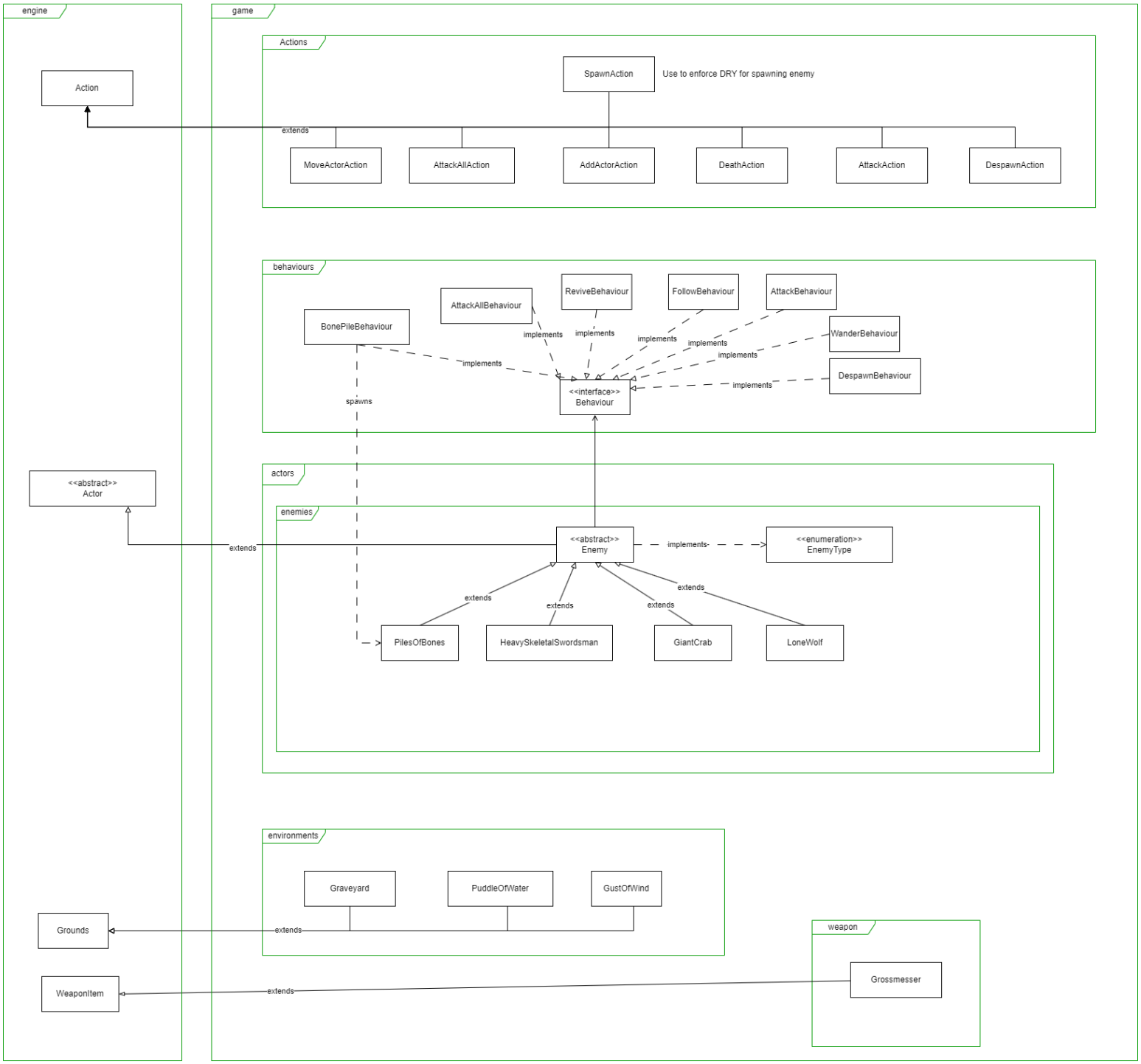
# FIT2099 S1_Lab03Team06: A1 Design Rationale

