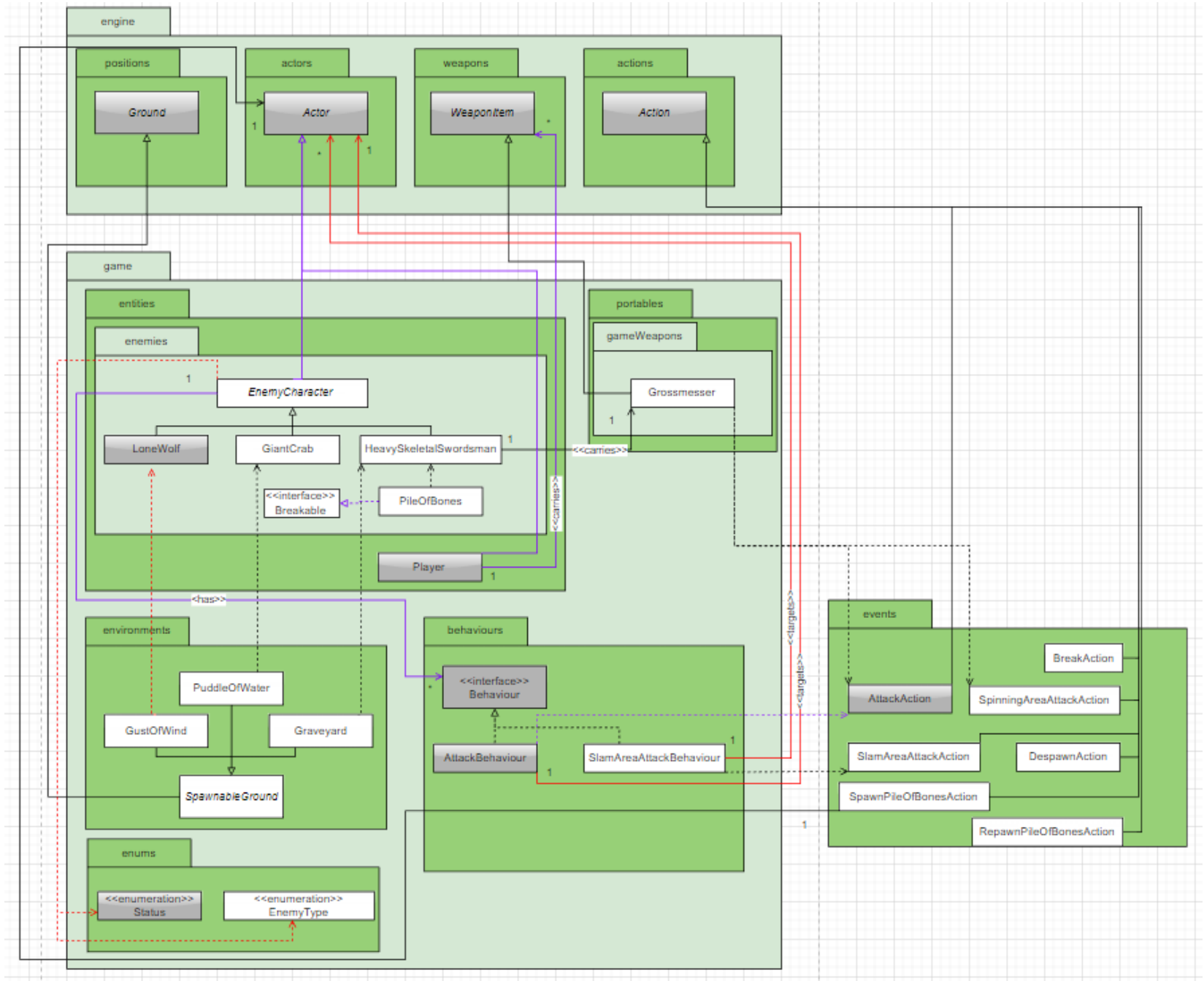
- Improvements/new parts of design rationale



- Improvements/new parts of design rationale

# Req1

**All environment types (Graveyard, GustOfWind and PuddleOfWater) extend SpawnableGround abstract class.**

The SpawnableGround abstract class inherits from the Ground abstract class.

Previously, environment types implemented an interface for spawning enemies but the use of an abstract class is better as it helps handle the enemy spawning process and groups classes with similar functions together (follows our assignment 1 feedback). This class will randomly generate a chance and determine the boolean for whether the enemy can be spawned or not in its tick method which gets called by the child class. This means that each of the child classes do not need to implement this logic and reduces repeated code which follows the 'Don't Repeat Yourself' Principle (DRY) as it is handled by the parent class.

**AttackAction, SlamAreaAttackAction and SpinningAreaAttackAction extend Action.**

All the concrete classes that inherit the abstract Action class have the methods to return written text before (menuDescription) and after the action (execute). SlamAreaAttackAction and SpinningAreaAttackAction are separate classes which carry out the function of attacking all targets around the actor. SlamAreaAttackAction is related to a specific actor (i.e. GiantCrab) while SpinningAreaAttackAction comes from a weapon (i.e. Grossmesser).

