FIT2099 Assignment 2

UML Diagrams & Design Rationales (UPDATED)

Group name: MA_Lab06_Group6

Group member:

1. **Che'er Min Yi 32679939**
2. **Chong Ming Sheng 32202792**
3. **Lam Xin Yuan 32245211**

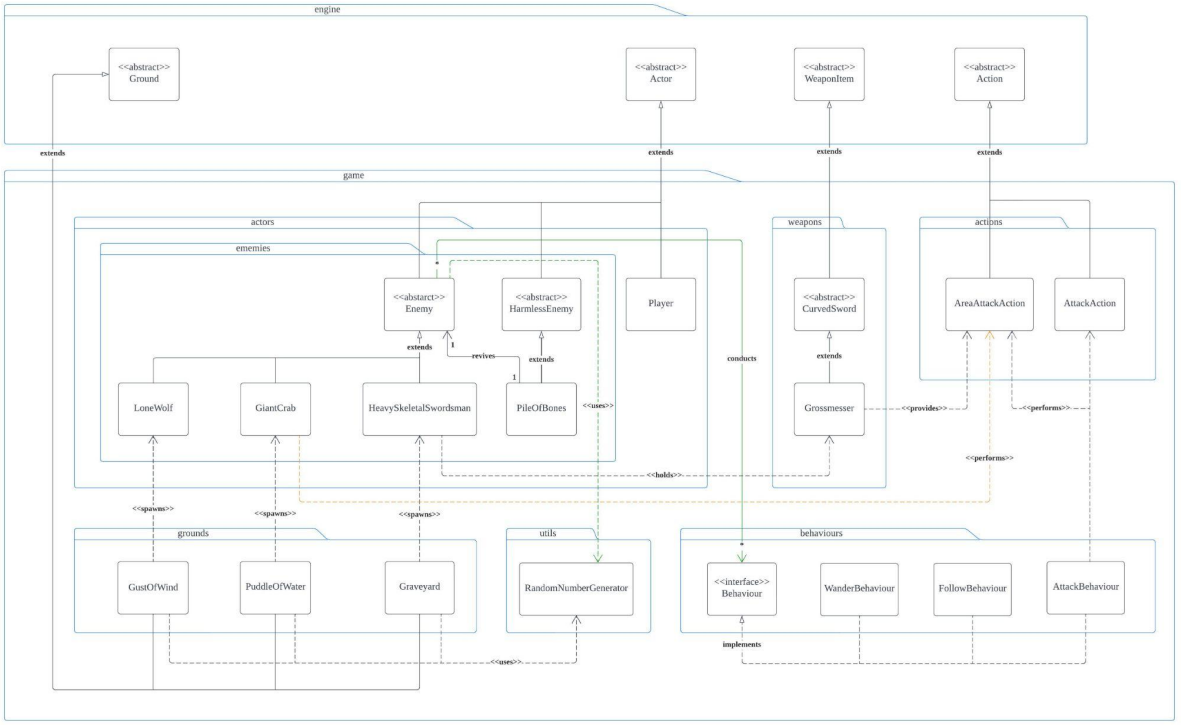
Link to group's contribution logs spreadsheet:
https://docs.google.com/spreadsheets/d/1vDWOl0y7OB4fWNsCS8IczHFDFUI4hhW5-LMqDeUrgBY/edit?usp=sharing

Below is the list of keywords and abbreviations for the design principles that our team used in the design rationales:

1. Don't repeat yourself - DRY
2. Single Responsibility Principle - SRP
3. Open-closed Principle - OCP
4. Liskov Substitution Principle - LSP
5. Interface Segregation Principle - ISP
6. Dependency Inversion Principle - DIP
7. Excessive use of literals

Note: The modification to the UML diagram and design rationale for each requirement in Assignment 2 comes after what has been written in Assignment 1. The title is highlighted yellow.

Game Requirement 1 (Assignment 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 the specific spawning chance provided. To follow the DRY 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 SRP as each subclass is responsible for defining its unique chance of spawning and instantiating its corresponding enemy.

In requirement 1 part B, we will be creating three new classes for hostile creatures: “HeavySkeletalSwordsman”,”LoneWolf”, and “GiantCrab” which extends the “Enemy” abstract class. “Enemy” class is made abstract because only instances of the subclass should be created when an enemy exists. This approach follows the DRY principle which

avoids repetition of code since all the enemy classes share common attributes and methods. 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.

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 players 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 the 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 displayed 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 enemies 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 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 follows OCP.

Our group decided to directly add one grossmesser into the Heavy Skeletal Swordsman weapon inventory in the Heavy Skeletal Swordsman’s constructor so that there is no need to manually add weapon for Heavy Skeletal Swordsman in the driver class which would introduce more dependency and violate SRP. Another advantage of this implementation is

that everytime a new Heavy Skeletal Swordsman is created, it will automatically hold a Grossmesser in its weapon inventory.

Pile of Bones would only appear when a Heavy Skeletal Swordsman is defeated. It will possibly be removed and then will revive the Heavy Skeletal Swordsman when it is attacked or hit by the player. To achieve this requirement, our team decided to add capability of *RESPAWNABLE* (enumeration) to the Heavy Skeletal Swordsman in the constructor, and then in the “DeathAction” class’s *execute* method, we would check whether or not the currently dying enemy has the *RESPAWNABLE* capability. If the statement returns true, then it means the dying enemy is a Heavy Skeletal Swordsman and therefore we can directly remove the dying enemy and create an instance of “PileOfBones” on that location.

However, we found out that this implementation has a limitation as it will only work if there is only one type of revivable enemy (which is Heavy Skeletal Swordsman) forever in the game. This is because if the Pile of Bones is not hit by the player within three rounds, then we will let this class handle the revival process of Heavy Skeletal Swordsman by simply creating a new instance of Heavy Skeletal Swordsman on the same location. Therefore, if there is more than one type of revivable enemies introduced into the game system in future, we will have to modify our previous implementation to ensure not only Heavy Skeletal Swordsman will be revived at the same location, but as well as the other revivable enemies.

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 utilise 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.

Here comes to our team’s final decision which is declaring an instance variable of type “Enemy” in the “PileOfBones” class. This instance variable is meant to remember the dying (harmful) enemy whom the Pile of Bones appears from. In this case, every instance of Pile of Bones is able to know the type of enemy that they should revive if they are not hit by the player within three rounds. This implementation could ensure an extensible and robust game system.

To drop Grossmesser after a Heavy Skeletal Swordsman is defeated (after a pile of bones is destroyed), our team agreed to pass the original weapon from the weapon inventory of Heavy Skeletal Swordsman to the Pile Of Bones. Relating this implementation to how we implement the revival of *RESPAWNABLE* enemies as explained above, since the Pile of Bones can remember the dying enemy, therefore we can get the first weapon of the particular enemy and put it into the Pile of Bones’s weapon inventory, and then clear the weapon inventory of the dying enemy. This is due to the assumption that each enemy can only have one weapon for simplicity (as mentioned by one of the staff in Ed Forum).

In this case, if the Pile of Bones is not attacked by the player within three rounds, its first weapon will again be passed to the original revivable enemy who is accessible by the

instance variable suggested previously. Else, if the Pile of Bones dies due to being attacked by the player, then it will go through the normal “DeathAction” in which its available weapon items in the weapon inventory will be dropped onto the ground where it is located.

We planned to assign the responsibility of keeping track of the number of rounds in the game to the “PileOfBones” class as this counter will be incremented every time when the game system goes through the *playTurn* method of the “PileOfBones” class. So, the checking condition of whether or not three rounds have passed can also be done in this *playTurn* method along with the logic of implementation. This follows the SRP as only the “PileOfBones” class will be responsible for its relevant methods and attributes regarding the enemies’ revival process.

Heavy Skeletal Swordsman carries around a weapon called Grossmesser, so a new class “Grossmesser” is created which extends the “CurvedSword” abstract class. 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 OCP and 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 including Heavy Skeletal Swordsman 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 player’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.

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. Besides, this also follows the SRP as “AreaAttackAction” takes on the responsibility of handling any area attacks which separates responsibility of normal attack to the unique area attack.

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.

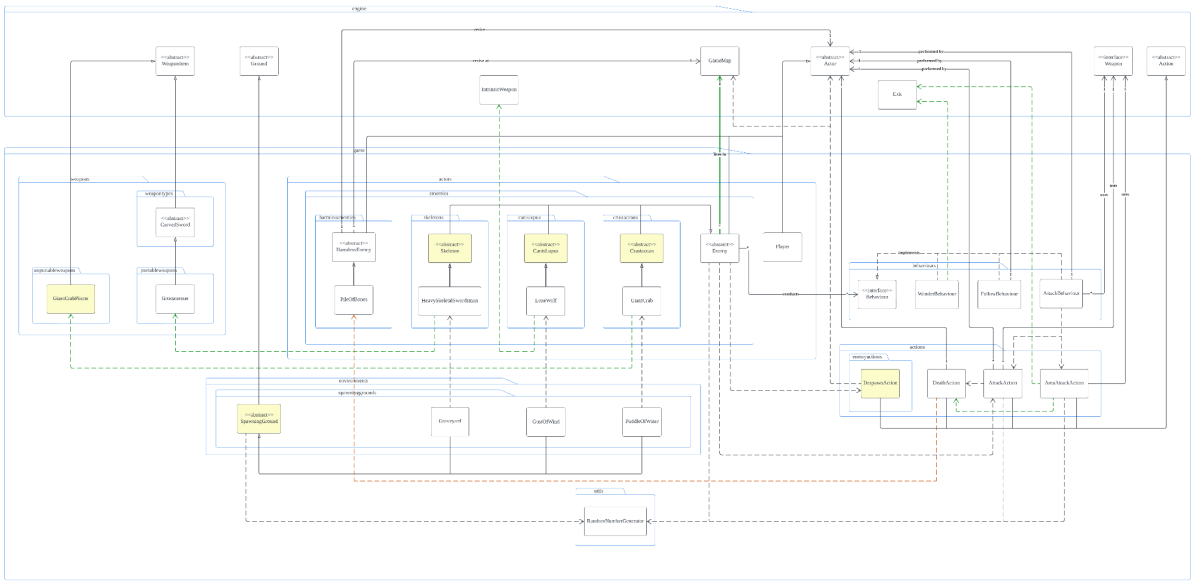
To check if two enemies are of the same type, we can simply access the name of enemies via the enemy instance’s implicit *toString* method, and then check if they are of the same type via *actor.equals(otherActor)*. This is because from the engine code namely “Actor” class, we can see that the overridden *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.

At each turn, an enemy has a 10% chance of being despawned unless they are following the player. In order to meet this requirement, our team decided to add a *FOLLOWING* capability to the enemy once they are close to the player and this capability will not be removed because the enemy will continue to follow the player until they are defeated. Then in “Enemy” abstract class’s *playTurn* method, we would check whether or not the current enemy has the FOLLOWING capability. If the statement returns true, the enemy cannot be despawned, else the enemy has a possibility to despawn.

To implement the despawn chance, our team has made an assumption that not all the enemies added in the future will guarantee to have the same despawn chance of 10%. Since all the enemy instances of the same class have the same despawn chance, we decided to add a final static class variable to each enemy subclass which can be used for initialising their corresponding despawn chance. Then, in the constructor of each subclass, we will pass this class variable as a parameter into the *super()* method which then will initialise the instance variable of despawn chance in “Enemy” abstract class.