Contribution log link:
https://docs.google.com/spreadsheets/d/1itMLRX1DOQDyBiHIsicFYc-CcjWs0eaZE00CWoSnUc8/edit#gid=1000079367




## Requirement 1: Environments and Enemies

Environments (Graveyard, PuddleOfWater and GustOfWind) are subclasses of Ground as the Ground abstract class. This is because the environment matches with the description of what a Ground is in documentation: "`Class representing terrain type`", and the Ground class has every method and attribute needed to implement functionality for the environments

Environments have dependency with the Enemy subclass that it is going to spawn (Graveyard spawns HeavySkeletalSwordsman, PuddleOfWater spawns GiantCrab and GustOfWind spawns LoneWolf).

Spawning enemies can be implemented by overriding the tick() method of Ground and using a SpawnAction (a subclass of Action). SpawnAction is executed every turn and uses a random number generator to determine whether or not spawn an enemy. All environment class also depends on SpawnAction

By extending from Ground abstract class and SpawnAction, we are enforcing a DRY design. And SpawnAction class helps make the environment classes comply with Single Responsibility Principle by delegating the responsibility of spawning Enemies to a different class.

Enemies (HeavySkeletalSwordsman, GiantCrab, PileOfBones and LoneWolf) all are subclasses of Enemies abstract class which is a subclass of Actor abstract class. We didn't directly inherit from the Actor abstract class because there is an additional common attribute that all three enemy types needed, which is a Behaviour HashMap. By avoiding having to add the Behaviour HashMap for every enemy type, we are complying with the DRY principle. And by having common attributes for Enemy, it makes interacting with them in code more consistent.

Behaviour that each enemies will have can be instantiated and added to the HashMap (with priorities being an int as the key). As an enemy's behaviour doesn't change over its lifetime, this approach does not violate the Open-close Principle by not allowing for extension.

Subclasses of the Enemy abstract class will depend on the EnemyType enumeration in order to determine what type of enemy they are and not hostile to. For example: HeavySkeletonSwordsman and SkeletonBandit are both type EnemyType.Skeletion so they will not attack each other, but they will attack enemies without the same EnemyType.

Even though PileOfBone don't move, they are a subclass of Enemy instead of Ground because they are temporary, the player must be able to hit it, it must have an inventory to drop upon being hit and it must have a behaviour to revive back to a Skeleton. All of these functionalities are more suited toward an Actor subclass rather than Ground. Doing this enforces Single Responsibility Principle because Ground subclasses are only supposed to manage the terrain's functionalities, not all of the additional functionalities listed above. (More PileOfBones description in REQ 5)

Purpose of each behaviour:

| Behaviour name | Description |
|---|---|
| WanderBehaviour | Like in the given class in the game package, WanderBehaviour.getAction() returns a MoveActorAction to a random adjacent location. If there is no exit in the current location, null is returned.<br><br>It is use to enable an Actor subclass to wander around the GameMap |
| FollowBehaviour | Like in the given class in the game package, FollowBehaviour.getAction() returns a MoveActorAction to the exit that has the closest distance to the Player. If there is no exit in the current location or the player is more than 1 block away, null is returned. |
| AttackAllBehaviour | The AttackAllBehaviour.getAction() method returns an AttackAllAction which attacks all Actors in adjacent locations when conditions for an attack are met, except enemies of the same type are also attacked (See row above). Used to implement Scimitar, Grossmesser's spinning and GiantCrab's slamming attack |
| ReviveBehaviour | Used by PileOfBones, after 3 turns or being called 3 times, ReviveBehaviour.getAction() will return an AddActorAction which will add the skeleton that was defeated back at the location of the PileOfBones. Else it would return null |
| DespawnBehaviour | Used by all enemies, every turn DespawnBehaviour.getAction() will use a random number generator to enforce despawning. 10% of the time, it will return a DespawnAction, removing the enemy from the map. Else it would return null |
| BonePileBehaviour | Used exclusively by the Skeleton enemies, BonePileBehaviour.getAction() will remove the skeleton enemy from the GameMap, and add an instance of PileOfBones at the location of the skeleton. |