Pros:

- We use separate Action classes for the area attacks so that we can distinguish between the attack types (slam and spinning) which will mean that it is a lot easier to make modifications to any of the classes if there are changes to the attack features. This will allow our design to follow the Single Responsibility Principle.

Alternatives and their cons:

- Alternatively, we can combine the classes so that it is one class that allows an actor to attack all surrounding targets. However, doing this will violate the Single Responsibility Principle because the class will be responsible for both the GiantCrab and Grossmesser's area attacks.

Extending functionality:

- If there are any new entities added that need to be able to do slam area attacks, the entities can add the SlamAreaAttackBehaviour to its Behaviour tree map and the entity will be able to carry out slam area attacks.
- If there are any weapons that need to be able to do spinning attacks, the SpinningAreaAttackAction can be added to that weapon.

**All enemy types (LoneWolf, GiantCrab and HeavySkeletalSwordsman) extend EnemyCharacter.**

- Improvements/new parts of design rationale

Instead of using NonPlayableCharacter, we decided to use EnemyCharacter as enemies are a large component of the game and need to be grouped together. They have features that are not implemented by other classes such as using behaviours to determine their actions.

All concrete classes that extend the abstract EnemyCharacter class have a tree map of behaviours (behaviour) and a method that returns an action after going through their list of behaviours in order of priority (playTurn). This is because all enemy types need to hold a list of behaviours and be able to select an action in each game iteration.

Pros:

- EnemyCharacte is an abstract class rather than an interface because interfaces only hold public static final attributes but all enemy characters such as enemies need to have a behaviour tree map attribute which can be updated with Behaviour types.
- In addition, all enemies have the same play turn process so by having the code in the EnemyCharacter abstract class, we are demonstrating code reuse and reducing the amount of code that has to be written again which follows the DRY principle.

**Enemy entities and Behaviour interface.**

All enemy entities will have an association relationship with the Behaviour interface and it will be 1 to many because each enemy type can have more than one Behaviour e.g. WanderBehaviour, FollowBehaviour.

Pros:

- This allows the design to follow the Dependency Inversion Principle because the enemy entities will depend on the Behaviour abstraction rather than a concrete class. This also encourages decoupling, making it easier to change classes without affecting other classes.

Alternatives and their cons:

- An alternative would be the enemy entities each having an attribute for each behaviour they have e.g. an attribute of type WanderBehaviour. This is not ideal as it means that it will violate the Dependency Inversion Principle as it will be depending on a concrete class. In addition, if the classes that implement Behaviour were modified, it would mean that the enemy entity's class would need to be modified. This would go against the Open-closed Principle.

**HeavySkeletalSwordsman becomes PileOfBones when killed.**

To implement this feature, the AttackAction 'execute' method will be modified so that when it checks if the target is unconscious, it will also check if it has the BECOME_PILE_OF_BONES capability. If it does, then it will create and execute a SpawnPileOfBonesAction to add a PileOfBones to the location after the HeavySkeletalSwordsman has been removed from the map.

Pros and extending functionality:

- The use of a capability enum to create and add a PileOfBones to the location allows us to follow the Open-closed Principle. This is because we would be able to add new

- Improvements/new parts of design rationale

enemy types that spawn PileOfBones when killed without modifying our AttackAction class - we would only need to add the BECOME_PILE_OF_BONES capability to the new classes.

- Using an Action class for spawning the pile of bones rather than directly adding a pile of bones to the locations allows us to follow the Single Responsibility Principle as the AttackAction will not need to consider the implementation of adding the pile of bones to the location.

Alternatives and their cons:

- An alternative would be to check the identity of the enemy class before deciding if we need to spawn a PileOfBones however, this would mean that we would need to modify our AttackAction class every time we add a new enemy that is able to become a pile of bones, violating the Open-closed Principle.

**Revive a HeavySkeletalSwordsman from the PileOfBones.**

Previously, we wanted to use a behaviour for spawning the heavy skeletal swordsman from the pile of bones. However, we realised that we only need the action because the pile of bones class itself can keep track of the game ticks/turns using the playTurn method. The PileOfBones will have a counter variable that is incremented each time the playTurn method is called and will check it to see if it is the third turn then use the RespawnPileOfBonesAction if it is. This does create a dependency on the RespawnPileOfBonesAction however, we would have been creating a dependency on SpawnHeavySkeletalSwordmanBehaviour if we were to use the behaviour. This means that we aren't creating any extra dependencies. In addition, we don't use behaviours so we don't need to use the same code from EnemyCharacter to manage behaviours which allows us to follow the DRY Principle.

