site of Lost Grace. Also, every player will start the game with Flask of Crimson Tears, which is a special item that allows the player to restore their health by 250 points, and it can be used twice as maximum. The amount of usage will be reset to its maximum once the game is reset by the player. Last but not least, based on this game requirement, the player will drop their total runes on the last location just before they died, and if the player died again without picking up the dropped runes, the runes will be gone from the game map.

A new class called “FlaskOfCrimsonTears” is created which extends the “Item” abstract class to reduce repetition of code, which obeys the DRY principle. SRP is also followed as this newly created class will mainly be responsible for managing the usage of this specific item by the particular player, without coding all these relevant methods inside the existing “Player” class itself, resulting in high cohesion and low coupling. Since this unique item cannot be dropped, therefore when calling *super()* to inherit the instance attributes of “Item” class in the constructor, we can just pass a boolean of *false* into the *portable* parameter to indicate that it cannot be dropped.

In order to prevent excessive use of literals in the code (for Assignment 2 later), a final static variable can be used for initialising the maximum usage of this flask of crimson tears (which is fixed at 2 for every instance), so that it can be accessed by other classes and clarity of code could be improved. Whenever the Flask of Crimson Tears is not used for twice or more by a player, in every round or every player’s play turn, it will provide an option for the player to consume this special item. So, “ConsumeAction” class extending “Action” class is created to handle this process by incrementing targeted player’s health points, incrementing the usage by the particular player, and printing descriptive string to the console, obeying the SRP without putting all these implementations inside the “FlaskOfCrimsonTears” class.

Since the Site of Lost Grace is a new type of ground, a class called “SiteOfLostGrace” is created and it extends the “Ground” class. This class is made abstract so that it cannot be instantiated by the other classes, as our team has assumed that there will be more specific sites of lost grace existing in the future. Therefore, instead of direct instantiation of “SiteOfLostGrace”, a new class called “TheFirstStep” is created which is a more specific instance of “SiteOfLostGrace”. This implementation can help to improve the robustness and maintenance of code in order to make it more extensible in the future especially if there is more than one “TheFirstStep” sites on one single game map, or another specific site of lost grace such as “TheSecondStep” (which may have its unique functionality) being invented for the game. This implementation also follows the SRP and OCP as we just need to modify the subclasses (if there is any unique functionality mentioned) without making any changes to the “SiteOfLostGrace” parent class which provides common functionalities for all of its subclasses.

When the player enters this specific ground, which is The First Step, the game will directly be reset by calling the method in the “ResetManager” class. This can be done by checking if the current location is containing an actor, since only a player (an instance of “Actor”) who has a capability of RESTING (enumeration) can only enter this specific ground. This implementation helps to reduce dependency between “Player” and “SiteOfLostGrace” as we do not have to use *instanceof* to check whether or not the actor standing on The First Step is a player, as this would

break the object-oriented principle. Initially our group thought of an alternative to implement this by creating a new subclass of “Action” class called “ResetAction” to manage the game reset, but eventually we found this option impractical since the player will not be given a choice to reset the game and therefore it is not suitable for this subclass to implement the “menuDescription” abstract method from the “Action” abstract class, as this method is responsible for printing out the player’s selectable action in the console.

Since it is told that even if the player dies the game will be reset, so the “DeathAction” class can as well call the “ResetManager” to implement the functionality of game reset which is similar to the condition when the player is resting on The First Step. This follows the DRY principle. However, the only difference between these two conditions (player dying and player resting) is that if the player is resting then his runes will not be cleared (an assumption made by our team), whereas if the player dies due to being attacked then their runes will be dropped onto the location just before they die. Likewise, we assumed that if the player died, their weapons and items (except runes) will not be dropped but both their inventories will be cleared.

