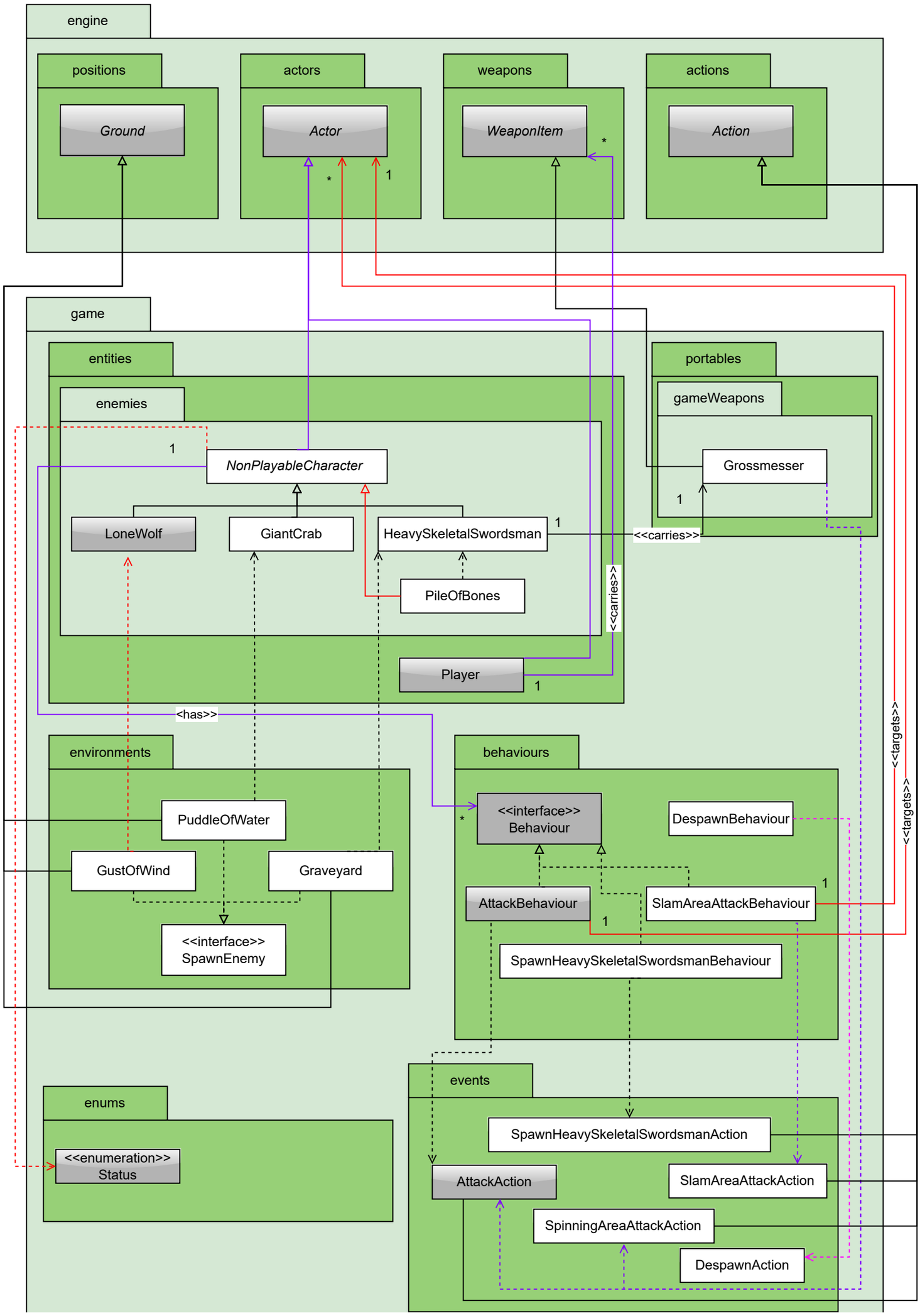

engine
positions
Ground
actors
Actor
weapons
WeaponItem
actions
Action
game
entities
enemies
NonPlayableCharacter
LoneWolf
GiantCrab
HeavySkeletalSwordsman
PileOfBones
Player
portables
gameWeapons
Grossmesser
<<carries>>
<<carries>>
<has>>
environments
PuddleOfWater
GustOfWind
Graveyard
<<interface>>
SpawnEnemy
behaviours
<<interface>>
Behaviour
DespawnBehaviour
AttackBehaviour
SlamAreaAttackBehaviour
SpawnHeavySkeletalSwordsmanBehaviour
<<targets>>
<<targets>>
enums
<<enumeration>>
Status
events
SpawnHeavySkeletalSwordsmanAction
AttackAction
SlamAreaAttackAction
SpinningAreaAttackAction
DespawnAction

Req1

All environment types (Graveyard, GustOfWind and PuddleOfWater) implement SpawnEnemy interface.
All concrete classes that implement the SpawnEnemy interface are able to randomly create a corresponding enemy type (e.g. HeavySkeletalSwordsman, LoneWolf or GiantCrab) and place it at its location.