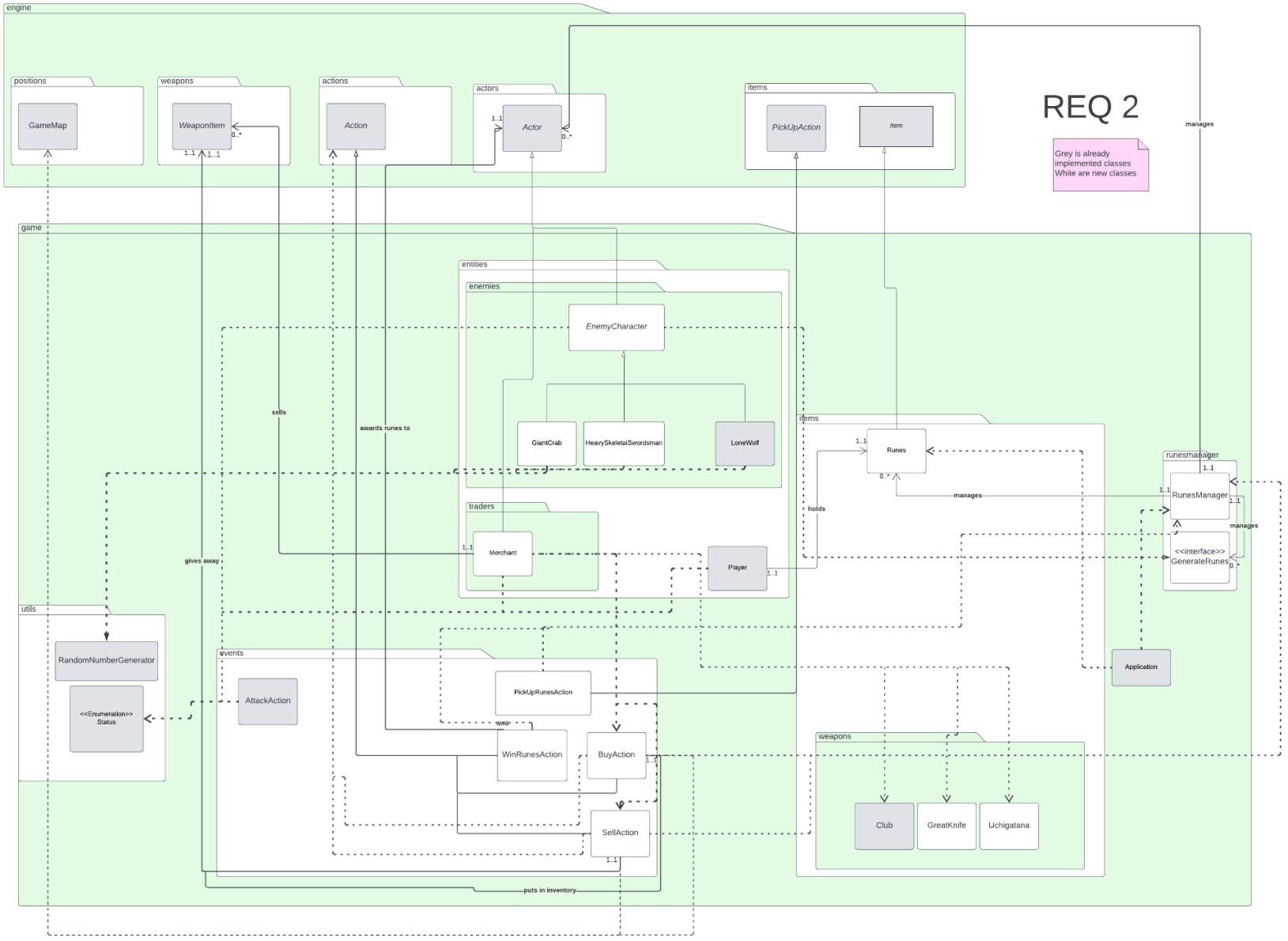
**Randomly despawn enemy entities.**

Following our assignment 1 feedback, despawning does not require behaviours and only needs an action. It has been implemented in the playTurn method in the EnemyCharacter class as it is called once per game turn so it can calculate the chance of despawning and despawn the enemy character by returning a DespawnAction which removes them from the map if needed.

**Enemy types attack each other if they are different.**

We added an EnemyType enum which specifies the type of enemy (canine, crustacean or skeleton). This is more flexible than using status capabilities such as HOSTILE_TO_CANINE and IS_CANINE in AttackBehaviour to check if the actors can attack each other. The latter will violate the open-closed principle if we were to add new enemy types as we would have to modify the method in AttackBehaviour to check for the new enemy type. Instead, using the EnemyType enum and just checking if it's different for two classes before allowing them to attack allows us to follow the Open-closed Principle as we can just add new enemy types to the enum without worrying about AttackBehaviour.

- Improvements/new parts of design rationale



- Improvements/new parts of design rationale

- Improvements/new parts of design rationale

# Req2

**GenerateRunes Interface**
Enemy entities, includingLoneWolf,  GiantCrab and HeavySkeletalSwordsman, hold some shared functionality, such as the random generation of Runes upon death. I propose for a realisation relationship with an interface specifically catered to enemy entities that implement this functionality. To improve on this design, an generic EnemyCharacter abstract class that represents most if not all enemy classes will implement this interface, where its children will complete the contract.