This approach prevents excessive use of literals in the code (for Assignment 2 later) and is also more efficient as class variables use less memory compared to instance variables because only one copy of the class variable exists in memory and this copy can be shared among all instances of the same specific enemy class. Besides, this implementation can help to improve the robustness and maintainability of code because if in future there are any changes to the despawn chance of a particular enemy, it is much easier to modify the class variable instead of manually searching and replacing every occurrence of despawn chance magic number/value. This approach will also make the system more extensible, as if there are new enemies being added into the game in the future, any changes made regarding the despawn chance will only happen in the enemy subclasses, whereas “Enemy” abstract class will remain unchanged. This follows OCP.

Modification to Game Requirement 1 (Assignment 2)



(Please refer to the appendix section for the higher quality of the UML Diagram)

1. Introduced “DespawnAction” class extending from “Action”
    - Instance of “DespawnAction” will only be executed by the enemies whenever they meet a chance to despawn from the map in every round.
    - Previous implementation: Directly remove the enemies from the map in the “Enemy” abstract class.
    - Con of previous implementation: Break SRP principle as the “Enemy” class itself should not handle this despawning functionality, but instead, let the “DespawnAction” to take on this responsibility.
    - Decision: In the “Enemy” class’s *playTurn()* method, whenever the chance of despawning is met, then return the instance of “DespawnAction” to be executed by this particular enemy subclass in this round.
    - When “DespawnAction” is executed, then this particular enemy will be removed from the map.
    - OOP principle: SRP principle
    - Pro: Reduce the workload to be handled in “Enemy” class.
    - Pro: Improve the clarity of code, so that apparently the enemy will not simply remove themselves from the map during their own playing round (when executing the *playTurn()* method).

2. Created “Skeleton”, “CanisLupus” and “Crustacean” abstract classes.
    - These abstract classes define the type of different enemies.
    - Previous implementation: Specialized enemies directly extend from the “Enemy” abstract class.

- Con of previous implementation: We have to manually handle the commonalities or shared behaviours as well as attributes between the same type of enemies. This reduces the code maintainability.
- Con of previous implementation: From the UML diagram's point of view, it does not give any hint such that some enemies are of the same type.
- Decision: Let the specialized enemies of the same type inherit the same abstract class which is used to define their type and commonalities.
- OOP principles: OCP principle and DRY principle.
- Pro: Reduce repetition of code in every specialized enemy subclass. For example, adding the same capability in the constructors of many different specialized enemy subclasses of the same type.
- Pro: Extending from either one of these existing abstract classes means this particular enemy already has the characteristics provided by its type.
- Future extension: If there are other enemies of the same existing types being introduced into the game, we can make these new enemies inherit their corresponding enemy type's abstract class, so that they can share the same attributes or behaviours without having to manually handle it.

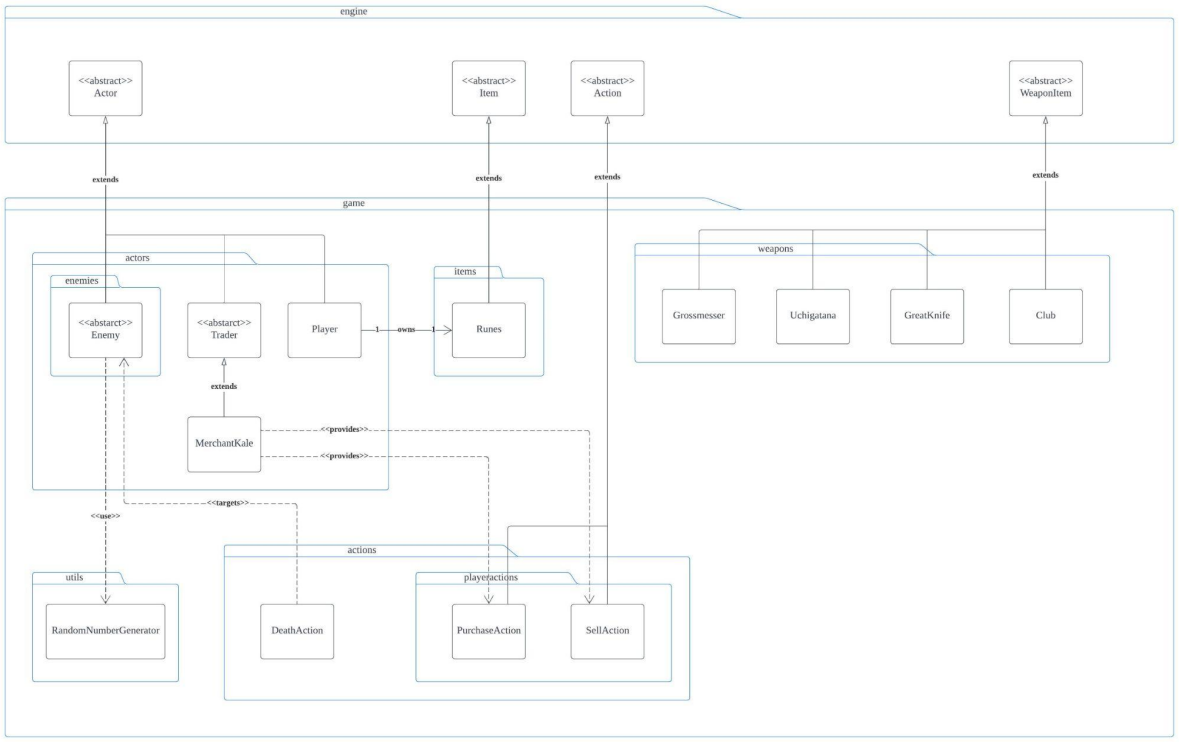
3. Created "SpawningGround" abstract class.
    - This abstract class is used to differentiate between grounds that can spawn enemies and grounds that cannot.
    - Previous implementation: Specialized ground subclasses such as "Graveyard", "PuddleOfWater" and "GustOfWind" directly extends from "Ground".
    - Con of previous implementation: Unable to differentiate between grounds that can spawn enemies and grounds that cannot. If in the future these enemy-spawnable grounds have some commonalities or shared behaviours among themselves, then we will have to apply modifications to every of these specialized ground subclasses.
    - Decision: Our team agreed to make "Graveyard", "PuddleOfWater" and "GustOfWind" extend from "SpawningGround", whereas "SpawningGround" extends from "Ground".
    - OOP principles: OCP principle and DRY principle
    - Pro: "SpawningGround" can be extended by every specialized ground that can spawn enemies so that they can share the same characteristics that are provided in the parent class.
    - Pro: Any uniform changes applied to the functionalities of these enemy-spawnable grounds will just require us to modify the parent class, resulting in a more maintainable code system.
    - Pro: If the specialized ground subclasses have some commonalities, we have to only add code to their parent class.
    - Future extension: A good example of how the "SpawningGround" helps to produce a more extensible game system can be seen in Game Requirement 5 at the later section of this document.
    - Future extension: Another good example will be: imagine in every round of the game, only the enemy-spawnable grounds will have the same chance to

turn into an obstacle, say giant rocks, then we will just code the implementation in the *tick()* method of the “SpawningGround” abstract class, without coding them repetitively in the specialized subclasses.

4. Added “GiantCrabPincer” class extending from “WeaponItem”
    - Based on the requirement, it looks more appropriate for “GiantCrabPincer” to be part of the weapon items existing in the game, which is only owned by “GiantCrab”
    - Previous implementation: Previously we treated “GiantCrabPincer” as an intrinsic weapon of “GiantCrab”
    - Con of previous implementation: “GiantCrabPincer” cannot provide any kind of possible special abilities to “GiantCrab”, making the system inextensible.
    - Decision: We treated “GiantCrabPincer” as a kind of “special” weapon item that is only owned by “GiantCrab”, and extends this class from “WeaponItem”. Since this “special” weapon item is not supposed to be dropped by “GiantCrab” for now, then we can make it unportable via *togglePortability()* method provided in the “Item” class.
    - OOP principles: OCP principle.
    - Pro: We can easily make the “GiantCrabPincer” to be possible for offering different attack behaviours if the game extends, by creating specialized subclasses of “GiantCrabPincer” parent class, which may have different unique characteristics but in some way similar to each other.
    - Future extension: Imagine if the pincer of “GiantCrab” can be upgraded or downgraded during the game, for example if it hits certain number of enemies then the pincer can be upgraded from “BeginnerGiantCrabPincer” to “IntermediateGiantCrabPincer”. Both of these pincers have similar attack behaviours but perhaps some attributes are different for example the attack accuracy and damage points, as well as how easily they can be defeated by others.

5. Added dependency relationship “AreaAttackAction” to “RandomNumberGenerator”
    - Based on the requirement, if an area attack action is performed, then every target in the surroundings of the attacker will have an independent chance of getting hit.
    - Previous implementation: No use of “RandomNumberGenerator”
    - Con of previous implementation: Breaking the game rule or not following the game requirement, as every target around the attacker will definitely be hit in this case.
    - Decision: Use the “RandomNumberGenerator” in the *execute()* method of the “AreaAttackAction” so that for every target in the surroundings, the attacker can only attack them successfully if it meets the specific attack accuracy of the specific weapon item.
    - Pro: To follow the game requirement and ensure the correct effect of “AreaAttackAction”

Game Requirement 2 (Assignment 1)



The UML diagram above represents the object-oriented system for Game Requirement 2 where 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. For the current implementation, Uchigatana, Great Knife and Club are purchasable by the player from the trader whereas Uchigatana, Great Knife, Club and Grossmesser are sellable by the player to the trader if the weapons are available in their inventory.

Initially, 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 because runes should have a specific responsibility as an unique item to execute relevant operations such as addition or subtraction of runes amount, keeping track of total amount etc. Hence, to ensure the SRP holds, we decided to create a new class called “Runes” which extends the “Item” abstract class where “Runes” class will mainly be responsible for managing all relevant operations instead of coding them inside the existing “Player” class, resulting in high cohesion and low coupling.

One of the implementations that we discussed was to create 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 preferable 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 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.

Hence, 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 methods in the “Runes” class itself, reducing any unnecessary dependencies between classes and ensuring the SRP. Also, this comes in handy when we need to extend other relevant methods in the future. These implementations could result in high cohesion and low coupling between relevant classes.

Moving on, 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 to prevent excessive use of literals in the code (for Assignment 2 later) and makes the maintenance of code easier. With the final static variable initialised, we just pass in the variable into the “RandomNumberGenerator” class to generate a number which is the exact amount of runes that should be rewarded to the player. Hence, if there’s any changes to the range number in the future, we can just modify at the top of every class where the final static variable is initialised without going through the implementation to change one by one.

Another thing to be aware of is that the player will be rewarded runes if they kill the enemy directly, whereas the player should not be rewarded with runes if one enemy is killed by another enemy. To ensure the enemy is killed by the player, our group decided to check whether the attacking actor is the player via enumeration of RESTING (provided that we decided to add this capability to players only since only players can have the ability to rest), instead of using *instanceof* during the execution of *deathAction()* as this would break the object-oriented principle.

If the attacking actor is indeed the player, runes will be rewarded. Initially, our team’s implementation is to have the enemy dropping runes onto the ground and the player picks it up to put into the inventory. This however is an extra process to drop the runes and pick up again which causes redundancy, since the runes will enter the inventory eventually. Instead, we decided to utilise the method that may be implemented in “Runes” class such as *incrementAmount()* to add the awarded amount of runes directly to the player’s current total amount to make the code simpler and reduce complexity.

Furthermore, “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 from “Trader” abstract class. This achieves the OCP and makes our system extensible in future. In Requirement 2, only one trader can exist in the existing system which is named “MerchantKale”.