The environment types currently inherit the abstract Ground class so SpawnEnemy must be an interface rather than an abstract class as the environment classes cannot inherit from two abstract classes.

The environment classes must implement the SpawnEnemy interface because they all have the ability to spawn enemies of specific types so the interface allows them to have a common method which allows different environments to consider different chances of spawning enemies and to spawn different enemy types.

The spawnEnemy method will be called by the tick method of the environment each time.

Pros:

- This follows the DRY principle ('Don't Repeat Yourself') as we know that all the environments will spawn enemies randomly so we use an interface to relate them and provide this common method.
- In addition, this satisfies the Open-closed principle because each of the environment classes aren't being modified and only extended as the SpawnEnemy interface adds the ability of the environment to spawn enemies.
- The Interface Segregation Principle is also being followed as only the classes that need to spawn enemies (the environment classes) implement this interface.

Alternatives and their cons:

- An alternative would be to call the spawnEnemy method for each of the environments in the World class. However, this would violate the Single Responsibility principle as it would mean that the World class wouldn't only be responsible for running the game and getting the Actor and GameMap classes to execute their own logic. The World class would also have to manage the environments that are able to spawn enemies, making it more difficult to maintain the code if any changes had to be made to the environments.

Extending functionality:

- This interface allows the design to easily be extended if there are new environment types in the future that are capable of spawning enemies. The new environment classes would only have to implement the SpawnEnemy interface.

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 NonPlayableCharacter.
All concrete classes that extend the abstract NonPlayableCharacter 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:

- NonPlayableCharacter is an abstract class rather than an interface because interfaces only hold public static final attributes but all non-playable characters such as enemies need to have a behaviour tree map attribute which can be updated with Behaviour types.
- In addition, all non-playable characters have the same play turn process so by having the code in the NonPlayableCharacter 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 a PileOfBones will be added to the map at the location and the original HeavySkeletalSwordsman will be 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 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.

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.
The PileOfBones can have a behaviour class SpawnHeavySkeletalSwordsmanBehaviour that will have a counter variable in it that will be incremented at each game iteration. That behaviour class can then check if it is the third turn by checking the counter and call the SpawnHeavySkeletalSwordsmanAction if it is.

Pros:

- This follows the Single Responsibility Principle as the SpawnHeavySkeletalSwordsmanBehaviour and SpawnHeavySkeletalSwordsmanAction each have their own function of checking if the heavy skeletal swordsman can be be revived and reviving the enemy. This also follows the Open-closed Principle as we are not directly modifying the PileOfBones class and only adding the behaviour class to it.