Pros:
- Other Actor classes that interact with enemies can use polymorphism with the abstract interface to recognise similar enemy entities that can create runes. This upholds the Liskov Substitution principle.
- The GenerateRunes interface ensures implementations of behaviours that are required of enemy entities, upholding the Interface Segregation Principle as only classes needing this functionality will implement the interface.

Alternatives and their cons:
- If I placed this functionality in the abstract Actor class, this would provide common functionality to all child classes, even to those that are not meant to access it. This violates the Open-closed principle as it modifies existing code, as well as violates assignment specifics.

Extending functionality
- For example, Requirement 5 introduces new enemies including Skeletal Bandit, Giant Dog and Giant Crayfish. These new entities can be called to generate runes by other classes through the interface. This upholds the dependency inversion principle by not letting the Player depend on low-level concrete classes.
- Generating runes may be used in the future for other needs besides a defeat prize. For example, if there are item boxes in the game map which generate random runes when broken.

**RunesManager Class**
I have found that the engine classes utilise classes that are the parent to many game classes, preventing use of specialised functionality. Currency in the game is a newly added feature of the game, hence I have decided on an external class RunesManager similar to the ResetManager to manage any entities, items or action that can manipulate the Runes items, including Runes itself. This class will call upon the generateRunes method and any of the methods belonging to Runes.

**Enemy entities**
Enemies would have a dependency relationship with the RandomNumberGenerator class's static methods to generate a random value of runes to drop upon death. To utilise this functionality, the AttackAction will call upon WinRunesAction when the enemy target is dead,

- Improvements/new parts of design rationale

only if the loser has the capability to generate runes and the winner has the capability to receive them.

Pros:

- This ensures the DRY (don't repeat yourself) principle is upheld by utilising the already implemented code to achieve code reusability, instead of implementing the same logic multiple times.

**Runes inherit Item class**

Enemies have the ability to drop all items upon death, including the Runes they hold. It should be capable of existing on the world as an object for a period of time. Runes are able to display their value when in possession of a player. I propose a generalisation relationship, where Runes inherits from the Item class. This is to be first instantiated in Application for the class and added to Player as its inventory attribute. Overriding the getPickUpAction and get DropItemAction would allow the engine to call upon Rune-specific actions.

Pros:

- Uphold the DRY(don't repeat yourself) principle by inheriting shared functionality of representing itself in the game display through Actor.

**MerchantKale**

Will have a generalisation relationship with the Actor class, where the merchant will inherit from Actor. To improve the previous design, MerchantKale will include 2 allowable actions for an actor who holds the CAN_TRADE Status capability, BuyAction and SellAction to get the action of creating a menu/console of each possible inventory weapon the Player currently holds when in contact with the merchant. It also includes dependencies on most of the weapon classes as part of its wares in order to assign prices. These weapons are instantiated in the MerchantKale to per turn ensure unlimited stock. The original TraderAction was split into the buy and sell action due to having buying conflicting with the selling function, which I assume would violate the single responsibility principle. The naming choice was also scrapped for better function extendibility, as there is a possibility of other actors or inanimate Items besides the merchant trader that can sell or buy.

Pros:

- It must be able to display itself with a 'k' value and the floor. The Actor class has the functionality required to utilise the Display, hence the MerchantKale class should extend the Actor abstract class's functionality to maintain the DRY(don't repeat yourself) principle.

Cons

- If further Items are added to its stock, then modifications are needed on TraderAction to incorporate new prices to its menu console, violating Open and Closed principles.
- The merchant continuously instantiates all possible wares to buy per turn, which may affect performance. However, as I make a new sell/buy action per item in inventory as separate turns, I cannot see an efficient way of checking what item is selected and then instantiating only that weapon using an if else or switch case without more code smells.

Alternatives

- Improvements/new parts of design rationale

- Instantiate weapons outside of MerchantKale in Application and add to inventory. This might make Application violate the single responsibility principle, as it would then be responsible for not only starting the game, but also creating inventory for traders.
- Create TraderBehavior or use the BuyAction/SellAction to place prices for buying and selling into. However, there would be no extendability when it comes to additional or different prices if multiple traders are implemented into the game. Thus I left the specific prices creation in the specific merchant class.

**Added capabilities to Status**

Further Status capabilities are created from the introduction of Runes. Enemies should only generate runes upon death to an actor like the Player, who can receive runes as its defeat prize. Thus I have created the CAN_GENERATE_RUNES and CAN_RECIEVE_RUNES status to differentiate between actors without specifying the Player or enemy class. The former is to be implemented into the parent EnemyCharacter abstract class for better code reusability. Player is currently the only actor that can trade with the merchant, but in case for future actors beside Player that may be able to trade, I have also created an CAN_TRADE status for the merchant to differentiate which actors can call upon the BuyAction and SellAction. This capability along with CAN_RECIEVE_RUNES will be in the Runes object.