To ensure that “MerchantKale” cannot move around (to sit at a specific location around the building in the middle of the map), our team decide to not give an instance of *HashMap Behaviour* inside the “MerchantKale” class so that Merchant Kale will not be able to wander around the map, follow or attack any actor. Also, the fact that Merchant Kale should sit around the building in the middle of the map that is surrounded by floor and wall, only the player can approach the Merchant Kale (to appear in the Merchant Kale’s exit). This also

means that enemies should not be able to perform “PurchaseAction” and “SellAction” with Merchant Kale; and the enemies cannot attack the Merchant Kale. However, to ensure the player will not be able to attack the Merchant Kale, Merchant Kale should not have the capability of *HOSTILE_TO_ENEMY* (enumeration) initialised in its constructor.

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 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 classes having 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. This also violates the LSP if “Grossmesser” class is implementing “Tradable” interface as it will have a do-nothing implementation for *purchasable()* since the player can only sell the Grossmesser to the trader and cannot purchase from the trader.

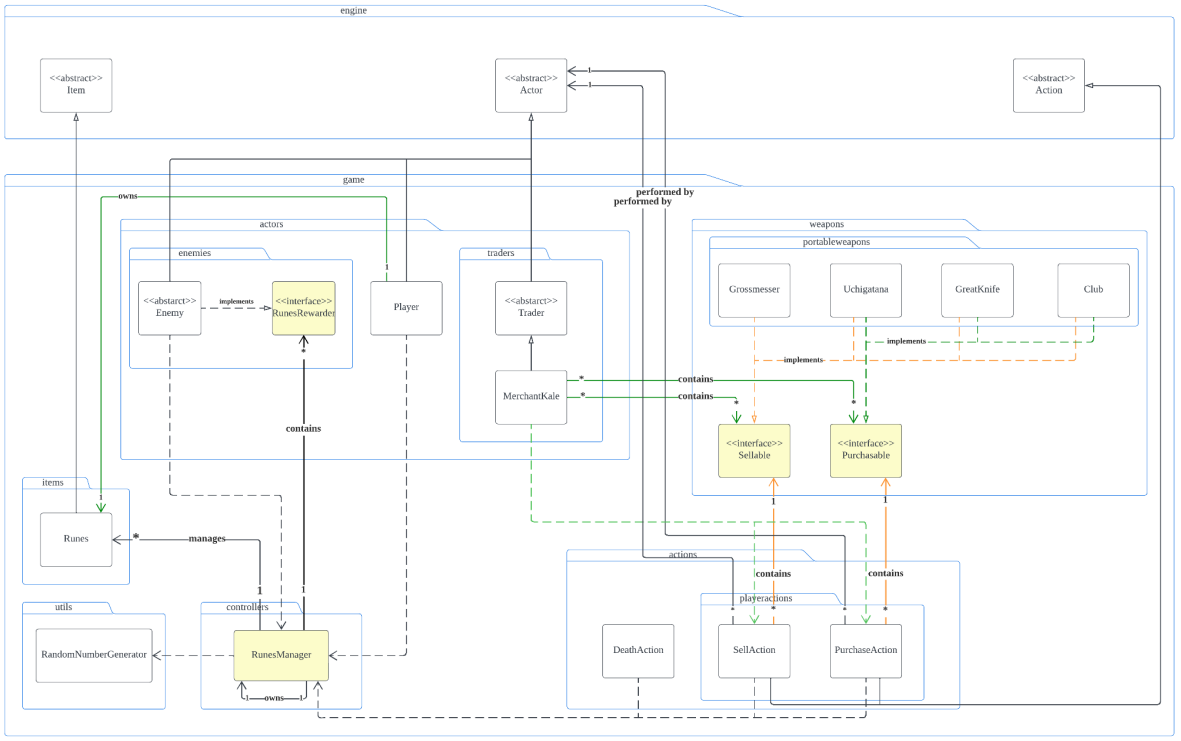
However, we can separate the interface into “Sellable” and “Purchasable” as the principle of ISP mentions. So, weapons can implement the interface accordingly depending if they are sellable or purchasable. However, if we utilise the interface in a *for* loop statement to loop through weapon items to check some conditions via polymorphism, we might still need to use *instanceof* to get the specific weapon which will break the object-oriented principle.

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 since we are not allowed to modify the engine code of “WeaponItems” class to have the instance variables, hence the need to declare the variables in each and every weapon subclass. This can cause repetitive codes.

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

With all the alternatives discussed above, our final decision is enumeration checking. 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.

Modification to Game Requirement 2 (Assignment 2)



(Please refer to the appendix section for the higher quality of the UML Diagram)

1. Added new interface “Purchasable” and “Sellable”
    - “Purchasable” interface is implemented by all weapon items that the player can purchase from the trader such as “Uchigatana”, “GreatKnife” and “Club” whereas “Sellable” interface is implemented by all weapon items that the player can sell to the trader such as “Grossmesser”, “Uchigatana”, “GreatKnife” and “Club”.
    - Previous implementation: Instead of implementing an interface, enumeration checking is applied for all weapon items that can be purchased and/or sold where the weapon item will have the capability of “Purchasable” and “Sellable” respectively.
    - Con of previous implementation: It is not flexible to retrieve some value of the purchasable item when executing “PurchaseAction” without performing downcasting which results in the violation of object-oriented principle. Same goes to sellable items when executing “SellAction”.
    - Con of previous implementation: Always have to make sure that we remember to add these capabilities in their constructors, otherwise forgetting to do so will result in specific weapon items cannot be sold or purchased, making the system more prone to logic error.
    - Decision: Allow all weapon items that the player can purchase from the trader implement “Purchasable” interface, with promised methods called *getPurchasePrice(), getPurchaseAction()* and *getPurchaseInstance()*; whereas for all weapon items that the player can sell to the trader will

implement “Sellable” interface, with promised methods called *getSellPrice()* and *getSellAction().*

- OOP principle: OCP principle and ISP principle.
- Pro: A larger interface has been splitted into smaller ones. One of the alternatives was to add a new interface of “Tradable” instead of two interfaces “Purchasable” and “Sellable”. But since purchasing and selling are two different capabilities that one weapon item can have and therefore different promised methods are required, the interface is splitted into smaller ones so that weapon items can implement the interface accordingly depending if they are sellable or purchasable.
- Pro: We can categorize every weapon item that can be purchased to implement “Purchasable” interface; and every weapon item that can be sold to implement “Sellable”. With this, the trader would be able to track all the purchasable weapon items and sellable weapon items and give allowable action to the player with the precise purchasable weapon item list and sellable weapon item list to execute “PurchaseAction” and “SellAction” respectively.
- Pro: We can have more weapon items introduced in the future that no matter if it is purchasable or sellable, can group itself to the respective capabilities (purchasing or selling) by implementing the interface accordingly.
- Pro: With these newly added weapon items implementing these existing interfaces, the promised methods will guarantee to be overridden in the specialized weapon item classes, which can help to reduce logic error.
- Future extension: If we have another weapon item that is purchasable or sellable by the player, we can first decide whether the weapon item is purchasable or sellable, or both, and then the weapon item concrete class can implement the “Purchasable” interface and/or “Sellable” interface accordingly and get all the promised method. So that the existing code can be reused to provide corresponding actions to the player in order to sell or purchase.

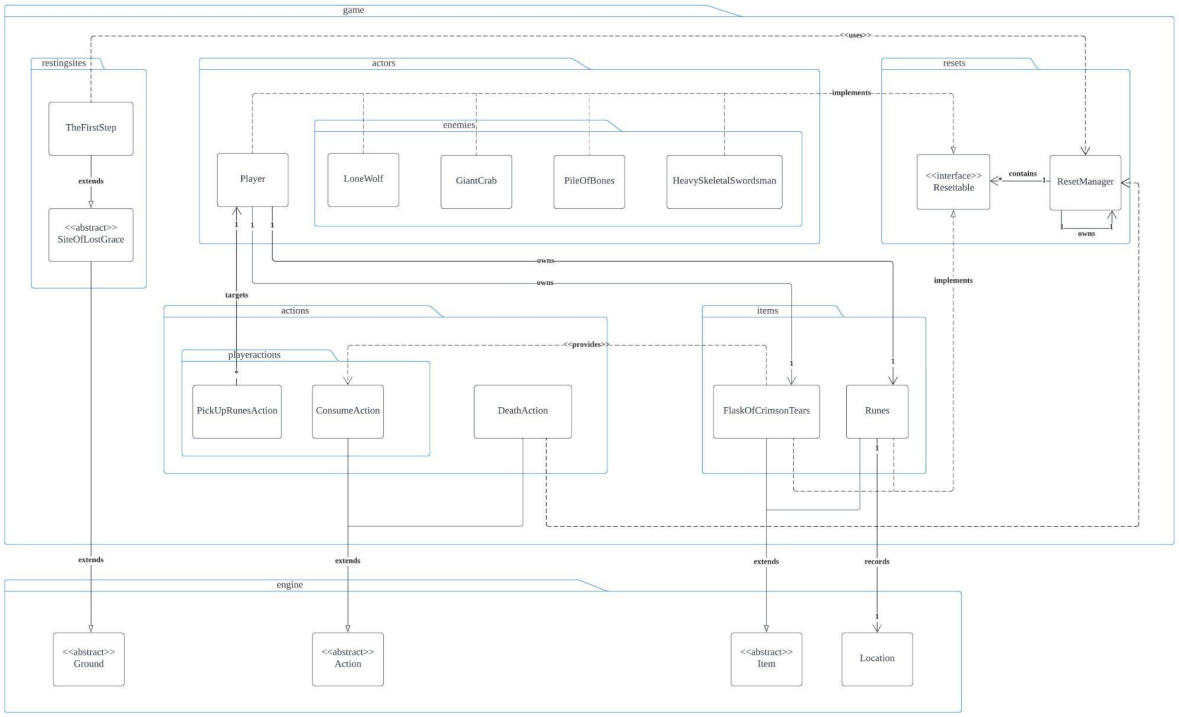
2. Added new class “RunesManager”
    - The purpose of “RunesManager” is to allow the management of “Runes” in the game such as rewarding runes, increment and decrement of runes amount and so on.
    - Previous implementation: We did not have a “RunesManager” class from the previous implementation because we think that the “Runes” class itself can manage any functionalities related to runes.
    - Con of previous implementation: According to the feedback given in Assignment 1, it was realized that managing “Runes” should be another responsibility and “Runes” item itself should not be able to manage itself.
    - Con of previous implementation: The “Runes” class will be a “God” class as it handles too many implementations.
    - Con of previous implementation: Feature Envy Code Smell issue: “Runes” class will be handling too much information that is outside of its class, increasing unnecessary relationships with other classes.

- Decision: Our team has decided to create a new class of “RunesManager” where there’s only one instance of “RunesManager” to manage all runes in the game. Some functionalities that are implemented in “RunesManager” class to manage runes include rewarding runes to the player after defeating enemies, increment or decrement of one’s runes amount, register or remove one’s runes object.
- OOP principle: SRP principle
- Pro: Differentiate the responsibility between “Runes” class and “RunesManager” class where “Runes” should just be a concrete class that acts as an item to save relevant attributes like *totalAmount.* Whereas the responsibility for “RunesManager” is mainly to manage runes as explained above, with relevant attributes methods coded in this class.
- Pro: Prevent Feature Envy Code Smell issue, as discussed above.
- Future extension: “RunesManager” will be able to manage more “Runes” objects that are added in the future. For example, if another actor, says the trader, should now be able to own a “Runes” object and trading the weapon items with the player will reflect the changes in trader’s “Runes” amount. Trader’s “Runes” object can be registered into “RunesManager” to allow “RunesManager” to manage the runes.