Purpose of each Action:

| Action name | Description |
|---|---|

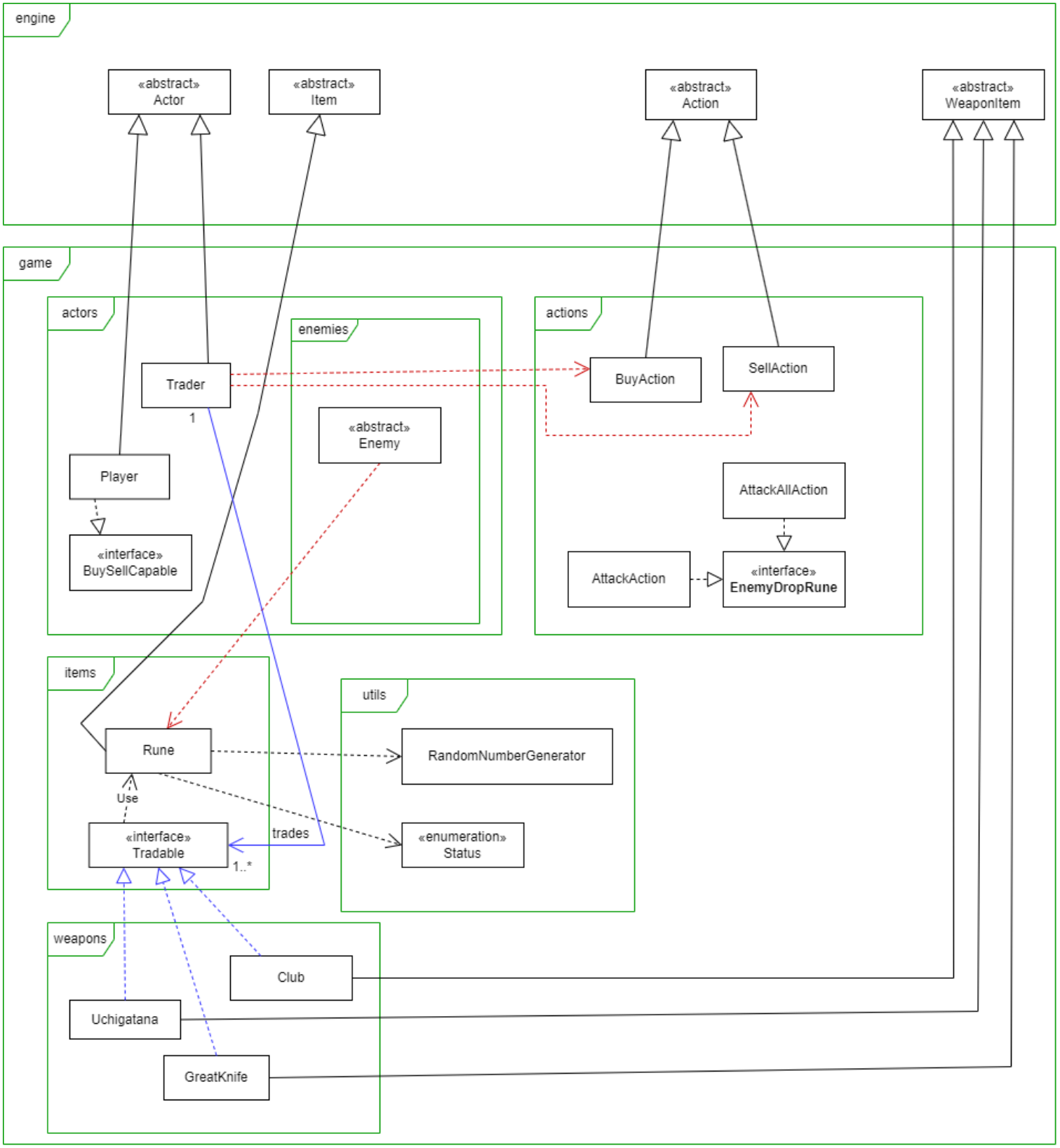
| | |
|---|---|
| SpawnAction | Instantiated with a float representing a chance, an Actor and a Location, SpawnAction.execute() would use a random number generator to add the Actor at the Location |
| MoveActorAction | Like in the given class in the game package, this action moves an Actor to a Location on the GameMap |
| AttackAllAction | Using a weapon, attack all Actors in adjacent Locations to the attacker. All targets is attacked with the same damage and accuracy |
| AddActorAction | Used to add an Actor to the GameMap |
| DeathAction | Used to remove an Actor from the map, dropping their inventory |
| AttackAction | Using a weapon of the attack, inflict damage on to a target Actor |
| DespawnAction | Remove an Actor from the GameMap without dropping their inventory |