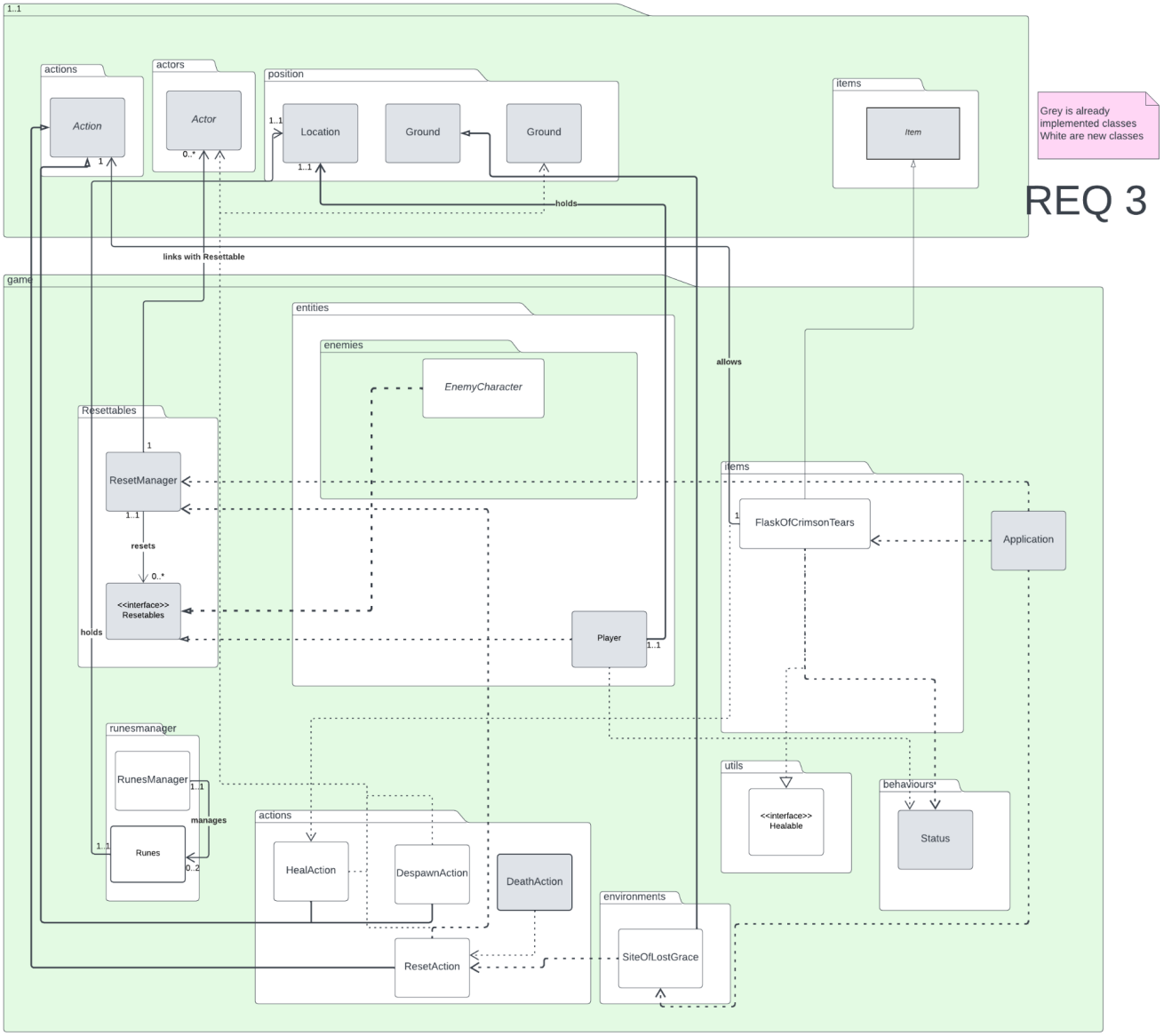
Pros

- As mentioned, this ensures that only the actors that should have this capability can utilise the actions of that capability.

Extending functionality

- These terms can be added to the capabilitySets of future actors as they are generally defined.

- Improvements/new parts of design rationale



- Improvements/new parts of design rationale

# Req3

**Flask of Crimson Tears**
An object that inherits the Item class under the possession of a Player with an generalisation relationship. This is instantiated in the Application and added to inventory. It is not portable. It has the capability CAN_HEAL that is shared with the Player. Through this an healAction can be called to add 250 hitpoints to the actor that holds a flask. This item will implement the Resettable interface to be able to reset its number of uses per iteration of the game.

Pros
- This maintains the single responsibility principle as the instantiation of the healing item is part of the Application's functionality in starting a game, since the Player must begin with the Flask of Crimson Tears in its inventory when the game starts.

Alternatives
- Instantiate Flask of Crimson Tears in Player constructor and place it into Player inventory. This violates the dependency inversion principle because of its concrete dependency to the low-level healing item class. This will also violate the Player's single responsibility principle, since it is also responsible for creating a flask as well as representing a player in the game.