3. Added new interface “RunesRewarder”
    - This interface is implemented by the “Enemy” class since it is mentioned in the requirement that all enemies that are defeated by the player will reward some runes to the player.
    - Previous implementation: When the “DeathAction” of the target is executed, the reward of runes from actor target and actor attacker (the player) is directly implemented with the player’s “Runes” object” in the *execute()* method itself. In addition, after creating the “RunesManager” to manage the “Runes” object, we decided to let “RunesManager” handle this responsibility to maintain the OCP principle.
    - Con of previous implementation: “RunesManager” will not be able to access the enemy’s minimum and maximum amount to be rewarded without downcasting the actor target to be enemy, which has violated the object-oriented principle.
    - Decision: Let “Enemy” abstract class and “Pile of Bones” concrete class implements “RunesRewarder” interface, with a promised method called *getMinRewardRunes()* and *getMaxRewardRunes().*
    - OOP principle: DIP principle and OCP principle
    - Pro: “RunesManager” will have a list of “RunesRewarder” so that when “RunesManager” needs to reward player some runes after defeating the enemy, “RunesManager” can retrieve the minimum and maximum runes to be rewarded by the defeated enemy via the promised method *getMinRewardRunes()* and *getMaxRewardRunes();* Hence, we successfully retrieve some values from “Enemy” class from the point of “Actor” without downcasting the parameter from “Actor” to “Enemy” when executing reward function.

- Pro: Reduce unnecessary dependencies between “RunesRewarder” and every concrete class of enemy that can reward runes to player, as well as the “Pile of Bones”. Now, we only need to let “RunesManager” handle a list of “RunesRewarder”, and “RunesRewarder” is implemented by every class of runes source in the game.
- Pro: Any new enemies introduced in the future extending from the “Enemy” abstract class will automatically implement the “RunesRewarder” interface, if this new enemy can reward runes to player once defeated by player.
- Pro: This makes the interfaces to be open for extension, as enemies added in the future can easily implement these interfaces if they have the same functionalities.
- Future extension: If there is another enemy for example “Monster” added into the game such that it can reward the player with runes, we do not have to go back to our existing code and modify their implementations, however we can just let this “Monster” class implements the “RunesRewarder” interface and register it as a “member” of the list of “RunesRewarder” to be handled by the “RunesManager” class in terms of rewarding runes.
- Future extension: If there is another “HarmlessEnemy”, for example “PileOfRocks” being added into the game, such that this enemy is actually turned from a death of another type of typical enemy, say “Monster”. This “PileOfRocks” does not have to implement the “RunesRewarder” interface provided that it cannot reward the player with runes if they are broken by the player.

Game Requirement 3 (Assignment 1)



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 from “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 maintainability 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 weapon and item 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 the console via *hotkey* method provided in the engine code and trace back the previous location of the 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.

Since our team decided to let every player have their own single instance of “Runes” to make the system more manageable and maintainable, a new class called “PickUpRunesAction” comes in handy and SRP applies here 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 the system much more extensible. The main reason behind this is because we would like to separate the typical actions of picking up normal items with this special action of picking up runes as the player’s current runes need to be updated and they are not required to be added into player’s item inventory like the other items. As mentioned previously, we make our system consistent such that we will assign a new class for every action that can be selected by players to execute via console.

Also, considering the last task (Game Requirement 2) where a player can buy weapons from the trader, the discussed alternative of putting multiple instances of “Runes” of different values into the player’s item inventory previously would not be practical if we want to subtract and remove a particular amount of runes from the multiple “Runes” instances in their inventory. This would largely increase the complexity of code and this would probably be more difficult to implement.

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 in the game can be reset in the future. 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. Otherwise if the previous code is not modified in this alternative, this would go against the LSP because the objects of resettable parent class will no longer be replaceable with the objects of non-resettable child classes in this case.

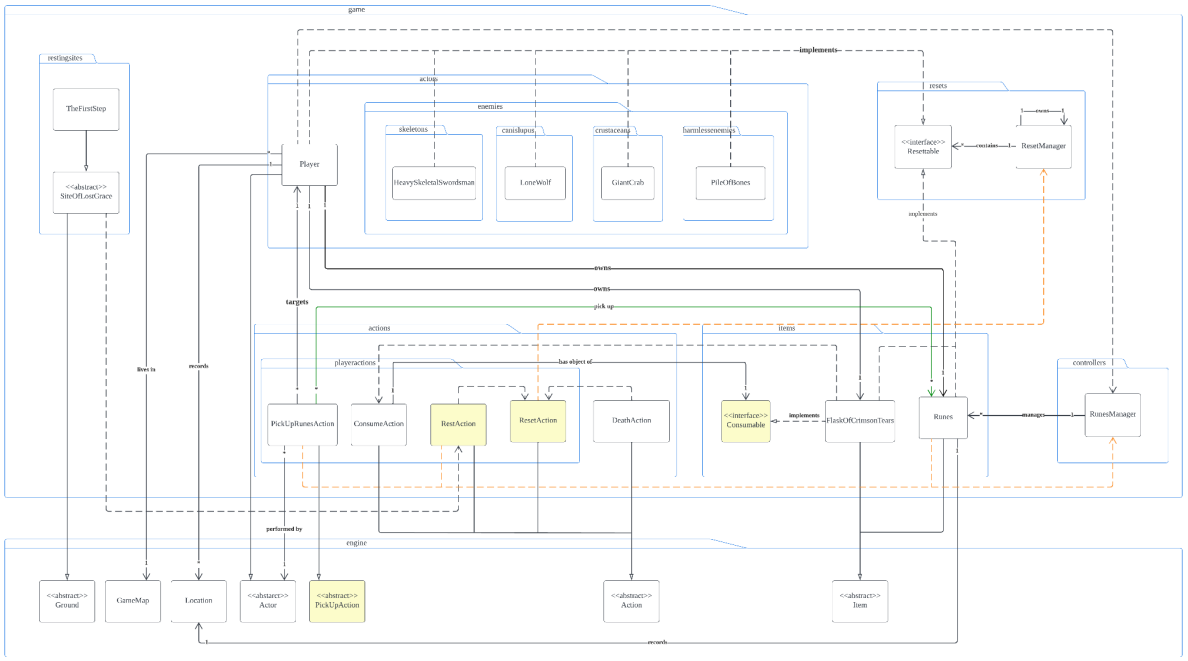
The “Player”, “FlaskOfCrimsonTears” and “Runes” will also be implementing this interface as some unique processes should be carried out or happening during the game reset in their respective *reset* methods, for example resetting the player’s hit point to maximum, resetting the usage amount of Flask 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.

From the specification, it is mentioned that if the world is reset, the dropped weapons will be left on the ground. This could definitely be achieved as weapons are not implemented as “Resettable”. Hence, when the game is reset, we will be going through the implementation of “ResetManager” and just reset the resettable actors, Flask of Crimson Tears and Runes without resetting the weapon items.

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 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, in which the reset logic goes here such as despawning enemies from the game map. Here we make sure the LSP is followed.

Modification to Game Requirement 3 (Assignment 2)



(Please refer to the appendix section for the higher quality of the UML Diagram)

1. Created “RestAction”
    - An instance of “RestAction” will be added into the selectable actions list of the player, so that in the console the player can choose to execute the “RestAction” in order to reset the game voluntarily.
    - Previous implementation: Instance of “ResetAction” will be added into the mentioned list instead of “RestAction”.
    - Con of previous implementation: Breaking SRP principle, as “RestAction” and “ResetAction” should be different in terms of their meaning, although their functionalities are similar to certain extent.
    - Decision: Separate “RestAction” from “ResetAction” to separate their responsibilities
    - OOP principle: SRP principle and OCP principle
    - Pro: While a player entering the “SiteOfLostGrace”, he should be given an option to “rest” but not “reset”, to intuitively separate the responsibility of both actions.
    - Pro: We can print different messages or menu descriptions in the console that differentiate whether the player is voluntarily resting on the “SiteOfLostGrace” or involuntarily dying due to no or negative hitpoints left.
    - Pro: Extra or unique functionalities could be added to both actions easily in the *execute()* method.
    - Pro: Both these actions can have different functionalities, such as in this requirement, “RestAction” will not cause the player to drop runes, but “ResetAction” will cause player to drop runes if he is defeated by enemies.

- Future extension: For example, imagine if the player is resting, he can be given an option to either “reset” the game, heal himself by increasing his hitpoints, or upgrade himself to the next level. So in this case, we can code these functionalities and implement them in the “RestAction” but not in the “ResetAction”, separating their responsibility and making the system more extensible in the future.

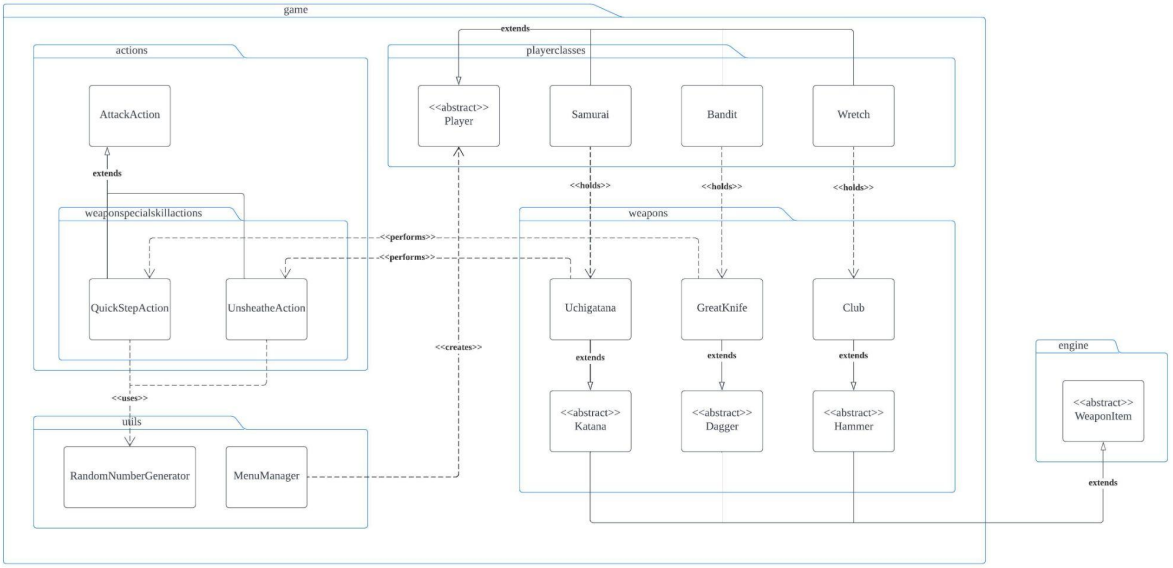
2. Added dependency relationship from “RestAction” to “ResetAction”
    - In this game requirement, it was only mentioned that when the player is resting, then the game will be reset.
    - Decision: Therefore, in this scenario, actually “RestAction” can reuse the same functionality of “ResetAction” when it wants to reset the whole game.
    - OOP principle: DRY principle.
    - Pro: Avoids repetition of code in “RestAction” as we can just reuse the “ResetAction” by calling its *execute()* method when it is time to reset.
    - Pro: Improve code clarity as it can be obviously seen that if the player chooses to rest, then he is actually resetting the whole game like what will be done in “ResetAction”.