All Behaviours and Action adheres to the Single Responsibility Principle as they are only responsible for a singular idea.

All behaviour extending from the Behaviour types helps us store many different classes of behaviour in a Behaviour collection like in the Enemy abstract class. Which adheres to Liskov Substitution Principle.

Higher priority Behaviour can nullify lower priority Behaviour because if a non-null is returned from a Behaviour, it is executed and the turns end. This is used to implement some feature like how an enemy won't despawn if it's following the player, as FollowBehaviour have a higher priority than DespawnBehaviour.

## Requirement 2: Trader and Runes



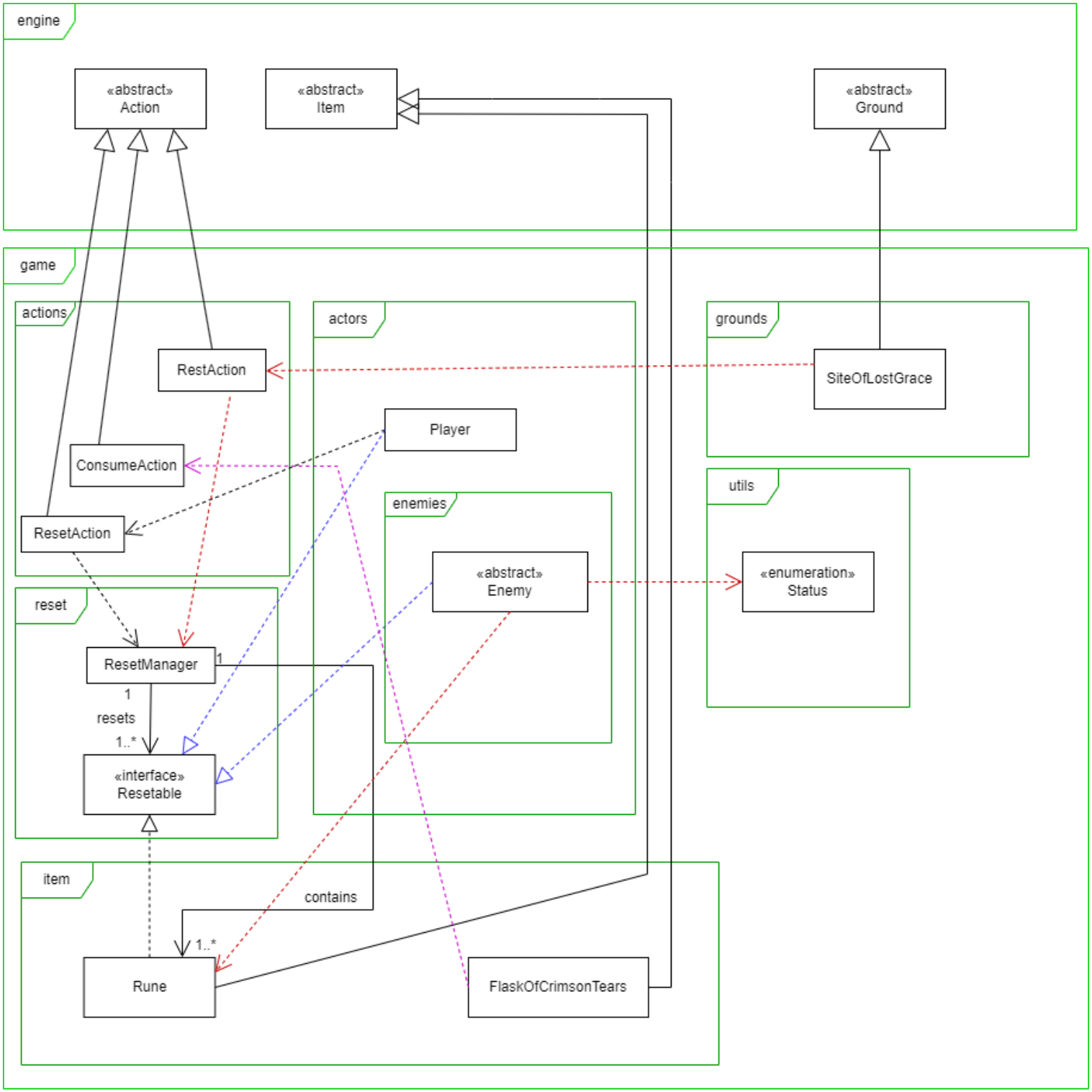
Trader has a dependency on SellAction (sell weapon) and BuyAction (buy weapon) actions. This allows the Trader to allow these actions to be performed on it by the Player. SellAction and BuyAction will be shown as sell weapons and buy weapon options in the menu for the Player when the Player is near the Trader. BuyAction is associated with the RunePurchasable interface so that it can store weapons that can be purchased using Runes. This adheres to the Liskov Substitution Principle (LSP) because only weapons with methods that can be used for Rune purchases are stored and invoked without breaking the program.