Extending functionality
- The  CAN_HEAL capability can be given to future items or weaponItems that can heal an actor and thus call upon the HealAction. The HealAction utilises any Healable item, further items that implement this interface can use this method. This would help code reusability.

**Site of Lost Grace**
The concrete class extends the Ground class and has a dependency on the Reset action as Reset action will be added to its list of allowable actions. The Reset action will have a dependency on the ResetManager class which will call its run method to iterate through the Resettables and execute their reset methods.

Rather than the player saving the last site of lost grace they have visited as a tuple, it will be saved as a location attribute of the player class in the Application class as it is instantiated there. The player will be able to be moved to the site of lost grace using this location attribute in their reset method.

Pros:
- This follows best practices as the action is attached to the object which will give the Player the option to reset and follows the practices followed by the rest of the game.

- Improvements/new parts of design rationale

- Using the ResetManager to execute the reset methods of the resettables follows the Dependency Inversion Principle as it depends on the abstractions rather than concrete classes.

Alternatives and their cons:

- An alternative would be to check if the ground the player is standing on is the SiteOfLostGrace and if it is, create the reset action as an option for the player to choose. However, if we want to add more ground types that allow the player to reset in the future, we would need to modify our code to check for these ground types. This would violate the Open-closed Principle as it means that we would have to edit our classes rather than extending them.
- Adding the Location attribute to Player does not follow the Reduce Dependencies Principle because we are creating a dependency between Player and Location however, we need a way to save the last site of lost grace that the player has entered so that we are able to follow the game requirements.

Extending functionality:

- The use of the Reset action means that if there are new grounds that allow the player to reset, we can easily add the Reset action to their allowable actions lists. This allows us to easily extend the design and incorporate new environments that allow resetting.

**Runes when the player dies or resets.**

After implementing RunesManager into the game, I've opted to make the class responsible for resetting any Runes object instead of the Runes themselves being resettable.The original boolean design choice for Runes to recognise Player death is still present when being called for the GetDropAction() method. This calls upon the DropRunesAction method.

RunesManager, when reset, will clear its collection of GenerateRunes entities and remove any runes that are already on the ground after Player death. The removal of runes will occur after Player instantiates new Runes to hold its currency, which automatically registers itself into RuneManager. This only occurs when Player is unconscious. Any enemy respawned after the reset will register itself into the manager once more through a dependency injection in its constructor.
This will add a new association with Location to Runes, which must recognise the place where the Player has died.

Pros:

- The design allows the RunesManager class to consider whether the player is alive or dead and to utilise the saved Location to remove the Runes from that location.

Alternatives and their cons:

- An alternative would be to go through each of the locations when the player dies and look for Runes in each of the locations and remove the Runes if there are any. However, this will add dependencies between DeathAction, Location and Runes which increases the coupling. This will mean that when we change one class, we would have to edit the other classes, violating the Open-closed Principle.

- Improvements/new parts of design rationale

**PlayTurn reset**

All resettable Actors will have a realisation relationship with the Resettable interface. The EnemyCharacter abstract class takes in this interface and lets its child enemy classes implement it in its place. Player will implement the reset method directly. To trigger the run() method, the Player must die or choose to rest at a Lost site of Grace during an execution of PlayTurn. A dependency on one ResetManager instance is formed as a result, which calls upon all the resettable Actors' and item reset methods, that calls upon the PlayTurn method to remove or move each Actor accordingly. This instance is used throughout the rest of the game's duration to manage resettable entities in the game.

Pros

- Restricting object creation allows better management of present objects for each iteration of the game with a singular instance of ResetManager.

Alternatives

- Iterating through each Actor managed in GameMap to remove each resettable Actor. This adds dependencies between the ResetManager, Player, Gamemap and Location which GameMap utilises. This can complicate the logic of it's functionality unnecessarily and will not have good code maintainability if any of these classes were to be modified in the future.

Extending functionality

- ResetManager holds a collection of Resettables instead of concrete classes. This makes the addition of new Resettable objects, like other enemies, easy to introduce into the collection without need for modification. These new objects are also obligated to implement the reset method because of its relationship with Resettable, thus code reusability is also applicable.

**Player Death**

The player's death is handled via the DeathAction class which has an execute method that we will edit to suit our purposes. When the player dies, they should not drop their weapons and items and should only drop their runes at their location. This means that we need to add the capability RETAINS_WEAPONS_AND_ITEMS to the player which signifies that the player can keep their weapons and items when they die. In the DeathAction execute method, we can check for this capability and if the actor has that capability, we will drop the runes from their inventory through a DropRunesAction to save player death location , then reset the game using the ResetManager through ResetAction.

Pros and extending functionality:

- This allows our design to be extensible because if we end up adding more Actors that are like players in the future (can retain their weapons and items and drop their runes), we can just give them the RETAINS_WEAPONS_AND_ITEMS capability. This will allow us to abide by the Open-closed principle as we won't need to modify the DeathAction class itself.