3. Added new interface “Consumable”
    - This interface is implemented by “FlaskOfCrimsonTears”.
    - Previous implementation: “FlaskOfCrimsonTears” did not implement any interface, and in every round it will just directly offers “ConsumeAction” to the player if there is still valid usage available.
    - Con of previous implementation: Only “FlaskOfCrimsonTears” is able to offer “ConsumeAction”. This is because previously when the “ConsumeAction” was executed, then it will directly heal the player’s hitpoints by 250.
    - Decision: Let “FlaskOfCrimsonTears” implements “Consumable” interface, with a promised method called *consumedBy()*
    - OOP principle: OCP principle and ISP principle.
    - Pro: We can make every item that can be consumed to implement this interface, so that different effects taken place after consuming different items could be handled in their own *consumedBy()* method
    - Pro: We can have more than one item or other thing in the future that can be consumed and can offer “ConsumeAction” to the player.
    - Pro: The “ConsumeAction” can just call and execute the *consumedBy()* method in any “Consumable” instance to perform the effects or functionalities, instead of just directly performing the effect given by “FlaskOfCrimsonTears”.
    - Future extension: If we have another consumable item, say “PowerPotion”, once drank by player, then he can be shielded and protected from any attacks in three consecutive rounds in the game. In this case, this “PowerPotion” will have to implement “Consumable” and code the implementation of “shielded” inside its own *consumedBy()* method.

4. Extends “PickUpRunesAction” from “PickUpAction”
    - “PickUpRunesAction” is given to the player once he is standing on the ground with dropped runes on it.
    - Previous implementation: Forgot to extend “PickUpRunesAction” from “ PickUpAction”
    - Con of previous implementation: Breaking OCP principle.
    - Decision: Make “PickUpRunesAction” a subclass of “PickUpAction” to inherit its functionalities.
    - OOP principle: OCP principle and SRP principle.
    - Pro: “PickUpRunesAction” should be considered part of “PickUpAction” as they share the similar functionalities, and also so that “PickUpRunesAction” will be automatically added into the list of user’s selectable actions in the “World” class given in the game engine code. This is because “PickUpRunesAction” inherits “PickUpAction”.
    - Pro: Despite sharing commonalities with parent class “PickUpAction”, “PickUpRunesAction” will need to print a unique message to the console, such as “Tarnished retrieves Runes (value: 123)”, which is different from the menu description in the “PickUpAction”.
    - Pro: Extra or unique functionalities could be added to “PickUpRunesAction” easily in the *execute()* method, before or after picking up the runes, making the “PickUpAction” class extensible.
    - Future Extension: If the player receives a bonus such that picking up the runes from specific ground can get the double of its runes value, then we can simply doubling the runes amount and increment the player’s total runes amount by this particular doubled amount in the *execute()* method of “PickUpRunesAction”, after calling the *execute()* method of its super class and before escaping from its own *execute()* method block.

5. The player’s item and weapon inventories will not be cleared once he is defeated by other enemies.
    - Previous implementation: As mentioned in the design rationale of Assignment 1: “We assumed that if the player died, their weapons and items (except runes) will not be dropped but both their weapon and item inventories will be cleared”.
    - We made this modification such that the player's item and weapon inventories will remain unchanged even if he dies simply because we decided to make the game easier so that the player can continue the game with all the items and weapons that he had picked up before he died.
    - Pro: Improve the player experience, so that they won’t feel rage-inducing about the game rules.

Game Requirement 4 (Assignment 1)



The UML diagram above represents the object-oriented system for Game Requirement 4 in which the player is given a choice to choose a character (player class) to play before the game starts. Each player class (“Samurai”, “Bandit” and “Wretch”) 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 player classes 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.

Before the game begins, the player needs to choose a player class. This is where the “MenuManager” class will have a method called *menuitems()* to print the selection 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, “MenuManager” class 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 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 them to be a GOD class and have different behaviours.

Now, since the player must 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 a direct instance of “Player” being created. With this, the design follows the OCP, where concrete classes like “Samurai”, “Bandit” and “Wretch” that should be in the existing system can extend the “Player” parent class and modify any specific functionalities within the concrete class, instead of modifying the “Player” parent class itself. This makes the existing system to be more maintainable and extensible if let’s say there are more player classes that can be introduced in the future, we just create the new player class to be a concrete class, and inherit the “Player” parent class.

According to the requirements, every weapon, for example, “Uchigatana”, “GreatKnife” and “Club”, each has their type, i.e., “Katana”, “Dagger” and “Hammer” respectively. We design to make the “Katana”, “Dagger” and “Hammer” to be an abstract class to prevent a direct instance of any of the classes being created since these are not the real weapon that an actor should hold, but just a type (class) that groups all possible weapons in the game based on their features and similarities. This achieve the OCP, where any specific weapon is created as concrete class and extend the abstract class based on what type of weapon they are, for example, “Uchigatana” extends “Katana”; “GreatKnife” extends “Dagger” and “Club” extends “Hammer”. With this, our existing system is more maintainable and extensible for any new weapons introduced to the game in the future. Same goes to if there’s a new weapon type added into the game, we can create a new abstract class and inherit the “WeaponItem” class.

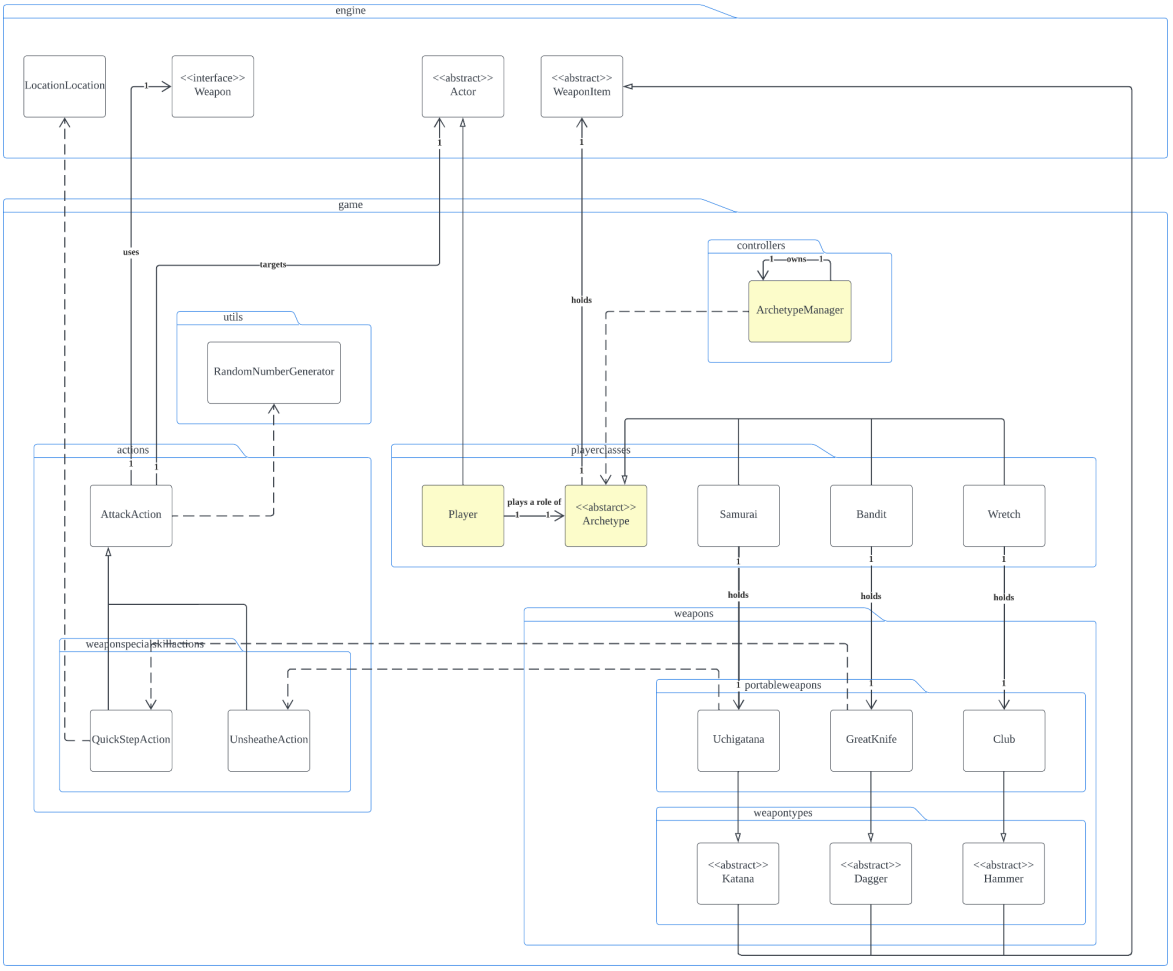
This inheritance concept in the “Player” and “WeaponItem” class that we discussed above potentially reduces repetition codes as usually the parent classes will already contain relevant common attributes or methods that are generic to all of their subclasses, in which we can just call *super()* method 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.

Alternatively, instead of using inheritance concept for all weapon types, our team has discussed using enumeration to check what type of the weapon item should be. This is however impractical as we will have unnecessary repetitive code in every subclass of the same weapon type which causes the system to have low robustness and be hard to maintain, as well as bringing the potentials of making false use of Java's enumeration as it is somewhat not in-line with the main purposes of using enumeration.

Moving on, if the player has the weapons of “Uchigatana” or “GreatKnife” in their weapon inventory, the player can choose to use the weapon and perform a special weapon skill action which are “UnsheatheAction” or “QuickStepAction" respectively. To maintain the SRP, we decided to create a new class for each special weapon skill action since they will have specific responsibilities and functionalities when the action is executed. This will keep the code to be more understandable and keep our system consistent, as well as making the system more maintainable if there is a need to modify the implementation due to changes in game requirements in future.

However, since both unsheathe action and quick step action have the similar behaviour as normal attack action, we design to make “UnsheatheAction” and “QuickStepAction” to inherit “AttackAction” to reduce repetition of code, following DRY principle. 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 “UnsheatheAction”), or moving away from the current position after attacking enemy to evade enemy’s attack (for quick step action). This also achieves the LSP since “UnsheatheAction” and “QuickStepAction” 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.

Modification to Game Requirement 4 (Assignment 2)



(Please refer to the appendix section for the higher quality of the UML Diagram)

1. Created an abstract class of “Archetype” and this abstract class will be extended by every specialized archetype class
    - Extended by every specialized archetype class like “Samurai”, “Bandit” and “Wretch”
    - OOP principles: DRY principle and OCP principle
    - Pro: Avoids repetition of code since all the specialized archetype classes share common attribute and methods
    - Pro If there are any common changes to be made to the general implementation of behaviour of these combat archetypes, we could just modify the abstract/parent class without having to modify all those specialized archetype classes.
    - Future extension: If there are more archetypes introduced in the future, our system can be extended easily - create a concrete class of the new archetype and extend it from the “Archetype” parent class.