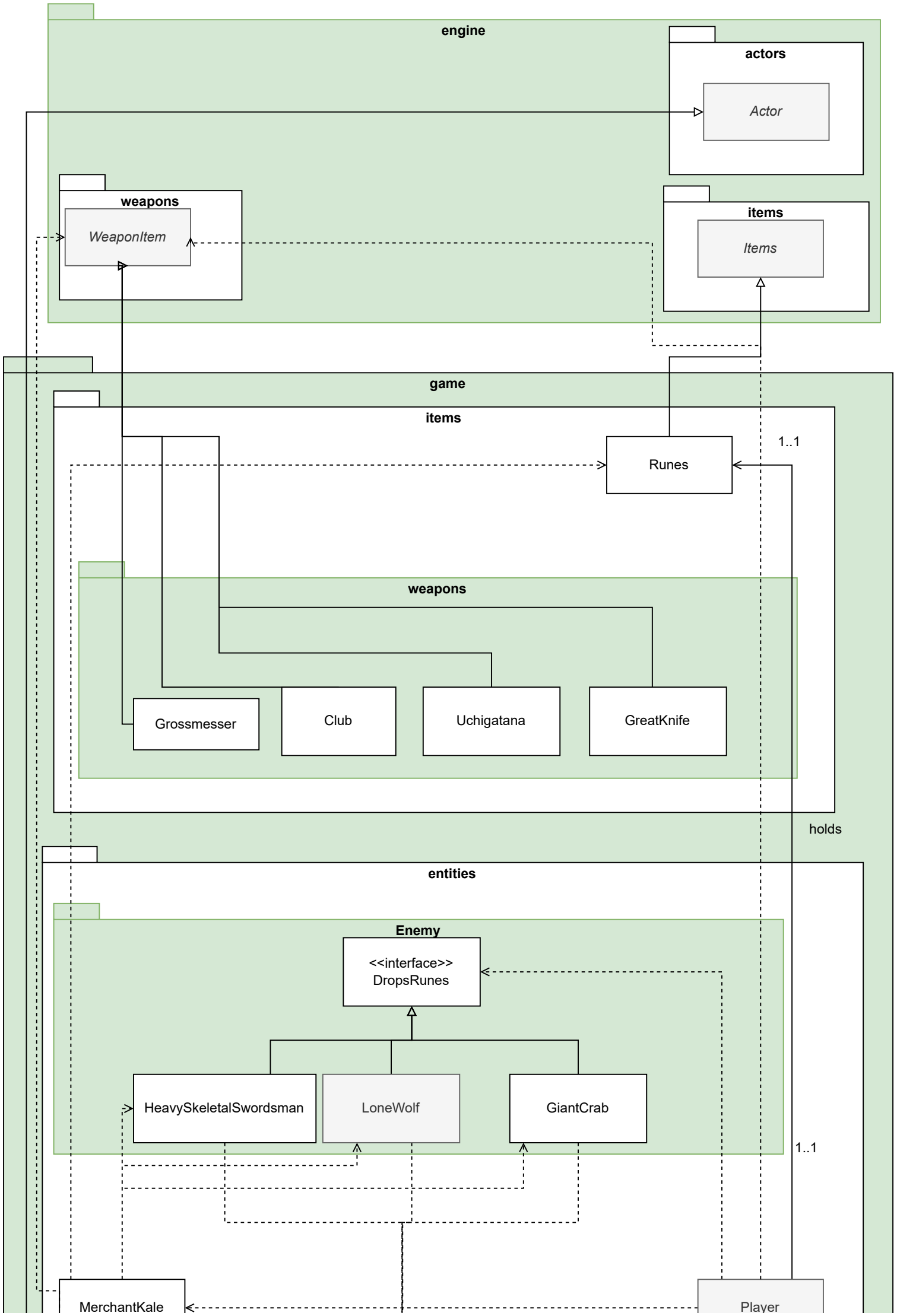
Randomly despawn enemy entities.
The despawn sequence will be implemented through a Behaviour and Action class because the Behaviour class method will be called at each game iteration. This means that the despawn chance can be calculated at each turn and activate the despawn action if necessary.

