Justifications can be based on:

- **SOLID principle**
    - **single responsibility principle - each class should solve only one problem.**
    - **open-closed principle - Open for extension, Closed for modification**
    - **Liskov substitution principle**
    - **interface segregation principle - Clients should not be forced to implement interfaces they do not use**
    - **dependency inversion principle**
- **Reduce dependency - a object has to deal with other objects as little as possible to understand**
- **DRY principle (don't repeat yourself) -> use abstract classes, interfaces and public constants**
- **Fail fast -> if an error occurs, it's best to detect it ASAP**
- **Privacy leaks**
    - **it's better to return a copy of an instance**
    - **Changes to the copy won't affect the original**
- **Include example of extending functionality (e.g. new characters can be added)**

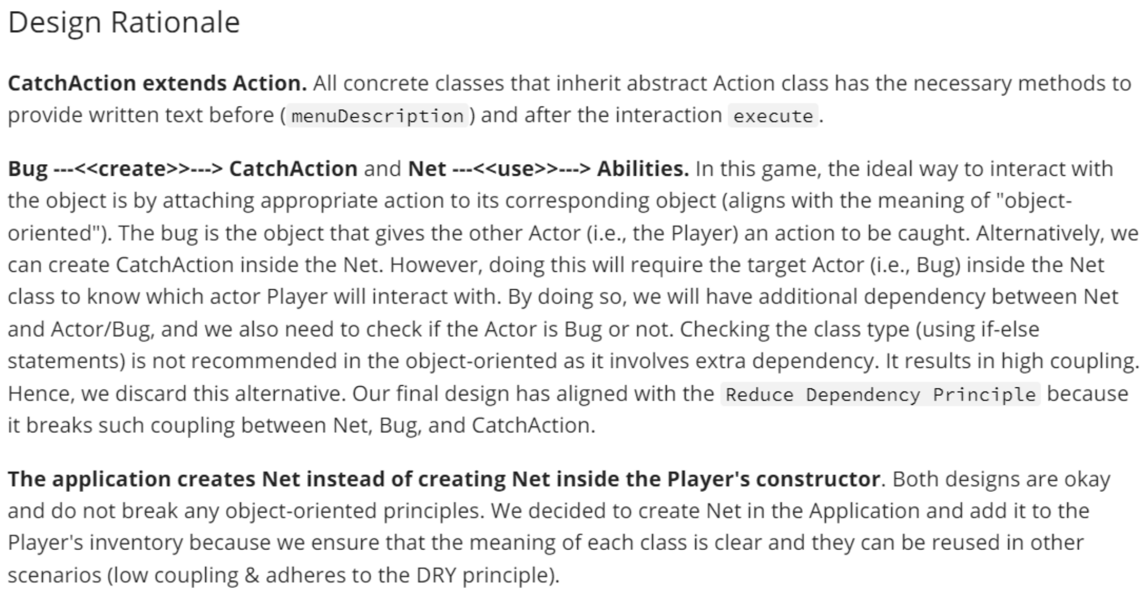
Tips:

- Discuss alternatives e.g. using interfaces rather than abstract classes because of _____ (already extended from Actor and cannot be extended from another abstract class)
- Can use bullet points - don't need lengthy paragraphs
- Don't need to list all classes created in the system -> just need to explain why we did things the way we did them

Req1

Plan:

1) Go through all the new classes created and explain why they extend/implement other classes
2) Go through all relationships -> dependencies and associations
3) Justify unique implementation ideas if they haven't been covered by the above dot points yet (e.g. despawn)



Design Rationale

CatchAction extends Action. All concrete classes that inherit abstract Action class has the necessary methods to provide written text before (`menuDescription`) and after the interaction `execute` .

Bug ---<<create>>---> CatchAction and **Net ---<<use>>---> Abilities.** In this game, the ideal way to interact with the object is by attaching appropriate action to its corresponding object (aligns with the meaning of "object-oriented"). The bug is the object that gives the other Actor (i.e., the Player) an action to be caught. Alternatively, we can create CatchAction inside the Net. However, doing this will require the target Actor (i.e., Bug) inside the Net class to know which actor Player will interact with. By doing so, we will have additional dependency between Net and Actor/Bug, and we also need to check if the Actor is Bug or not. Checking the class type (using if-else statements) is not recommended in the object-oriented as it involves extra dependency. It results in high coupling. Hence, we discard this alternative. Our final design has aligned with the `Reduce Dependency Principle` because it breaks such coupling between Net, Bug, and CatchAction.