Rune inherits from item abstract class in the game engine. They need to be able to be dropped and picked up by the player and enemies should drop them. Item class provides these characteristics and by inheriting it code repetition is reduced adhering DRY. The RunePurchasable interface is implemented by weapons (Uchigatana, Club & GreatKnife) that could be purchased using Runes to give them extra methods to make them purchasable with Runes. This adheres to the Open-Closed Principle because the existing classes are not modified and just extended with features using interfaces. The Rune class depends on Status enum to help other classes determine if the item is a Rune or not. The Rune class depends on RandomNumberGenerator to help give it random values based on different ranges for different enemies. Code repetition is reduced (adhering to DRY principle) by doing so because RandomNumberGenerator provides methods to generate random numbers in a given range.

The AttackAction and SlamAction implements the EnemyDropRune interface. This is required because an enemy that dies after being attacked by an enemy drops the runes to the ground but passes it to the player if the player dies. Since the enemies. Since the enemies could only die from AttackAction and SlamAction we can extend these classes to drop the Runes (determined using Status) correctly based on the Attacker (determined using Status). This adheres to the Open-Closed Principle because the existing classes are not modified and just extended with features using interfaces.

The Player implements the BuySellCapable interface. The Player needs extra code to handle buy and sell actions of weapons with Trader using Runes without affecting its existing functionality. Hence, by implementing the BuySellCapable interface the code in Player does not need to be modified and it just gets added functionality which adheres to the Open-Closed Principle because the existing classes are not modified and just extended with features using interfaces. All the enemies depend on Rune class because they need to add Rune to their inventory so that they can be dropped when they die. The Rune was inherited from the item so that it can make use of existing drop item related methods in actor class to drop runes when enemies die.