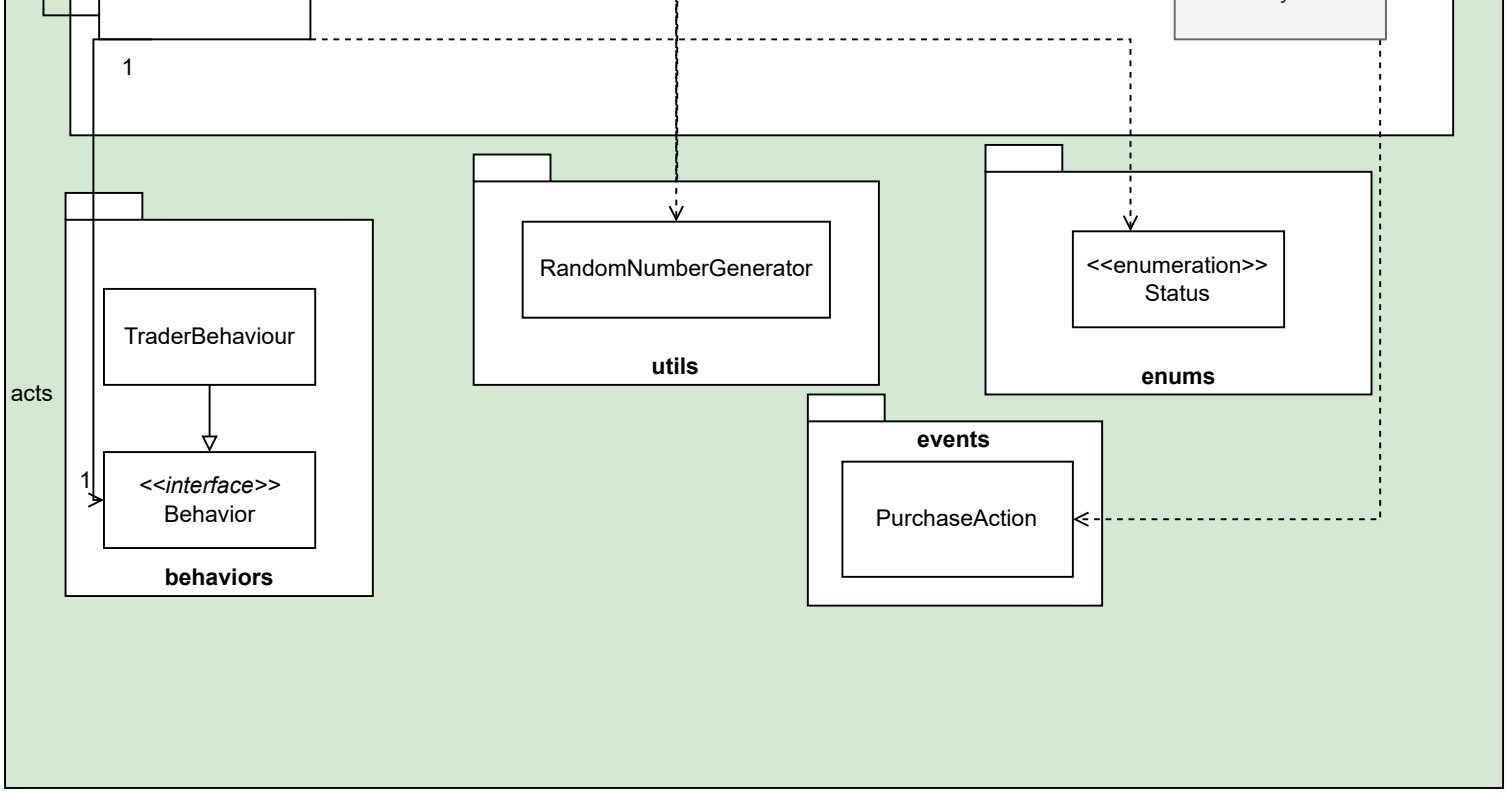
Pros:

- This was the chosen process because it means that we can extend the enemy entity classes without modifying them which follows the Open-closed Principle.
- In addition, there are other classes that will extend from the NonPlayableCharacter class and some of them will not despawn so the logic for handing despawning should not be placed in that class.

Alternative and their cons:

- An alternative method would be to change the NonPlayableCharacter class's playTurn method and remove the enemy from there but this is not appropriate as there are other characters that will inherit from that class and they will not despawn such as MerchantKale. This would also violate the Open-closed Principle because we would be directly changing the class's playTurn method rather than adding functionality to it.


engine
actors
Actor
weapons
WeaponItem
items
Items
game
items
Runes
1..1
weapons
Grossmesser
Club
Uchigatana
GreatKnife
holds
entities
Enemy
<<interface>>
DropsRunes
HeavySkeletalSwordsman
LoneWolf
GiantCrab
1..1
MerchantKale
Player


1
RandomNumberGenerator
utils
<<enumeration>>
Status
enums
acts
TraderBehaviour
1
<<interface>>
Behavior
behaviors
events
PurchaseAction

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.

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. Hence, a general interface can be used..

Enemy entities and the Actor class

GiantCrab and HeavySkeletalSwordsman should inherit from the abstract class NonPlayableCharacter as specified in Requirement 1 rationale with a generalisation relationship. These Enemies are to be instantiated within the Application.

Pros:

- This upholds the DRY (don't repeat yourself) principle by utilising already implemented code instead of creating new methods and attributes to represent an Actor in the game.

Enemy entities and RandomNumberGenerator

Enemies would have a dependency relationship with the RandomNumberGenerator class's static methods to generate a random value of runes to drop upon death.

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.

Cons

- These two classes are tightly coupled as the RandomNumberGenerator is crucial in rune generation functionality

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. This is to ensure that clearly understood and be possibly used in other scenarios

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. An association with the Behaviour class to use TraderBehavior to get the action of creating a menu/console of items to buy for a Player. It also includes dependencies on most of the weapon classes as part of its wares. These weapons are instantiated in the MerchantKale class upon being bought by the player to ensure unlimited stock.