Alternatives and their cons:

- Improvements/new parts of design rationale

- An alternative would be to check the actor’s type in the DeathAction class to see if it is the player or not and to execute our functionality accordingly. However, this would mean that if we add a new character that does the same thing as the player, we would need to modify our DeathAction code which violates the Open-closed Principle.

- Improvements/new parts of design rationale

# Req4 * this is left for our third team member

**StartingClass Abstract class**

The Player depends on starting archetype classes in order to create the hitpoint value and weapon upon instantiation. This can be done by utilising a general abstract starting archetype class.

Pros

- Uphold the DRY(don't repeat yourself) principle by using shared attributes for all starting class archetypes
- Reduces dependencies when creating them in the Application class, rather than in the Player class upon instantiation

Alternatives and their cons:

- Modify the application class to hard code the specific values(int and weapon) of each archetype depending on input. This will produce code smell if differentiation of user inputs is done through numbers (1,2…), therefore hard to understand if not given proper meaning.

Extending functionality

- If additional starting classes were made in the future, they can inherit from the abstract class. Since Player utilises the general abstract class, rather than the concrete low level classes, there would be no further modification to their code to incorporate additional classes and their attributes.Thus, uphold the dependency inversion principle.

**Special skills of Weapons**

To incorporate the special skills of the Uchigatana and the Great Knife, override the getSkill default methods that were implemented in the Weapon interface. GreatKnife holds a Status capability that allows for Player evasion, this action created by the enemy entity through the AllowableActions method in the abstract NonPlayableCharacter much like AttackAction is.

Pros:

- Upholds the Liskov substitution principle. Reference to parent abstract class Item can call up possible special skills of individual weapons.

Alternatives

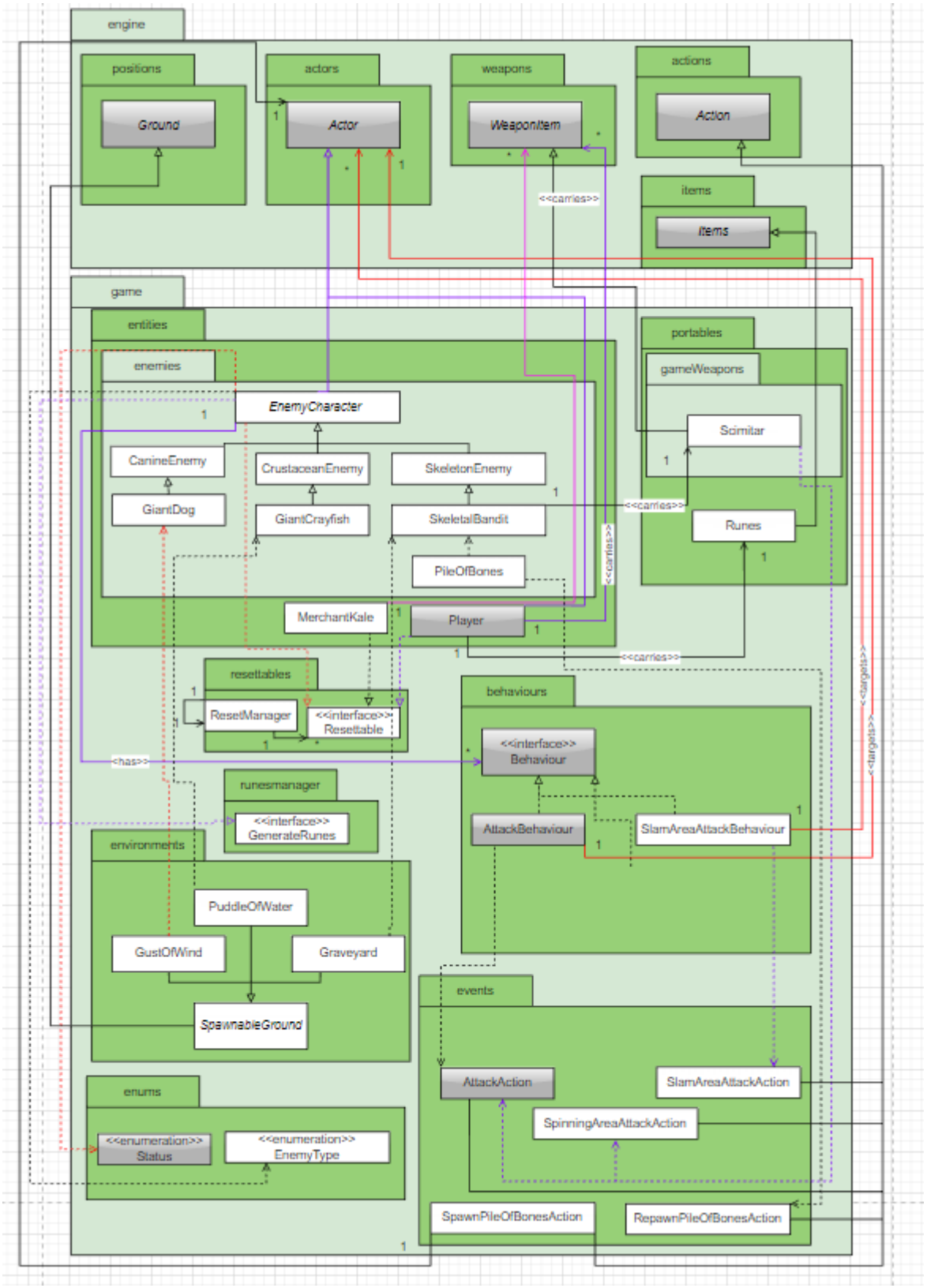
- If add evadeAction in GreatKnife: Requires enemy entities to know Which Actor Player will interact with.There will be additional dependencies to check if Actor has evade capability/status or not, which utilise if-else statements. This results in high coupling.