## Requirement 3: Grace and Game Reset




Resettable interface is implemented by actors, items etc that should be reset when the game is reset. These resetables are stored in the ResetManager class that contains code to perform reset actions on all the resetables. Only classes that are a subtype of the Resettable interface can be added to Resetmanager class to ensure that only the reset methods of objects that contain them are called. By doing so we can be sure that the objects in ResetManager have functioning reset methods that can

be called without causing errors in the program when the game needs to be reset and this adheres to the Liskov Substitution Principle (LSP).

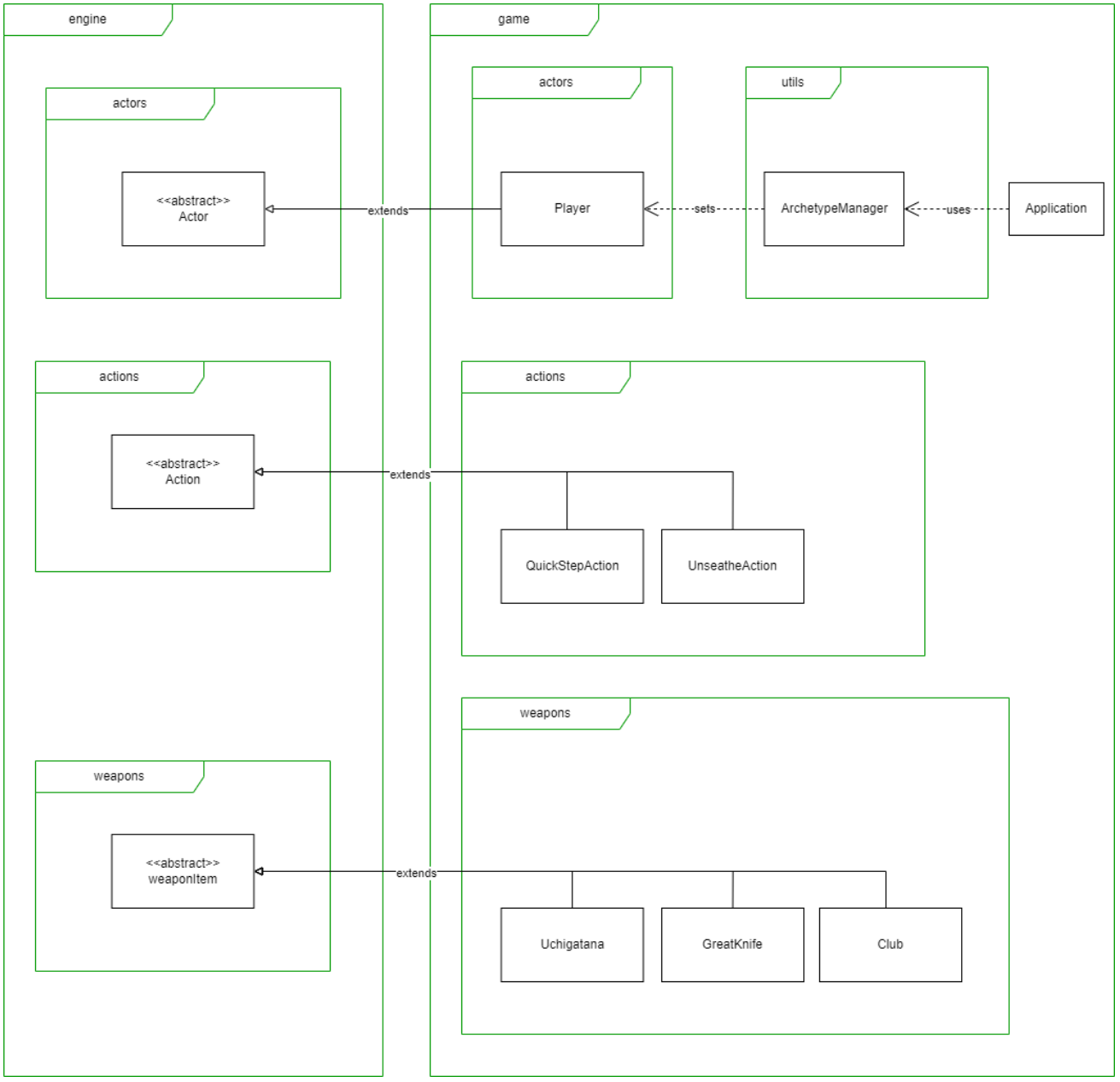
ResetAction in the reset package is inherited from the action abstract class and this action will be returned by the Player if the player gets killed to trigger the reset functionality of the game. ResetAction has a dependency on ResetManager. If a new actor, item and etc are resettable added to the game, code does not have to be modified in ResetAction as these resetables will be managed by ResetManager adhering to the Open-Close Principle.