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

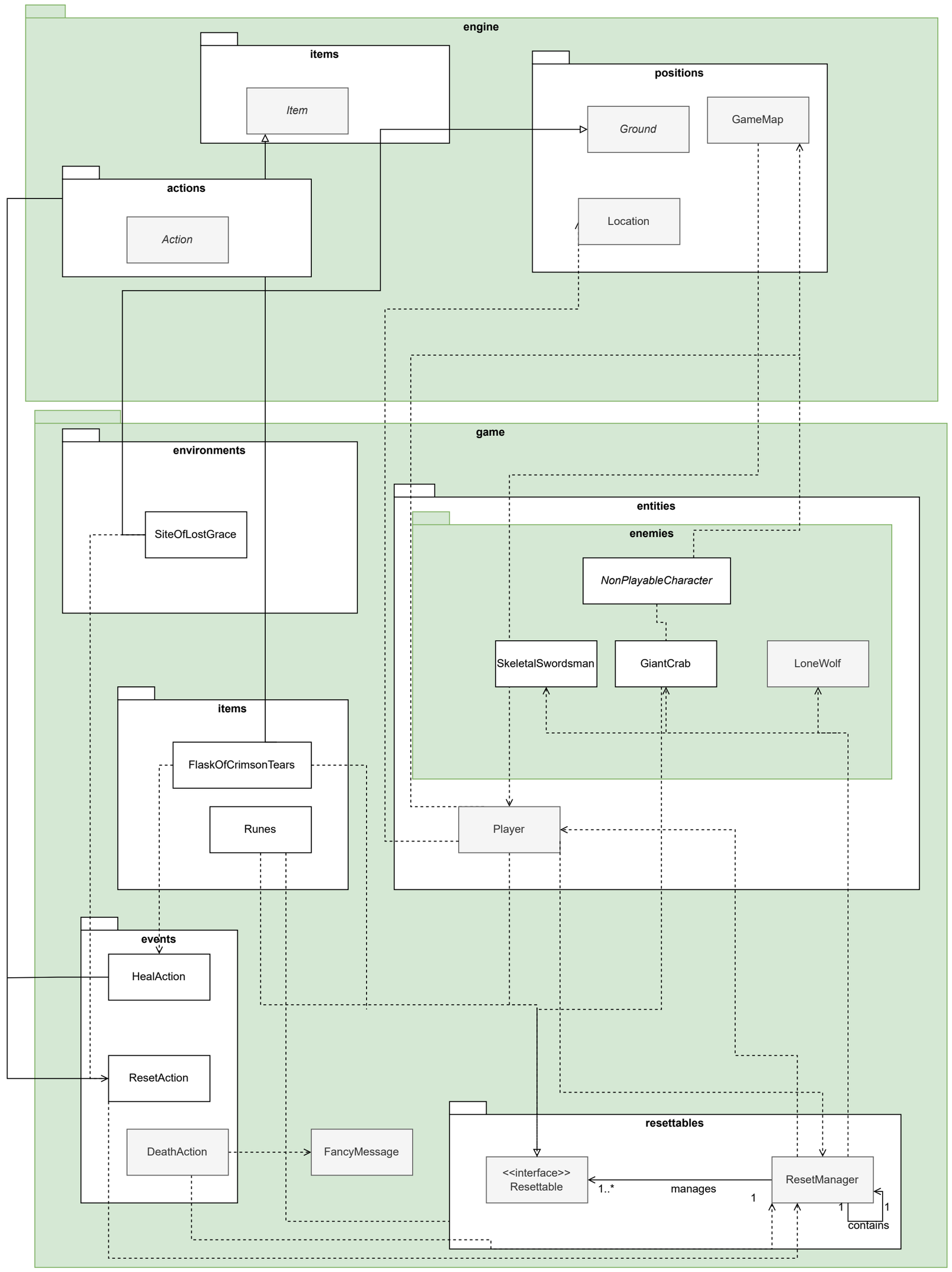
- If further Items are added to its stock, then modifications are needed on TraderAction to incorporate new prices to its menu console, violating Open and Closed principles.

Alternatives

- 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.