The application creates Net instead of creating Net inside the Player's constructor. Both designs are okay and do not break any object-oriented principles. We decided to create Net in the Application and add it to the Player's inventory because we ensure that the meaning of each class is clear and they can be reused in other scenarios (low coupling & adheres to the DRY principle).

Creation of new classes/interfaces etc.

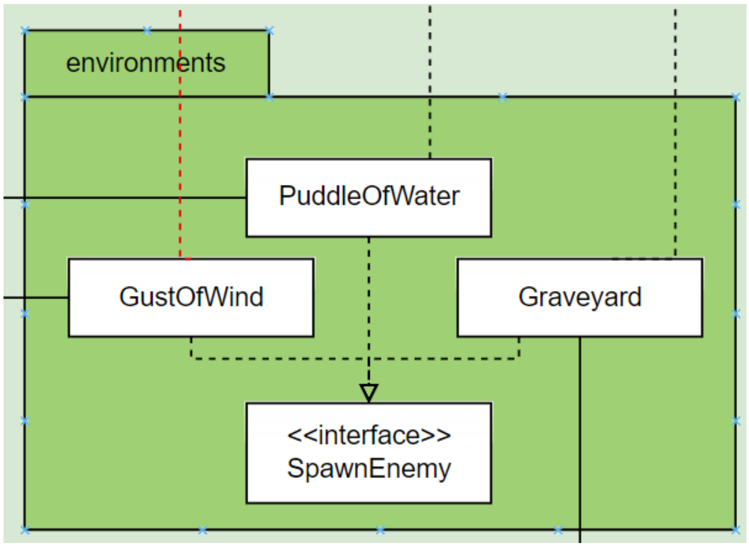
New classes created:

- Actions
    - SlamAreaAttackAction
    - SpinningAreaAttackAction
- Behaviours
    - SlamAreaAttackBehvaiour
- Enemies
    - GiantCrab
    - HeavySkeletalSwordsman
    - PileOfBones
    - NonPlayableCharacter (abstract OR normal class)
- Weapons
    - Grossmesser
- Environments
    - Graveyard
    - GustOfWind
    - PuddleOfWater

New interfaces created:

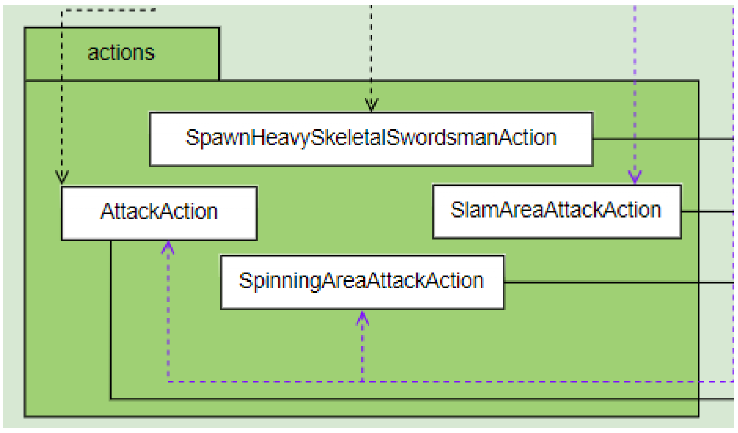
- SpawnEnemy

All environment types (Graveyard, GustOfWind and PuddleOfWater) implement SpawnEnemy interface.