So, our team decided to check whether or not the currently dying actor is a player, by using enumeration of RESTING, instead of using *instanceof* which is not encouraged as explained above. If the currently dying actor is a player, then an additional process will be executed by using the *lastAction* argument in the “Player” class’s *playTurn* method to remember their last move executed. To get back the last location where the player stayed just before they died, our team agreed to access the last move executed by the player in console via *hotkey* method provided in the engine code and trace back the previous location of player, in order to drop their runes there. This checking process is considered robust since there are only a limited possible moves (maximum 9) that can be performed by a player (cannot be further extended in the future). Then, this particular location will be assigned to an instance variable inside the “Runes” class such that logically the runes will remember the location where they are being dropped onto. Therefore, if the player died again without picking up the runes dropped previously, then we can simply access the runes’ location via its instance variable and remove the runes from the game map.

Our team had made the decision to let every player have their single instance of “Runes” so that it is easier to manage. For example, when the player picks up or drops runes, we just have to simply modify the total amount of runes by basic addition and subtraction in the “Runes” class itself, reducing any unnecessary dependencies between classes and ensuring SRP. Also, this is where another new class called “PickUpRunesAction” comes in handy and SRP applies here as well such that this newly created class can have its own unique functionalities (other than those from its “PickUpItemAction” parent class) such as incrementing the targeted player’s runes amount, printing specific descriptive string to the console, and perhaps other relevant methods in the future, making it much more extensible. These implementations could result in high cohesion and low coupling between relevant classes.

An alternative to the implementation above could be creating multiple instances of “Runes” whenever the player drops their runes (remove from player’s item inventory) or picks up the runes (add into player’s item inventory). However, this implementation is not so preferrable as it would probably result in lengthy code or complex implementation. For instance, when the player drops runes, we have to do a *for* loop statement iterating through all items in the player’s item inventory, checking if the particular item is an instance of “Runes”, and lastly remove these runes instances from the inventory.

The “Resettable” interface provided in the game package is helpful for identifying all the actors that can be reset via the *reset* method as it keeps a promise such that every resettable actor should implement (override) this *reset* method inside their specific classes.

As indicated in the UML diagram, our team made every specific instance of enemy, namely LoneWolf, GiantCrab, HeavySkeletalSwordsman and PileOfBones, to implement the “Resettable” interface. Our aim is to make the game system more robust as we assume that not every enemy can be reset. To elaborate, initially we came up with an alternative to just let the “Enemy” and “HarmlessEnemy” classes implement the “Resettable” interface, however this would make the system inextensible such that if in the future there is any other type of “Enemy” who cannot be reset, we will have to modify the previous code in order to fulfil this specific criterion, making the code or system hard to maintain.

The “Player”, “FlaskOfCrimsonTears” and “Runes” will also be implementing this interface as some unique processes should be carried out or happening during the game reset in their respective *reset* methods, for example resetting the player’s hit point to maximum, resetting the usage amount of Flask of Crimson Tears to maximum, as well as removing the dropped runes from the game map if the player died again without picking up the runes dropped previously during their first death.

Last but not least, we chose to let the “ResetManager” manage an ArrayList of type “Resettable” instead of multiple ArrayLists of different types, as this is to follow the Dependency Inversion Principle (DIP) by avoiding dependencies between multiple classes that are resettable with the “ResetManager” class. Since the “ResetManager” will mainly be responsible for resetting the game. Hence, this can be done by coding a *for* loop statement in the “ResetManager” class which will iterate through all the instances of “Resettable” and execute their respective *reset* methods. Here we make sure the Liskov Substitution Principle (LSP) is followed.

Requirement 4

In Requirement 4, the player is given a choice to choose a character (player class) to play before the game starts. Each character has different hit points and weapon items for the player to start off the game. Similar to the weapons introduced in previous requirements, the weapons held by different characters have attack action. Except for Uchigatana and GreatKnife, these weapons also have special skill actions where the player can choose if they want to perform the special attack, given that the mentioned weapons is inside the player's inventory. New classes that are introduced in Requirement 4 are utils include **MenuManager**; all player classes include **Samurai**, **Bandit** and **Wretch**; all weapon items include **Uchigatana**, **GreatKnife** and **Club**; all weapon types include **Katana**, **Dagger** and **Hammer**; lastly, weapon special skill actions include **QuickStepAction** and **UnsheatheAction**.