FlaskOfCrimsonTears inherit from item abstract class in the game engine. FlaskOfCrimsonTears share the same characteristics of an item. SiteOfLostGrace inherits from the ground class in the game engine. SiteOfLostGrace shares the same characteristics of a ground. By inheriting from abstract classes code repetition is reduced adhering to the Do Not Repeat Yourself (DRY) principle. SiteOfLostGrace would have a private constructor with a factory method allowing only 1 instance of the class to exist in the game. This is done because SiteOfLostGrace is unique so having only 1 instance in the game ensures it is not duplicated.

Rune inherits from item abstract class in the game engine. They need to be able to be dropped and picked up by the player and enemies should drop them. Item class provides these characteristics and by inheriting it code repetition is reduced adhering DRY. Rune class implements resettable so that Rune objects can be removed if the player dies again (ResetAction triggered) after they were dropped onto a ground.

When enemies inherit the actor abstract class they can store capabilities in the form of enums. Status enum is used to identify enemies that are spawnable. By doing so the use of instanceof and magic strings that can break the program can be avoided. Enums help ensure consistency when checking for a certain type.

## Requirement 4: Classes (Combat Archetypes)




This diagram represents the implementation of the combat archetypes, a class of character roles that can be selected by the user.

These archetypes determine the starting hitpoint of the user and the starting weapons of the user. So user input would be required to set the user's archetype. Hence, we have a class called ArchetypeManager that directly implements the process of asking for the user to select a number corresponding to the archetype, and from there directly sets the player's HP and weapon. The pros of this is if you want to add a new archetype, then simply add code here. But extensibility is reduced as there isn't an easy function to simply add in a new archetype. Future implementations should try to create an enum variation with methods to add new archetypes.