2. “Player” class should not have a direct generalisation relationship with all the archetypes
    - This is because the archetype should not be an actor in the game, but instead a role that a player can choose to play the game
    - Decision: Remove all the inheritance of archetypes classes from the “Player” class and create a new abstract class “Archetype” that handles all archetypes

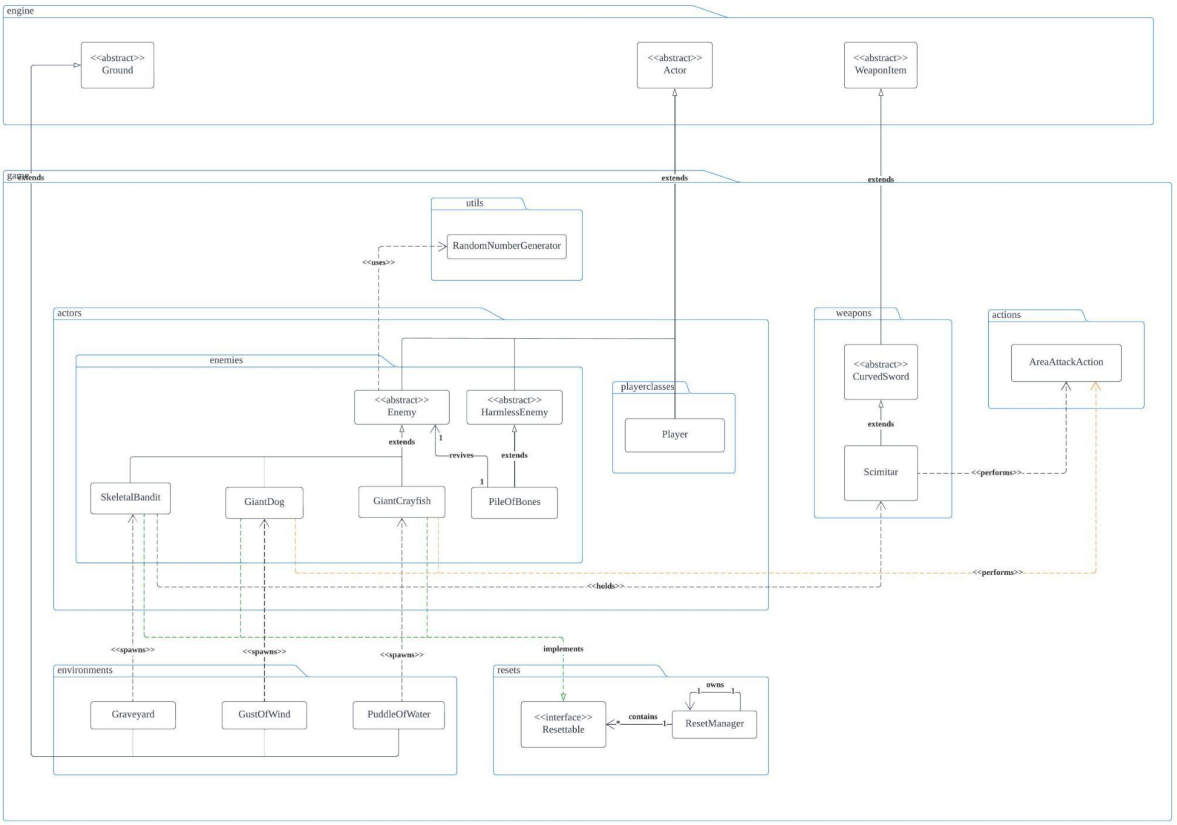
3. More than one player should be expected and each player will be given a choice to choose what archetype they want to play with during the game.
    - Previous implementation: only one player of a specific archetype can exist in the entire game system.
    - Con of previous implementation: It does not allow other actors other than player to share the existing archetypes, breaking OCP principle.
    - Modification: “Player” class should not have a direct generalization relationship with all the archetypes (as explained in First changes (1)) but “Player” class should know what type of archetype (role) that he is holding.
    - Decision: “Player” class should have an association relationship with “Archetype” abstract class.
    - OOP principles: LIS principle and OCP principle
    - Pro: More than one player could be instantiated and their archetype will be determined freely by the user’s selection.
    - Pro: Our game design allows different actors to share the same existing archetypes, which determines their starting weapon and hitpoints.
    - Future extension: More archetypes added in the future will not affect the implementation of the “Player” class - “Player” class can associate with any archetype freely via polymorphism.
    - Future extension: In the future if other enemies or characters added into the game can have different archetypes, our game could easily be maintained and reused without additional changes required.

4. Created “AchetypeManager” class - It allows the user to select the starting class in the console.
    - Previous implementation: Previously we used “MenuManager” class, and it is used to create a specific player with archetype “Samurai”, “Bandit” or “Wretch”.
    - Con of previous implementation: Break OCP principle as “MenuManager” can only create player with specific archetypes, but not other actors.
    - Decision: Now the “ArchetypeManager” class will be used to return an archetype object, and this object is passed into the constructor of any actors (for now only player) to specify its archetype.
    - OOP principles: SRP principle and OCP principle.
    - Pro: Avoids a large block of code in the “Application” class - to assign role to player

- Pro: As explained above, this “ArchetypeManager” is not limited to assigning archetypes to certain actors only, but any possible actors.
- Future extension: If other actors other than player would like to be assigned with one archetype, we can just add one additional line of code above any actor to be created, and then pass in this archetype object into their constructor, so that this particular actor can share the same hitpoints and starting weapon with the other actors of the same archetype.

5. “QuickstepAction” and “UnsheatheAction” extends “AttackAction”, where “AttackAction” has association relationship to “Actor” and “Weapon”
    - Decision: Both “QuickstepAction” and “AreaAttackAction” are just a “special” “AttackAction” so that both classes can reuse the same code in “AttackAction”.
    - For “QuickstepAction”: We just have to move the player to a valid exit in his surroundings in advance before the “AttackAction” is executed.
    - For “UnsheatheAction”: We just have to create a method to double the damage in the “AttackAction” class before it is executed. Call this particular method in the “UnsheatheAction” class constructor.
    - OOP principles: OCP principle and DRY principle.
    - Pro: Avoids code repetition in every “special” attack action class. Instead, we can just perform their special functionalities before calling the *execute()* method of their parent class, which is “AttackAction”.
    - Pro: “AttackAction” can be open for extension as it serves as a base class for every attack action class where all the basic attack functionalities have been implemented here.
    - Future extension: If there is many other “special” attack actions are introduced into the game, these classes can just extend from the “AttackAction” class and we can just code the special functionalities in their specialized classes as no matter what they could just call the parent class’s *execute()* method to reuse the basic attack functionalities.

Game Requirement 5 (Assignment 1)



The UML diagram above represents an object-oriented system for Game Requirement 5 which is a combination of Game Requirements 1, 2 and 3 . The only difference is 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, whereas 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. Besides, in this requirement both the Giant Dog and Giant Crayfish can perform area attacks (which is similar to the Giant Crab in Game Requirement 1), and Skeletal Bandit has the unique ability of turning into Pile of Bones once defeated by the player (which is similar to the Heavy Skeletal Swordsman in Game Requirement 1).

Kindly note that since most of the implementations in Game Requirement 5 are similar to the implementations in Game Requirement 1, 2 and 3, therefore in the case where the design rationale overlaps or has already been covered in the previous Game Requirements, we will not repeat our thought processes here again as to improve the readability of markers and prevent lengthy design rationale.

Similar to the enemies in requirement 1 three enemy subclasses are created which are “SkeletalBandit”, “GiantDog”, and “GiantCrayfish”. This follows the DRY principle which avoids repetition of code and OCP applies where if there are other enemies introduced into the game in the future, only a new enemy subclass needs to be created without modifying the “enemy ” abstract parent class.

To split the game map into east and west half, our team decided to make use of the *location* parameter in the *tick* method of each specific instance of Ground (i.e. Graveyard, Gust of Wind, Puddle of Water) to get the game map of the current game, and then get the width of the game map by calculating the difference between the largest and smallest x-coordinates via the *getXRange()* method provided in the "GameMap" class, as well as the *min()* and *max()* methods provided in the "NumberRange" class of the given engine code.

Then, we can divide the map width into half, in order to get the middle x coordinate which is the median of width. The x-coordinate of the current location can simply be accessed via the *x()* method provided in the "Location" class of the engine code. By comparing the x-coordinate of the current location of each ground to the median of width of the game map, we can simply determine which side of the map a given ground's location belongs to.

If the location's x-coordinate is smaller than the median of width of the game map, it indicates that this specific ground belongs to the West side of the map and then it can potentially spawn Heavy Skeletal Swordsman, Lone Wolf, and Giant Crab (depending on the specific Ground's subclasses). Otherwise if the particular x-coordinate mentioned above is greater than the median, the ground belongs to the East side of the map, potentially spawning Skeletal Bandit, Giant Dog, and Crayfish.

In this approach, no additional classes or methods are needed, we can straight away perform direct checking by modifying the *tick* method of each specific subclass of "Ground" (from requirement 1) which would make the code much more understandable and readable. Also, since each specific instance of "Ground" (i.e. Graveyard, Gust of Wind, Puddle of Water) only takes their own responsibility to spawn its corresponding enemies, we ensure that SRP is obeyed.

Each enemy has a 10% chance of being despawned if they are not following the player. The implementation is similar to requirement 1 where the capability of *FOLLOWING* is used to check whether an enemy is following the player. If the player is not following the player, there is a possibility for them to be despawned from the map. A final static class variable is added to each enemy subclass which is used to initialise their despawn chance as enemy of same type has the same despawn chance and we have made an assumption in requirement 1 that not all the enemy introduced in the future has despawned chance of 10%. The advantages and rationale of using static class variables has been explained in the design rationale of Game Requirement 1 above.

Both Giant Dog and Giant Crayfish have a chance to perform area attack (slam) on all creatures within their surroundings, which is similar to the Giant Crab in requirement 1. Similarly, we can add an instance of "AreaAttackAction" into the list of executable actions of "GiantDog" and "GiantCrayfish", and then use "RandomNumberGenerator" to determine the possibility of "GiantDog" and "GiantCrayfish" executing this area attack action. Our team has assumed that the probability of slam area attack being executed by both "GiantDog" and "GiantCrayfish" is 50% since it is not mentioned in the specification.

In order to check for the type of specific enemies since we are told that some enemies cannot attack their same type (unless certain unique attack is performed) in this Game Requirement 5, we can utilise the similar implementation as explained in the design rationale of Game Requirement 1 above, which is checking the type of enemies via the implicit call to

the *toString()* method provided in the “Actor” class of engine code, as it will show the name (type) of enemies in String format.

Skeletal Bandit has a weapon called Scimitar which is a curved sword type, so a new class “Scimitar” is created and extends the “CurvedSword” abstract class. The advantages and rationale of this implementation has already been described in the design rationale of Game Requirement 1 above.. Since Scimitar can allow the user including Skeletal Bandit to perform an area attack (spinning attack) instead of a normal single attack, therefore in the “Scimitar” class we can directly add an instance of “AreaAttackAction” into the list of the player’s executable actions. Provided that Scimitar can be sold or purchased to/from the Trader, the same implementation of how we treat sellable and purchasable weapon items as described in the design rationale of Game Requirement 2 will be applied here.

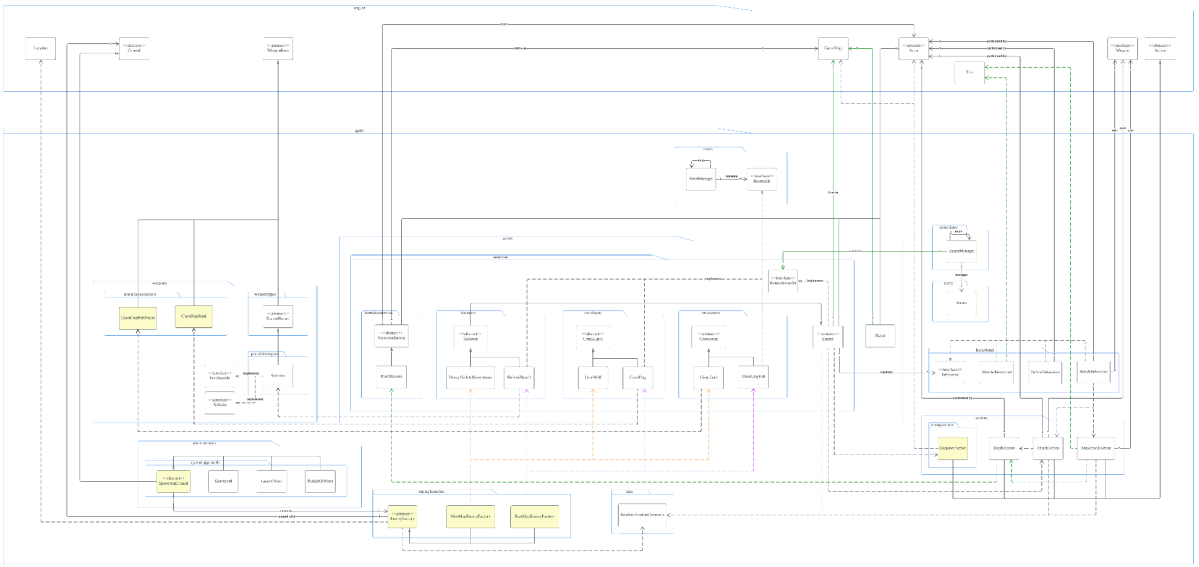
The unique ability of Skeletal Bandit where it can turn into Pile of Bones if being killed, can be fulfilled by using the same implementation of how we treat Heavy Skeletal Swordsman (who has the exact same ability) as mentioned in the design rationale of Game Requirement 1 above.