Before the game begins, the player needs to choose a player class. This is where the Menu Manager class will have menu items to be printed onto the console and get the player's input for the player class that they want to play the game with. By having the selected class, Menu Manager will create an instance of the player with the specific player class, and the game will start. Here, creating a Menu Manager class achieves the SRP principle where this class is solely responsible for printing out available options for player classes and getting the player's input. If we are to design this functionality in other existing classes like Player or Application (driver class), it will cause the them to be a GOD class and have different behaviours.

Now, since the player should choose one of the player classes to play the game, the instance of the subclasses should be created instead of an instance of the player class. In fact, the player class should now be an abstract class to prevent an instance of the Player being created. With this, the design follows the OCP principle, where one concrete class can extend the Player class and modify the functionalities within its own class, instead of modifying the Player parent class itself.

Similar concept for WeaponItems and WeaponTypes, an instance of WeaponItems and WeaponTypes should not be directly created because each weapon has different functionalities and should be modified in their own classes to achieve the OCP principle.

To explain more about the OCP principle in the player class, weapon item and weapon type, it is assumed that there are more players, weapons and weapon types that can exist to interact with actors in the game. Hence, if there's any new element added to the game, our existing system can be maintained and extended easily. For example, new player elements can inherit the player class; new weapon elements can inherit the corresponding weapon type; new weapon type elements can inherit weapon items. New elements are allowed to implement its specific functionalities in its own class without modifying the parent class. This also potentially reduces repetition codes as usually the parent classes will already contain relevant common attributes or methods that are generic to all of their subclasses, in which we can just call *super()* method in the subclasses where necessary and applicable, in order to inherit all the commonalities existed in their parent classes without having to repeat the similar implementations.

If the player has the weapons of Uchigatana or GreatKnife in the inventory, the player can choose to perform a special weapon skill action which is unsheathe action or quick step action. To maintain the SRP principle, a new class of each special weapon skill action should be created because they have specific responsibilities when the action is executed, as well as to keep the code more understandable and keep our system consistent. However, unsheathe action and

quick step action have the similar behaviour as attack action. We design to make unsheathe action and quick step action to inherit attack action to reduce repetition code. All we can do during execution is to call the parent's class method via *super()*, then modify a little implementation from parent class such as doubling original attack while changing attack accuracy (for unsheathe action), or to perform the additional functionality of each special weapon skill action (additional functionality such as moving one step away from the enemy after attacking (for quick step action). This also achieves the LSP principle since unsheathe action and quick step action theoretically uphold the same behaviour as attack action and becoming the subclass of attack action does not change the meaning of attack action's behaviour.

Requirement 5

The UML diagram above represents an object-oriented system for Game Requirement 5 in which it is a combination of Game Requirement 1,2,and 3 .The only main differences are that more types of enemies are added to the game and the map will be divided into two halves, with the West side spawning Heavy Skeletal Swordsman from the Graveyard, Lone Wolf from a Gust of Wind, and Giant Crab from a Puddle of Water. On the East side of the map, Skeletal Bandit will spawn from a Graveyard, Giant Dog from a Gust of Wind, and Giant Crayfish from a Puddle of Water.

Similar to the enemies in requirement 1 three enemy subclasses are created which are “SkeletalBandit”, “GiantDog”, and “GiantCrayfish”. This follows the DRY principle which avoid repetition of code and OCP principle where if there is other enemies introduce in the future , only a new enemy subclass need to be created without modified the “enemy ” abstract class.

In our design, we will make use the GameMap’s width to divide the starting map into west and east half .By comparing the location of the each ground to the middle x coordinates that calculated using width, we can determine which side of the map a location belongs to. If a location's x coordinate is less than the middle x coordinates, it will belong to the west side of the map and spawn Heavy Skeletal Swordsman, Lone Wolf, and Giant Crab. On the other hand, if a location's x coordinate is greater than the middle x coordinates, it will belong to the east side of the map and spawn Skeletal Bandit, Giant Dog, and Crayfish