Added capabilities to Status

Certain Actors, like merchantKale must be non attackable by other entities. It would be useful to include a permanent status capability similar to IMMORTAL/UNKILLABLE which the merchant can hold to prevent from being attacked by any Actor entity. This can be extended to other NPCs, such as additional traders.


engine
items
Item
positions
Ground
GameMap
Location
actions
Action
game
environments
SiteOfLostGrace
entities
enemies
NonPlayableCharacter
SkeletalSwordsman
GiantCrab
LoneWolf
items
FlaskOfCrimsonTears
Runes
Player
events
HealAction
ResetAction
DeathAction
FancyMessage
resettables
<<interface>>
Resettable
ResetManager
1..*
manages
1
1
1
contains

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 Player. This item will implement the Reseetable 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 single responsibility principle as well as representing a player in the game.

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.

The player has a dependency on Location as it needs to be able to save the location of the last site of lost grace it has visited as a coordinate (tuple) so that they can be moved to that site after they have died.

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.
- 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 coordinate 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.

The ResetManager will have a boolean attribute for the status of the player (whether they are alive or dead) and in the DeathAction execute method, it will set this boolean to false (player is dead) if the player gets killed.

The Runes class will implement the Resettable interface and have a boolean attribute which will signify whether the runes should be removed from the map or not. When the reset method gets called (via the ResetManager class), it will check the ResetManager's boolean to see if the player is alive or dead. If the player is dead, it will set the Runes class's boolean value to true, meaning that the Runes should be removed from the map because the player is dead. If the player is not dead, the Runes class will not do anything. The Runes class's boolean will be used in the Runes class's tick method which will check if the Runes class should be removed and if it should, then it will remove the Runes from the current location.

Pros:

- The design allows the Runes class to consider whether the player is alive or dead and to use the current location and remove the Runes from that location.
- The design follows the Reduce Dependency Principle because it reduces the dependencies between DeathAction, Location and Runes as seen in the alternative below.

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.

PlayTurn reset

All resettable Actors will have a realisation relationship with the Resettable interface. The NonPlayableCharacter 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 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' 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

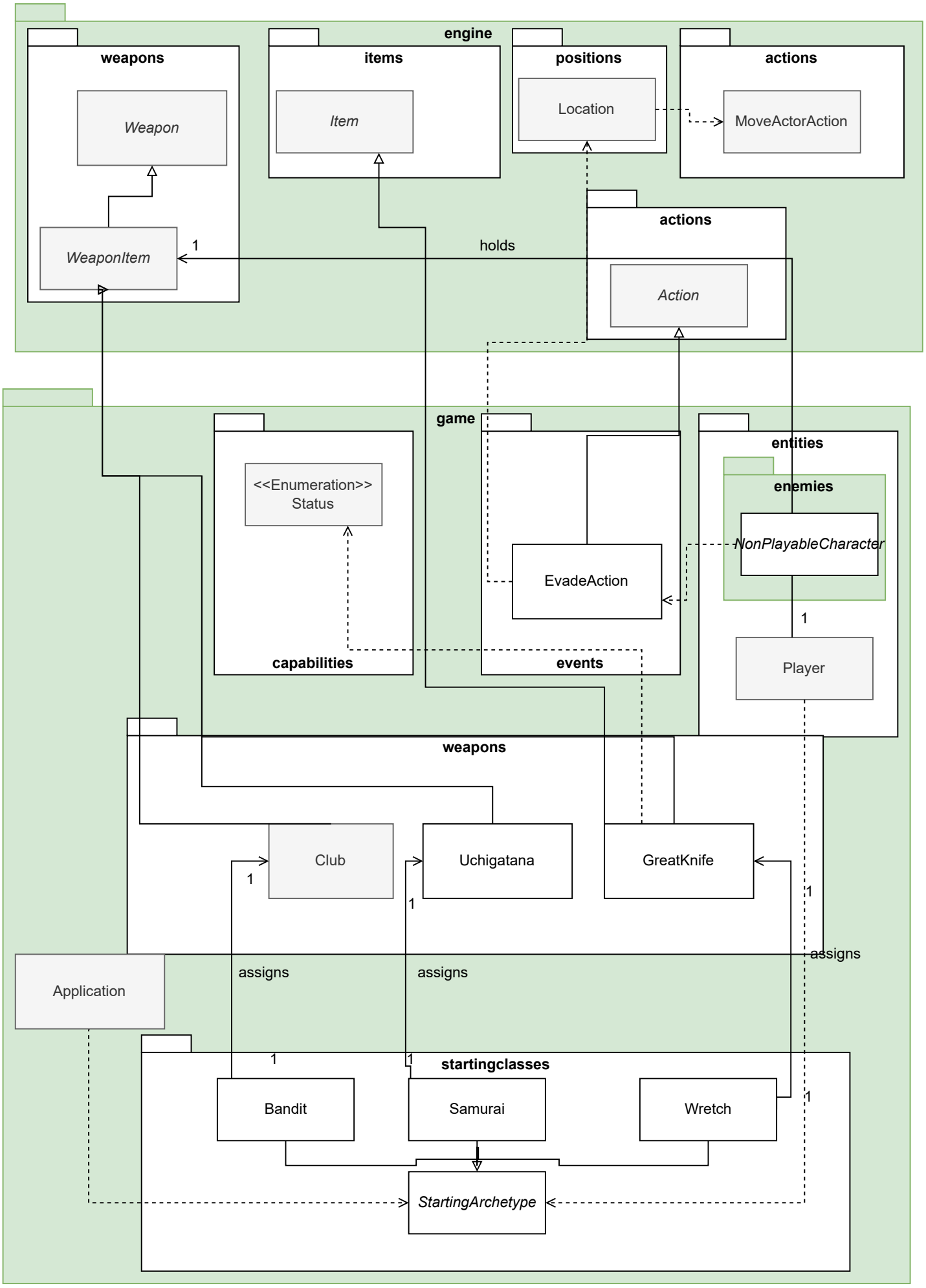
The player's death is handled via the DeathAction class which has an execute method that we will edit to suit our purposes. When the player dies, they should not drop their weapons and items and should only drop their runes at their location. This means that we need to add the capability RETAINS_WEAPONS_AND_ITEMS 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 and reset the game using the ResetManager.