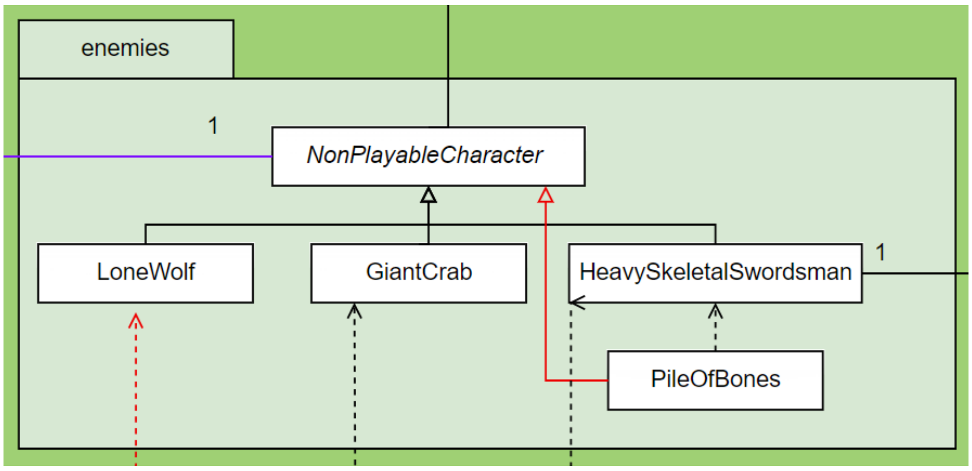
- All concrete classes that implement the SpawnEnemy interface are able to randomly create a corresponding enemy type (e.g. HeavySkeletalSwordsman, LoneWolf or GiantCrab) and place it at its location.
- The environment types currently inherit the abstract Ground class so SpawnEnemy must be an interface 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.
- 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 environments are responsible for handling their own logic for adding an enemy to their location (MAYBE ADD MORE ABOUT WHY IT SATISFIES THIS PRINCIPLE)

AttackAction, SlamAreaAttackAction, SpinningAreaAttackAction and SpawnHeavySkeletalSwordsmanAction 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). 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. We will thus 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.

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.
- 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.

Relationships
Template: Name —<<Association>>—> Name
- Association relationship between enemy types (e.g. LoneWolf) and Behaviour interface is 1 to many because each enemy type can have more than one Behaviour

(e.g. WanderBehaviour, FollowBehaviour, Despawn). Enemies also operate based on their behaviours because they aren't able to consciously select actions.

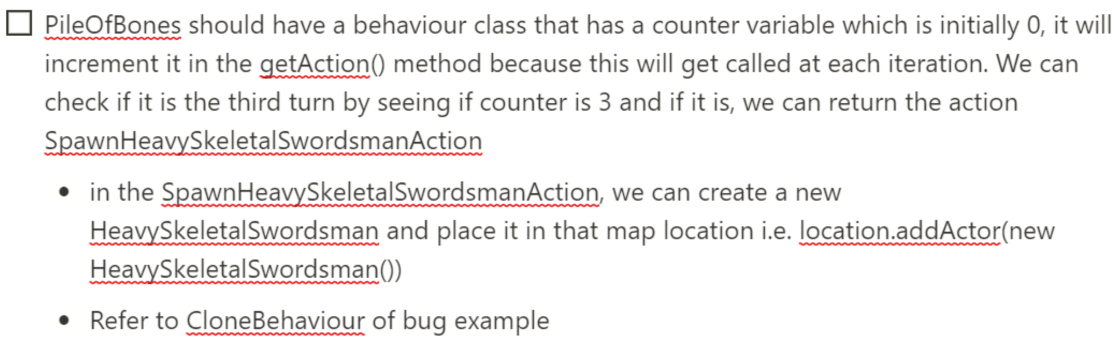
- Association between HeavySkeletalSwordsman and Grossmesser is 1 to 1 because each HeavySkeletalSwordsman instance will have one weapon or Grossmesser as their attribute.
    - Association between Player and Grossmesser (1 to 1) because the player can pick up a Grossmesser and place it in their inventory if they destroy the PileOfBones because it will drop it.
- PileOfBones inherits Actor because it should have hit points for the player to destroy it and it should also have a 'tick' method that tracks the number of game turns -> once it reaches 3 turns before it's hit, it creates a new instance of HeavySkeletalSkeleton and destroys itself (so it will have a dependency with it)
- Dependency between Behaviours and Actions because the behaviour checks a requirement and if it's met, then an Action is returned and if the requirement is not met, null is returned.
    - Alternative: have the behaviour and action in the same class -> this violates the single responsibility principle
- Dependency between Grossmesser and AttackAction as well as SpinningAreaAttackAction because the Grossmesser allows the user to attack just a single enemy which would depend on AttackAction or to perform a spinning attack which attacks all creatures within the user's surroundings and depends on SpinningAreaAttack.
- Dependency between environments and enemies e.g. PuddleOfWater has a dependency on GiantCrab because the environments are able to spawn specific types of enemies and place them at their location.
- Association between AttackBehaviour and Actor and association between SlamAttackBehaviour and Actor
- **AttackAction —<<targets>>—> Actor** and **SlamAreaAttackAction —<<targets>>—> Actor.** This is because each attack behaviour and action will be targeted towards another actor (or multiple actors for the slam attacks)

Become pile of bones

- Instead of a Breakable interface and BreakAction (as shown in the Week 5 Supplementary materials example), we decided to just modify the AttackAction 'execute' method
- modify AttackAction 'execute' method so that the if statement in line 84 has other statements (else-if) statements where it checks if the target is unconscious and if it has the BECOME_PILE_OF_BONES capability and if it does, then do the code on line 85 and then add a PileOfBones in its place

Revive a HeavySkeletalSwordsman from the PileOfBones.