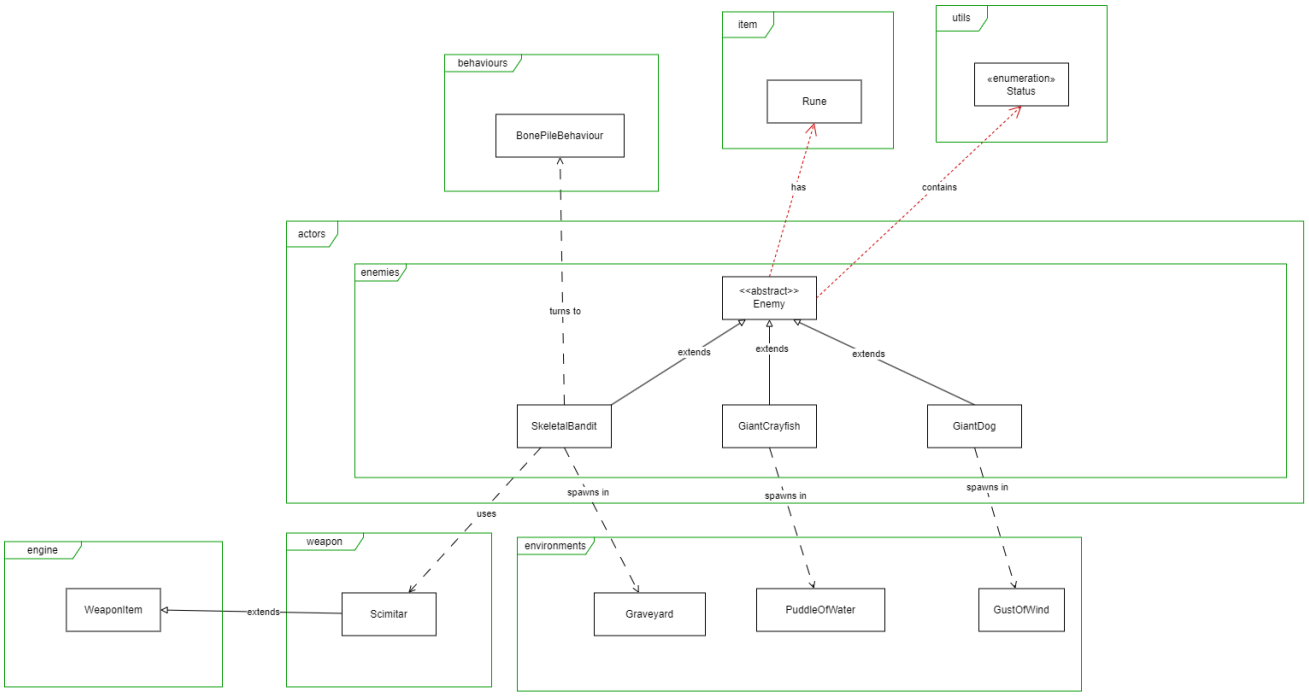
From there, the Application class can call on the ArchetypeManager to run and set up the Player instead of initialising the Player in the Application class. This follows the Single Responsibility Principle (SRP) whereby the Application class should focus on running the main code for the world, instead of also having to determine the Player's archetype.

The user will be directly extended from the abstract class Actor (from the engine) as it has the similar features and attributes. This will reduce repetition of re-coding the attributes and methods (DRY).

The special skills, i.e. QuickStepAction and Unseathe Action, which are associated with the weapons of the unique combat archetype, will be extended from Action (DRY). These skills will be accessible to the player via the getSkill() implementation within the weapon.

The unique weapon associated with each archetypy, i.e. Uchigatana, GreatKnife and Club, will be extended from the abstract Weapon class (DRY), and the special skill associated with these weapons will be implemented.

## Requirement 5: More Enemies




SkeletonBandit extends from the enemy abstract class since it uses all of its attributes and methods. In order to know which type of skeleton to revive, when instantiated PileOfBones will store the instance of the skeleton that was defeated. Then after 3 turns, it will reset the instance and add it back on to the map. The instance will be stored as a Resettable attribute, as all enemy types implement it, this also shows that our design complies with Liskov Substitution Principle.

The Scimitar functionality is almost identical to the Grossmesser, just with different statistics. To represent the spinning attack, the Scimitar.getSkill() will return an AttackAllAction, similar to Grossmesser and GiantCrab's slam attack.

The GiantDog and GiantCrayfish are enemies that extend the enemy abstract class. Since enemies have similarities, using an enemy abstract class and extending from it makes it easy to add functionalities to new enemies adhering to the Open Closed Principle. By doing so they have access to the behaviour attribute because enemy class has association with Behaviour interface. By doing so they can achieve targeted attacks using AttackAction and slam attacks using AttackAllAction. This reduces code repetition because the enemy class will have code to deal with behaviours so behaviours can be added to the enemy inheriting the enemy class to achieve the intended functionality (adhering to DRY). Since the enemy abstract class depends on the Rune class, the enemies inherited from the enemy abstract class will be capable of dropping Runes when they die. The enemies will make use of the Status enum to check for the same enemy type to ensure they do not attack the same type in a targeted attack.