In this requirement, if the game resets all the enemies will be removed from the map . The implementation is similar to requirement 3 where “SkeletalBandit”, “GiantDog“ and “GiantCrayfish” implement the “Resettable” interface. Similarly, the advantages and rationale of this implementation (letting every specific enemy subclass to implement “Resettable” interface) has been discussed in the design rationale of Game Requirement 3 above.

Furthermore, similar to requirement 2 as the player defeats enemies, they will receive a certain number of runes within a specified range as a reward. Similarly, the entire thought processes and rationale regarding this requirement as discussed by our team has already been covered in the design rationale of Game Requirement 2 above. This is because the whole working principle of how to treat the dropped runes by enemies and how the player can get the runes as reward is exactly the same in the entire game system, and so in the Game Requirement 5, in order to keep the system consistent.

To implement the behaviours of the enemy, similar to requirement 1, our team will be utilising Java's HashMap data structure to prioritise different behaviours among enemies. Similarly, our thought processes regarding this implementation has also been explained in the design rationale of Game Requirement 1 above.

Modification to Game Requirement 5 (Assignment 2)



(Please refer to the appendix section for the higher quality of the UML Diagram)

1. Created new abstract class “EnemyFactory”
    - As the name implies, this abstract class is used to manufacture enemies. In specific, given a location with specialized ground type, “EnemyFactory” will spawn the respective enemy based on the ground type and location like east or west side of the map.
    - Previous implementation: Spawning (manufacturing) of enemy happens in every ground class itself in the *tick()* method where every subclass of ground will have to care about the exact location to spawn an enemy, enemy to be spawned and the spawn chance of an enemy.
    - Con of previous implementation: Having to care about if the ground should spawn an enemy at every turn in *tick()* method makes the ground subclasses to hold multiple responsibilities since spawning an enemy is a different functionality.
    - Decision: Our team decided to create another class that handles the responsibility of spawning (manufacturing) enemies which is called “EnemyFactory”.
    - OOP principles: SRP principle and OCP principle.
    - Pro: By having “EnemyFactory” to hold the responsibility of spawning (manufacturing) can prevent “SpawningGround” to be a GOD class that handles multiple functionality. “EnemyFactory” abstract class becomes more crucial when there are more aspects that need to be taken care of when spawning (manufacturing) an enemy such as whether there's any actor in at the location and to consider about the spawning chance of an enemy at the specific ground.
    - Pro: “SpawningGround” will not have to care about what enemy should be spawned in each subclass. Instead, we will just let the “EnemyFactory” know

the ground type that is currently looping, and “EnemyFactory” will decide which enemy to spawn (manufacture).
- Future extension: If there are more ground type that can spawn enemy is introduced in the future, there’s no need for us to code the functionality of spawning enemy in the *tick()* method in the ground subclass; but to just let the “EnemyFactory” know a new ground type has been introduced and the enemy that the ground type should spawn.

2. Created new classes “EastMapEnemyFactory” and “WestMapEnemyFactory” extending from “EnemyFactory”
    - “EastMapEnemyFactory” and “WestMapEnemyFactory” are created so that the same type of ground on the east side of the map and west side of the map can spawn (manufacture) different enemies.
    - Previous implementation: In the *tick()* method of each ground subclass, the current location will be computed to check if the ground is on the east side or on the west side of the map to spawn the corresponding enemy.
    - Cons of previous implementation: As explained above in the creation of “EnemyFactory” abstract class, this implementation causes the ground subclasses to hold multiple responsibilities.
    - Decision: Since it is required in the requirement that the system should cater for ground type in two sides of the map, our team decided to create the subclasses of “EastMapEnemyFactory” and “WestMapEnemyFactory” that handle the functionality.
    - OOP principles: OCP principle and DRY principle
    - Pro: Each subclass holds the specific responsibility where the same type of ground on the east side of the map and west side of the map will spawn different enemies. This can be implemented in respective subclasses to spawn the corresponding enemy.
    - Pro: Common attributes and methods can be implemented in “EnemyFactory” abstract class and the subclasses can call or use the method directly or via *super*, to avoid repetition of code in every enemy factory subclass. For example, the method *manufactureEnemy()* that checks if a location has an actor and the respective spawn chance before spawning an enemy.
    - Future extension: If the map is further splitted into four sides, let’s say NorthEast, SouthEast, NorthWest and SouthWest to spawn different enemies, the system can be easily extended by creating two more concrete classes extending from “EnemyFactory”. The specific enemies to be spawn at each side will be implemented in the enemy factory subclasses.

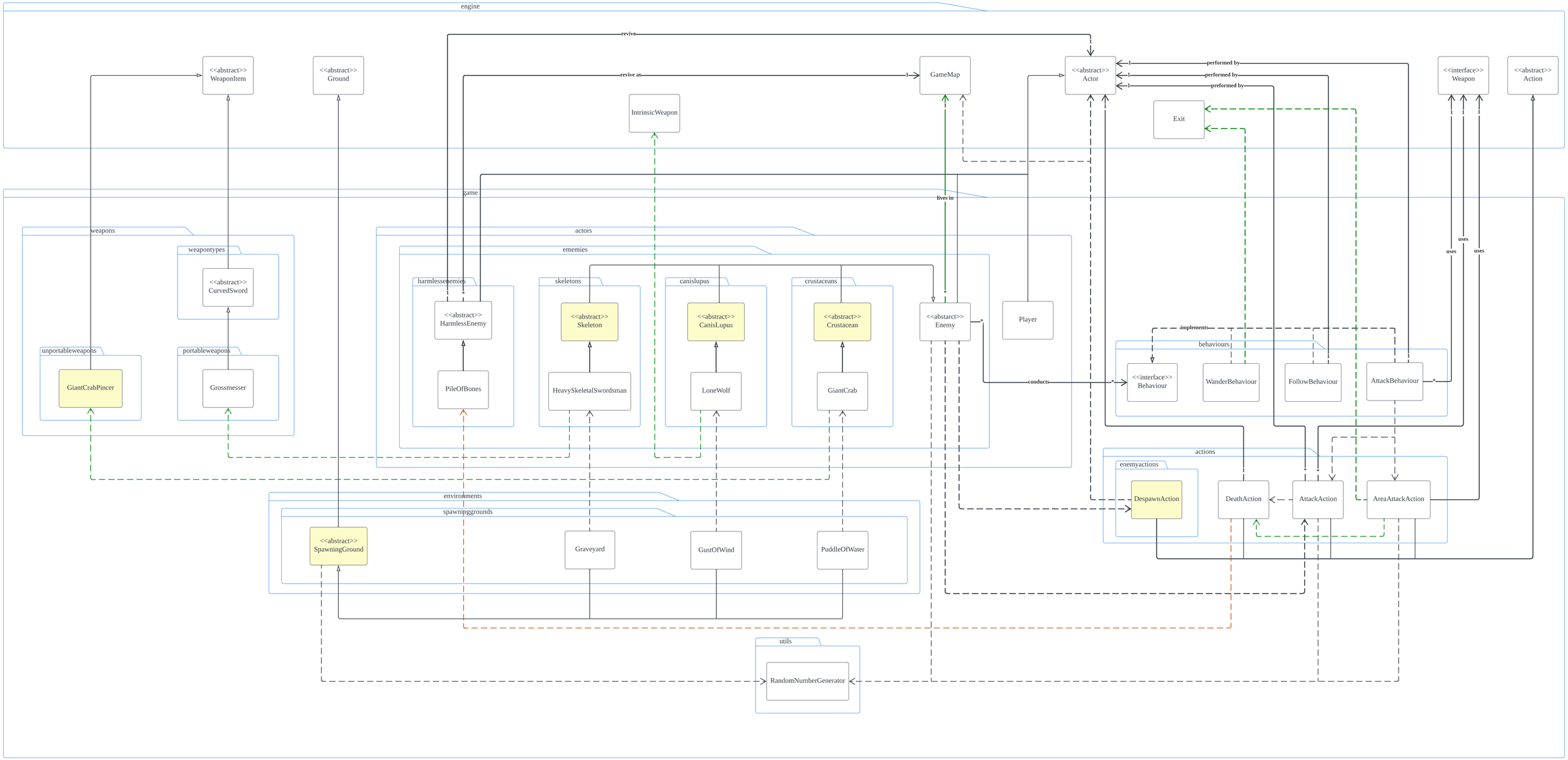
3. Introduced new class “SpawnAction”
    - The design rationale of creating this new class “SpawnAction” has been explained and discussed in Game Requirement 1 above.

4. SpawningGround
    - The design rationale of creating this new class “SpawnAction” has been explained and discussed in Game Requirement 1 above.
    - As mentioned previously in Game Requirement 1, one of the good example of how the creation of “SpawningGround” abstract class helps in making a more extensible game system is shown in this game requirement, where different specialized enemy-spawnable grounds can spawn different enemies depending on the location of the grounds (either it is located at the East or West side of the game map).