- We do not need to create any interfaces or classes for the pile of bones to be revived. The game specifications state that if the PileOfBones has not been attacked after 3 ticks, it will be removed from the map and there will be a new HeavySkeletalSwordsman in its place.



☐ PileOfBones should have a behaviour class that has a counter variable which is initially 0, it will increment it in the getAction() method because this will get called at each iteration. We can check if it is the third turn by seeing if counter is 3 and if it is, we can return the action SpawnHeavySkeletalSwordsmanAction

- in the SpawnHeavySkeletalSwordsmanAction, we can create a new HeavySkeletalSwordsman and place it in that map location i.e. location.addActor(new HeavySkeletalSwordsman())
- Refer to CloneBehaviour of bug example

Despawn

- The despawn sequence will be implemented inside the Enemy class's playTurn method and will remove the Enemy from the map there. This was the chosen process rather than implementing the despawn sequence as a Behaviour and Action because it is a lot less code to write and reduces the dependencies between Enemy, Behaviour and Action. If we implemented it as a Behaviour and Action, we would need to implement the Behaviour interface and extend the Action abstract class which means that we would have two extra files of code compared to just adding some logic in the existing Enemy class's playTurn method.

Interface segregation principle

- None of the classes implement any interfaces that contain methods or attributes that they don't use.

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 to implement this functionality.
Pros:

- Other Actor classes that interact with enemies can use polymorphism with the abstract interface to recognise similar enemy entities with the runes functionality. This upholds the Liskov Substitution principle. **Player →<<uses>> →GenerateRunes**
- The GenerateRunes interface ensures implementations of behaviours that are required of enemy entities, upholding the Interface Segregation Principle as only enemies implement this functionality for their use.

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. This also 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.

Enemy entities and the Actor class
GiantCrab and HeavySkeletalSwordsman should inherit from the abstract class Actor with an generalisation relationship. These are created 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 it's 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 map through Actor.

MerchantKale
Will have a generalisation relationship with the Actor class, an association with the Behaviour class and dependencies on most of the weapon classes
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, violating Open and Closed principles.

Status added capabilities
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.

Req3

Flask of Crimson Tears
An object that inherits the Item class under the possession of a Player. This Is instantiated in Application

Site of Lost Grace

Pick-up Runes

Game reset
Uses GameMap to get all Actors/objects on Map
Uses Resettable for Actors that can be resetted (use capability/status?)
GameMap.remove all actors/enemies
Added resettable to Flask of crimson tears
World.Add player to site of lost grace
Player action

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

Alternatives and their cons:

- Modify the application class to hard code the specific values 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.

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.Thus, uphold the dependency inversion principle.
- However, the application class will need to be modified to include further archetype options, violating the open-closed principle for further functionality

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 much like AttackAction.

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 use 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 functionality.

Req5

- UML same as REQ1
    - Check what *Lands Between* is (a ground? Or the environment?)
    - Add Runes (links between it and enemies, weapons, players, Merchant Kale)
    - Link between Scimitar and Merchant Kale
    - Add Merchant Kale to entities package
    - Resettable grounds -> despawns all enemies

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.
- 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.
- An alternative to this would be having if-else statements in the Application class that goes through the list of actors and resets them accordingly but this violates the ______ (reduce dependency principle?)

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.
- 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.
- 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.

Revive a SkeletalBandit from the PileOfBones.

- SpawnSkeletalBanditBehaviour and SpawnSkeletalBanditAction
- The PileOfBones can have a capability to determine whether it can be revived to form a SkeletalBandit or HeavySkeletalSowrdsman

MerchantKale

Player has a one to one association with the Runes class.

- The player carries a Runes object that will hold the total number of runes that they have.

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

- Have a capability EAST and WEST for each of the locations/environments and check this before spawning the enemies in the spawnEnemy method from the SpawnEnemy interface.