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:

- 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.


engine
weapons
Weapon
WeaponItem
1
items
Item
positions
Location
actions
MoveActorAction
actions
holds
Action
game
<<Enumeration>>
Status
capabilities
EvadeAction
events
entities
enemies
NonPlayableCharacter
1
Player
weapons
Club
1
Uchigatana
1
GreatKnife
1
assigns
assigns
assigns
Application
1
1
startingclasses
Bandit
Samurai
Wretch
1
StartingArchetype

Req4

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:

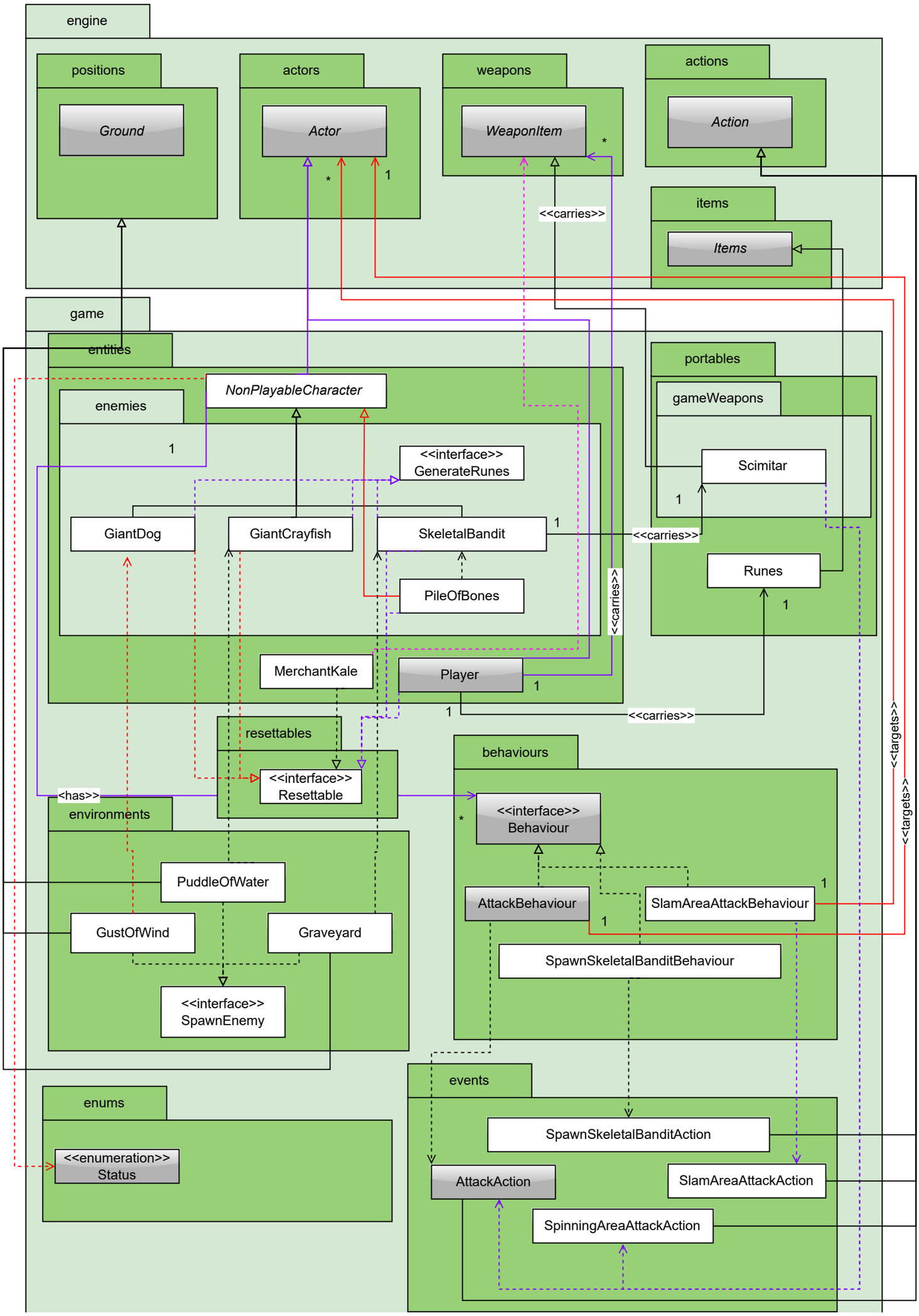
- Upholds the Liskov substitution principle. Reference to parent abstract class Item can call up possible special skills of individual weapons.