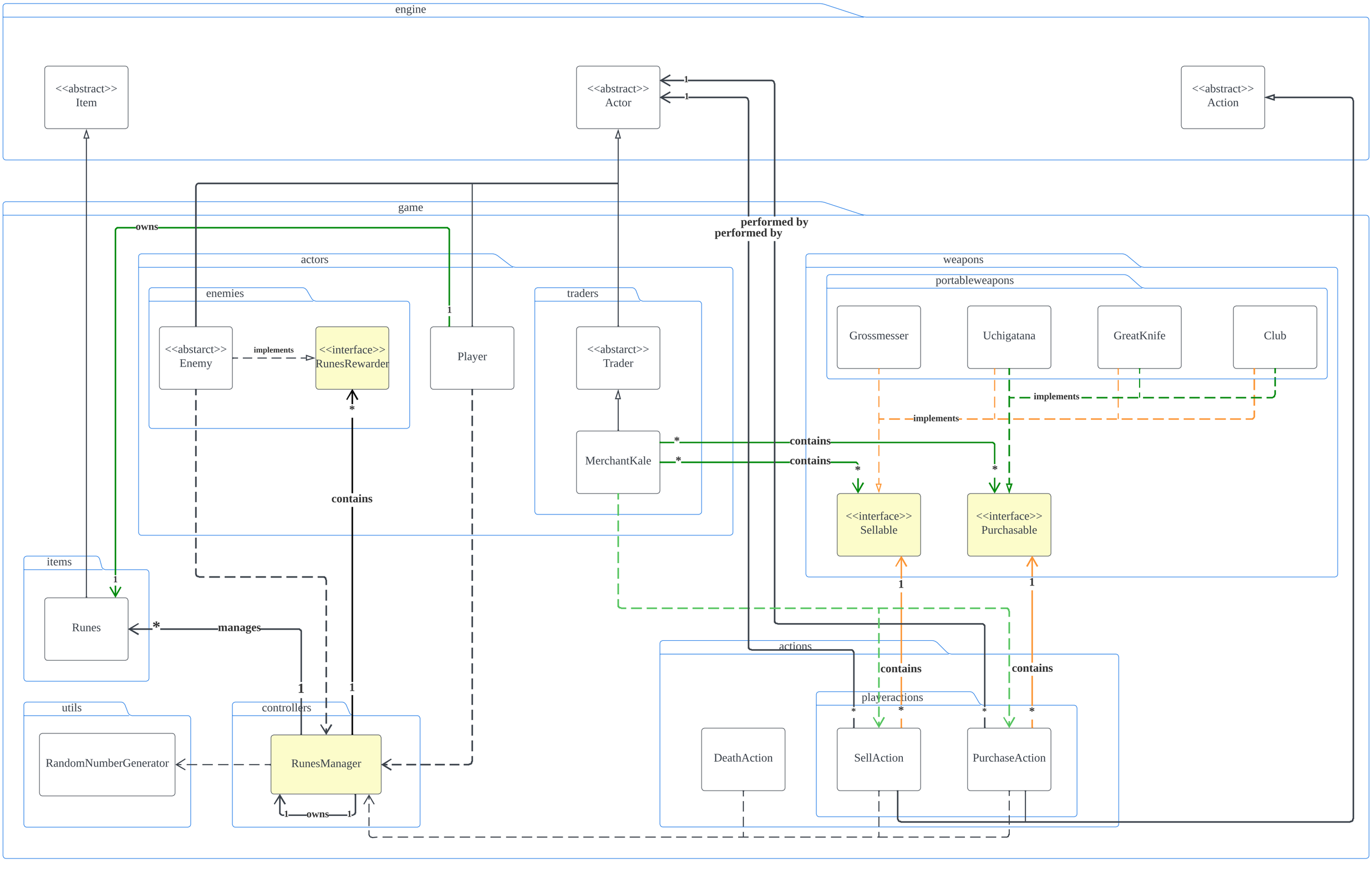
5. “GiantCrayfishPincer” and “GiantDogHead”
    - Similar to the “GiantCrabPincer” in Game Requirement 1, the design rationale of creating these two new classes “GiantCrayfishPincer” and “GiantDogHead” has been explained and discussed previously.
    - To summarize, we treated these two as some kind of “special” weapon items that are only owned by “GiantCrayfish” and “GiantDog” respectively. Their portability is to be toggled and made unportable as discussed previously.
    - This allows both these “special” weapon items to have extra unique functionalities and therefore it is possible for them to have their own specialized subclasses in the future.
    - An example of future extension would be: Upgrade or downgrade of “GiantCrayfishPincer” and “GiantDogHead” to the same weapon items but different in terms of their behaviours, functionalities, or attributes.
    - The design rationale in detail has been covered in Game Requirement 1 above.

6. Created new abstract classes “CanisLupus”, “Crustacean” and “Skeleton”.
    - This Game Requirement also shows a good example on how these abstract classes help for building an extensible game system.
    - As discussed previously in Game Requirement 1, the same type of enemies can share the same characteristics as well as behaviours or attributes, which can be seen in this updated UML diagram where “GiantDog” and “LoneWolf” extends from their type class “CanisLupus”; “GiantCrayfish” and “GiantCrab“ extends from their type class “Crustacean”; whereas “SkeletalBandit” and “HeavySkeletonSwordsman” extends from their type class “Skeleton”.
    - OOP principles: OCP principle and DRY principle.
    - The pros of this updated implementation as well as its possible future extensions have been fully covered in the Game Requirement 1 above.

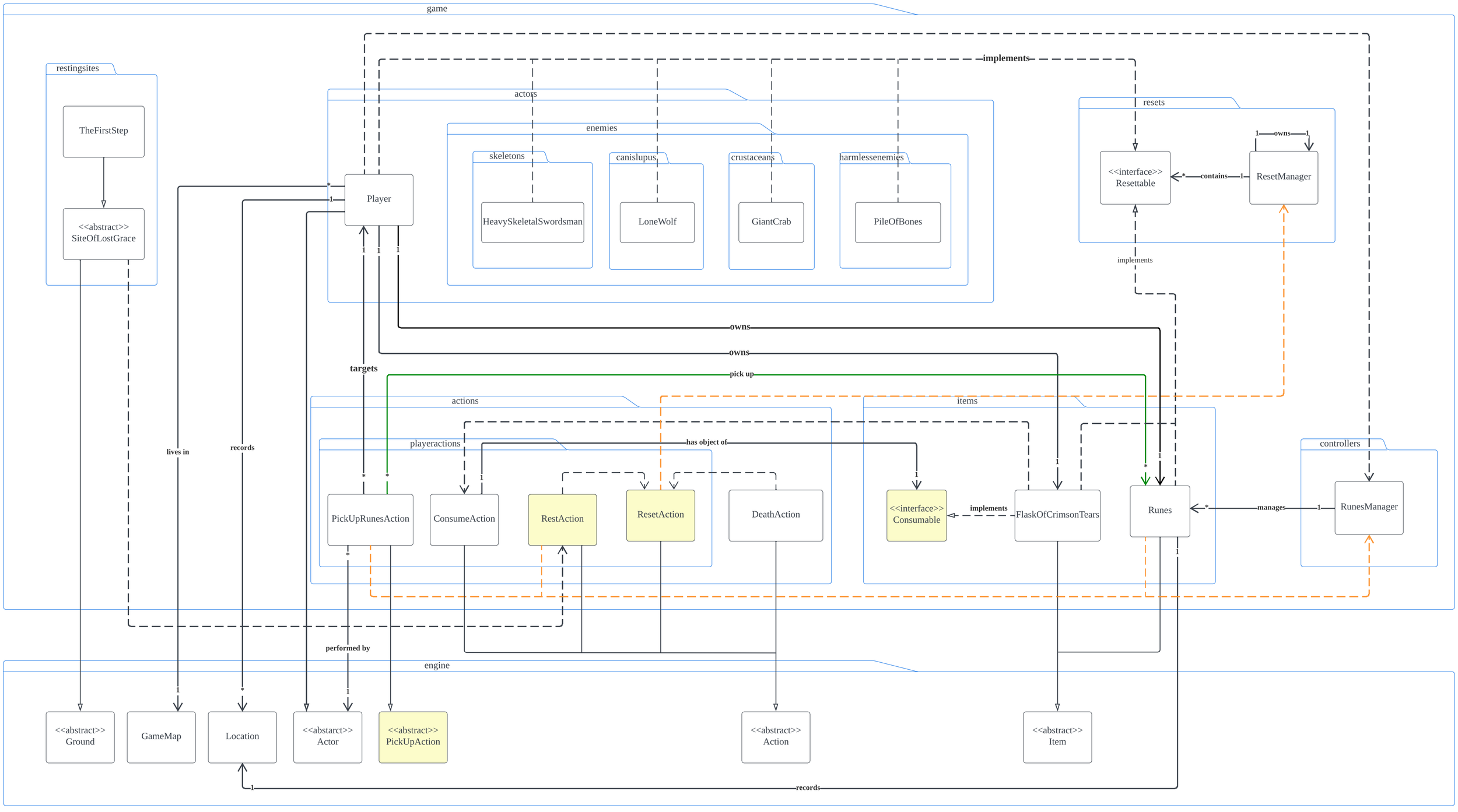
Requirement 1



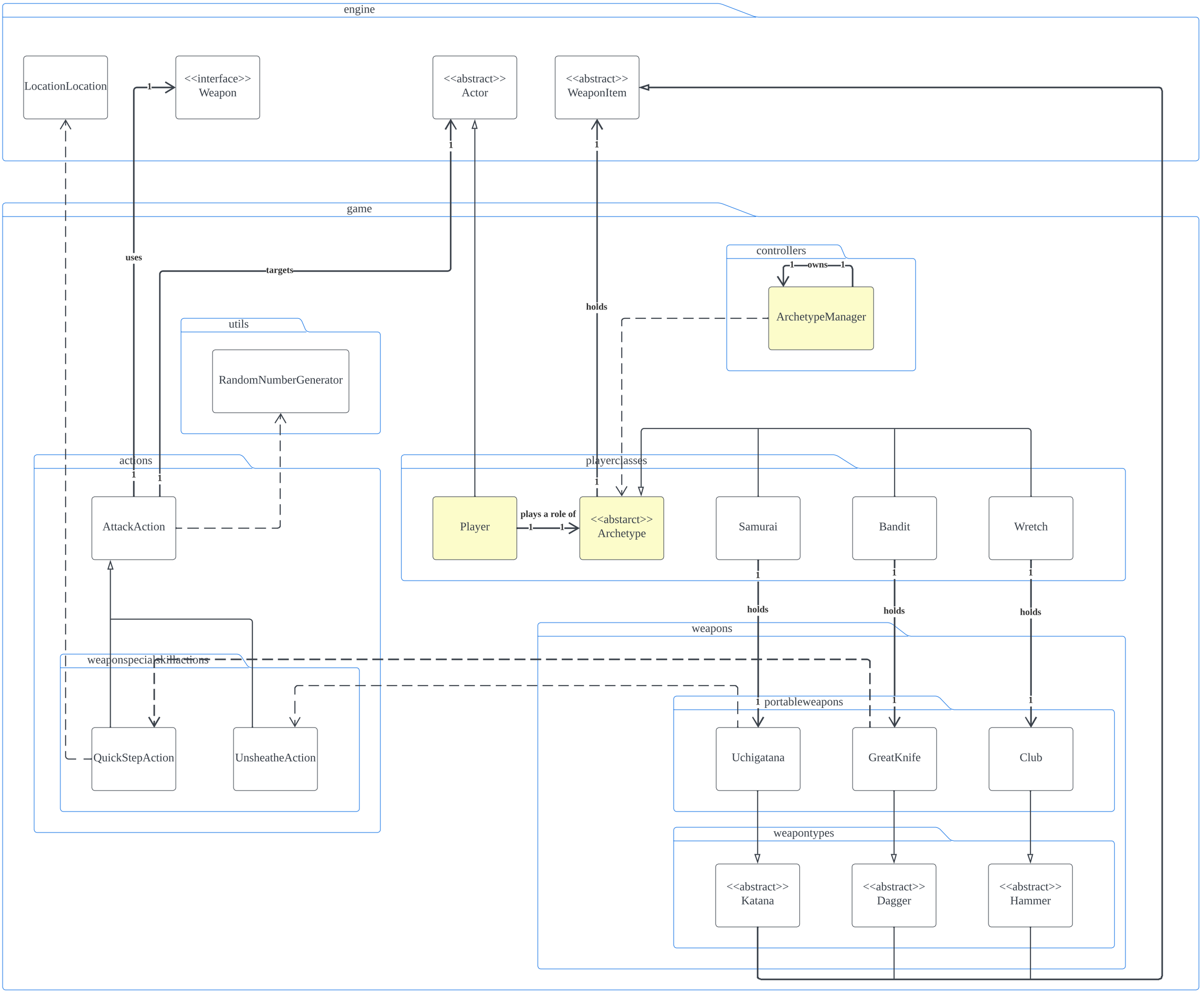
Requirement 2



Requirement 3



Requirement 4



Requirement 5