Extending functionality

- If an additional special skill is added to the Club, the Club class can override the default method in Weapon to incorporate added special skill functionality.

- Improvements/new parts of design rationale

# Req5



- Improvements/new parts of design rationale

**All entities (Player, MerchantKale and enemies) implement the Resettable interface.**
All concrete classes that implement the Resettable interface have a method that carries out the necessary logic when the object is reset e.g. removes enemies from the map.

Pros:
- The use of an interface allows us to follow the Open-Closed Principle as we are able to extend the classes without directly modifying the existing attributes and methods.
- We also follow the Interface Segregation Principle as classes that cannot be reset will not implement this interface so they will not depend on methods that they do not use.

Alternative and their cons:
- An alternative to this would be having if-else statements in the World class that goes through the list of actors and resets them accordingly but this violates the Single Responsibility Principle as the World class should only be responsible for running the game and not resetting it.

Extending functionality:
- If there are any existing or new weapons, items or actors that need to be able to reset in the future, the classes can implement the Resettable interface. This follows the Open-closed Principle as our design is extensible without modifications.

**EnemyCharacter implements the GenerateRunes interface.**
The child classes of EnemyCharacter has a method that returns an integer representing the number of runes that are dropped by an entity.

Pros:
- The use of the interface means that the enemies themselves can generate the number of runes based on their own range. This will allow us to obey the Open-Closed Principle as we are not modifying any existing classes.
- Classes that do not need to generate runes do not implement the GenerateRunes interface which means that we do not go against the Interface Segregation Principles.

Alternatives and their cons:
- An alternative would be to have if-else statements in the Runes class which checks the identity of the enemy before returning the number of Runes that the enemy will drop. This will violate the Open-Closed Principle because it means that if more enemies are added, we will need to modify the Runes class and add more else-if statements for the other enemies. This would also create unnecessary dependencies between the Runes class and other enemy types, resulting in high coupling. This would violate the Reduce Dependency Principle.

Extending functionality:
- If we want to add more enemies in the future, we can make them implement the generateRunes interface so that they can generate a different number of runes based on the kind of enemy they are.

- Improvements/new parts of design rationale

**Enemies extend their respective abstract class for their enemy type (i.e. CanineEnemy, CrustaceanEnemy, SkeletonEnemy).**
As per the assignment 1 feedback, abstract classes were added for each of the enemy types (canines, crustaceans and skeletons) to provide grouping effects so that capabilities for each enemy type can be added in the abstract class rather than being added again in each of the child classes e.g. we only need to add the BECOME_PILE_OF_BONES capability to the SkeletonEnemy class instead of adding it to both the HeavySkeletalSwordsman and SkeletalBandit class. This allows us to reduce the code repeated in the child classes which follows the DRY principle.

**Revive a SkeletalBandit from the PileOfBones.**
When the PileOfBones is created in the SpawnPileOfBonesAction execute method, we will call its setEnemyActor(actor) method to set the original actor or enemy that was there before it became a pile of bones. Then, when the pile of bones has not been attacked after 3 turns, it will use the actor it's saved and pass it to the RespawnPileOfBonesAction. This means that the RespawnPileOfBonesAction can take this actor, heal it then add it to the map. This will allow us to keep a reference to the original actor (heavy skeletal swordsman or pile of bones) before it's killed and turns into a pile of bones rather than creating new instances of the enemy types. This follows the Single Responsibility Principle because the RespawnPileOfBonesAction will handle removing the pile of bones and placing the new actor at the same location instead of the pile of bones handling it itself.

**Randomly spawning enemies based on whether the environment is on the East or West side of the map.**

The abstract SpawnableGround class has methods that check if the location is east or west. The child class can call this method before determining the type of actor they pass to the spawnEnemy(enemyActor) class in SpawnableGround. This follows the Open-closed principle because if we were to consider spawning in other map locations such as north or south in the future, we would only need to add methods to the SpawnEnemy class to check the cardinal direction. Then, in the child classes, we could use these methods to decide the enemy type we would like to pass to the SpawnEnemy spawnEnemy(actor) method e.g. if we wanted to check if it was north-east, we could use the isEast() and isNorth() methods in SpawnableGround and then pass the enemy we want to spawn to the spawnEnemy method.