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.


engine
positions
Ground
actors
Actor
weapons
WeaponItem
actions
Action
items
Items
<<carries>>
game
entities
NonPlayableCharacter
enemies
<<interface>>
GenerateRunes
GiantDog
GiantCrayfish
SkeletalBandit
PileOfBones
MerchantKale
Player
portables
gameWeapons
Scimitar
Runes
<<carries>>
resettables
<<interface>>
Resettable
<has>>
behaviours
<<interface>>
Behaviour
AttackBehaviour
SlamAreaAttackBehaviour
SpawnSkeletalBanditBehaviour
environments
PuddleOfWater
GustOfWind
Graveyard
<<interface>>
SpawnEnemy
<<targets>>
events
SpawnSkeletalBanditAction
AttackAction
SlamAreaAttackAction
SpinningAreaAttackAction
enums
<<enumeration>>
Status

Req5

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.

All enemies (GiantDog, GiantCrayfish and SkeletalBandit) implement the GenerateRunes interface.
All concrete classes that implement the GenerateRunes interface have 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.

Revive a SkeletalBandit from the PileOfBones.

The PileOfBones will have a SpawnSkeletalBanditBehaviour class in its behaviour tree map and this behaviour class will create an instance of the SpawnSkeletalBanditAction class if the skeletal bandit can be spawned. The behaviour class will check the PileOfBones for the capability CAN_BECOME_SKELETAL_BANDIT to see if the PileOfBones originally came from a skeletal bandit or a heavy skeletal swordsman.

Pros:

- This follows the Single Responsibility principle as we have a behaviour and action class handling this feature which means that each of the classes has one responsibility.

Alternatives and their cons:

- An alternative would be to have just one behaviour class instead of two behaviour classes (SpawnHeavySkeletalSwordsman and SpawnSkeletalBanditBehaviour) which checks if the PileOfBones has the capability CAN_BECOME_HEAVY_SKELETAL_SWORDSMANS or CAN_BECOME_SKELETAL_BANDIT. Then, it can return the corresponding action to spawn either type. However, this would violate the Single Responsibility Principle as the behaviour class will be responsible for checking whether the PileOfBones can spawn a heavy skeletal swordsman or if it can spawn a skeletal bandit.

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

We can handle this feature through capabilities EAST and WEST in each of the locations and check if the environment is on the east or west in the spawnEnemy method before spawning the enemy types.

Pros and extending functionality:

- The use of the capabilities means that if there was a future feature which wants the environments to consider if they're north, south, northeast, etc., we can add more capabilities to handle these different directions and spawn the corresponding enemies.

Contribution log link:
https://docs.google.com/spreadsheets/d/1l8OPx45qtbDYTUqgehJL3m7RJFSrLO7IgVp8Kqd6Lcl/edit?usp=sharing