Assignment 1 UML Diagrams and Design Rationale

REQ 1 + REQ 5:

Environment:

- **Graveyard, GustOfWind, and PuddleOfWater extends from <<abstract>> Environment. (Inheritance / Generalization) implements <<interface>> Spawn**
  - The Environment class is set to be an abstract class due to the different types of environment generated in the game, resulting in different attributes that are needed in each environment.
  - An abstract class is needed as we don't need to create any objects inside the Environment class but we will need to implement and create subclasses where it will dictate the different conditions and enemies that will spawn. This allows us to maintain the Dependency Inversion principle.
  - The current child classes which are created through an inheritance (generalisation) relationship are Graveyard, GustOfWind, and PuddleOfWater.
  - More types of environments may potentially be introduced, having an abstract class implemented with common environmental characteristics will prevent repetition.
  - The objective of incorporating a Spawn interface is to benefit from dependency injection and enhance flexibility. This is due to the fact that all child environment classes have the responsibility of generating enemies.
  - And making it easier to change the manipulation or integration of different enemy mechanisms without violating the core logic of the environment classes.
  - As a result, using the Spawn interface allows us to modify the method uniquely under each environment type. As each environment is limited to spawning particular enemies and possesses distinct probabilities for enemy generation using a RandomNumberGenerator.
  - Therefore, justified the use of Spawn Interface in this case.
  - Furthermore, this approach follows the Interface Segregation Principle (ISP), which emphasises maintaining a concise interface consisting of a single, high-quality method.
  - Yet, the potential cons can be increased complexity. Using an interface can potentially add an extra layer of abstraction, increasing the complexity of the overall design.
  - However, in this case we only have 1 interface. Thus, this con doesn't affect our code in any aspect significantly. Rather, this issue might be concerned if the design scope gets massive. Which makes more sense for us to implement in this way for now.

- **<<abstract>> Environments extends Ground (Inheritance / Generalization)**
  - By extending the abstract Ground class, we can create each environment with their appropriate display character. This enables the abstract Environments class to supply a generic method for initialising the correct corresponding symbols for each child environment class.
  - This approach provides few pros like reducing code complexity and easier maintenance, etc. As the abstract Environments class now contains all the necessary methods. Such as deSpawn and determineSpawnArea.
  - Otherwise this can cause duplicated codes since determineSpawnArea and deSpawn apply to all subclasses in the exact same way.
  - On top of that, each child environment would have to extend from the Ground class directly, which would be problematic since Java only permits single class extensions.
  - Now with a centralised implementation of common methods in the abstract Environments class, it becomes easier to modify or fix any issues that may arise in the shared functionality.
  - Moreover, this helps to achieve the Liskov substitution Principle (LSP). As this approach enables us to perform polymorphism. Enabling the use of a single interface to represent different types of environments and simplifying the interactions between different components in the system.
  - However, the con can be tight coupling. Since right now we create a whole chain of relationships between the parent Ground class and child Environment classes including parent abstract Environment class. Which can make the codebase harder to refactor and maintain in the long term.
  - Although the con may have a huge impact in the future, the benefits of this approach introduce code reusability, organised hierarchy, and simplified code. Nevertheless, as this assignment wouldn't exceed more than the current scope that much. This approach is considered viable and reasonable for current design.

- **<<abstract>> Environments imports <<Enum>> Direction.**
  - As per requirement, we have two directions of the map which are east and west. By using an enum, we can represent both map directions specifically as unchangeable values.
  - In addition, this allows us to extend the code in the future if necessary. Such as adding North and South directions without causing any refactoring issue.
  - Therefore, using enum class is a logical approach to achieve consistency in comparison to other methods.
  - Moreover, if strings were used, there are a few different ways it could be used. Eg. "West", "west" and "WEST" could be used to indicate an environment's relative direction. This could cause some inconsistencies when collaborating.
  - With directions taken into consideration when spawning enemies, having an enumeration for directions can indicate where the environment resides

relative to the map and what specific enemies spawn. Eg. In the Graveyard, east spawns SkeletalBandit and West spawns HeavySkeletalSwordsman.

- ➢ Having an enum that contains the directions, allows us to maintain the single-responsibility principle (SRP) as this enum is responsible for maintaining direction throughout the entire implementation.

- ❖ **Environment classes spawns and deSpawn East Enemy and West Enemy (Association)**
  - ➢ Each environment subclass is responsible for spawning enemies based on its type and direction, with a specific probability.
  - ➢ The environment subclasses keep track of the enemies (Actors) currently in their area to prevent new enemy spawns when other enemies are nearby.
  - ➢ When an enemy is spawned, the determineSpawn method should be used.
  - ➢ If an Actor is already at the location, the environment should initiate a despawn process, where the specific enemy has a 10% chance of being despawned.
  - ➢ The relationship between environment subclasses and enemies residing within them is an association relationship, with zero or many enemies associated with each environment.
  - ➢ Enemies can follow the player to another environment, and as a result, the relationship between the environment and the enemies has been designed to accommodate zero or many associations.
  - ➢ Each area is responsible for spawning an enemy given its environment and direction with specific probability.

Enemies :

Note: please see the weapons section at the bottom for information about enemies holding weapons.

- **<<abstract>> Enemy (Inheritance / Generalization)**
  - The Enemy class is set to be an abstract class due to the different types of enemy generated in the game to which they have similar properties such as health, weapon, IntrinsicWeapons, skills, behaviour, etc. but are initialised differently.
  - Behavioural attributes of each enemy are also implemented almost identically, with characteristics such as not being able to attack the same class, following the player to attack, attacking other types of enemies but not following, etc.
  - We will not create any objects inside the Enemy class but will in each subclass, which will create an inheritance (generalisation) relationship. If new enemies are introduced to the code, the same abstract class can be used to implement all the necessary behaviours and stats required for a valid enemy.
  - Through implementing the Enemy class as abstract, this enables future enemies to be implemented uniquely to the other existing classes without affecting their behaviours. (Dependency Inversion Principle)

- **New Skeleton, Crustacean and Canine Abstract classes**
  - By providing another layer of abstraction for these 3 types of enemies, this allows us to maintain the OCP principles as more enemies in the future can be implemented through utilising these abstract classes if they are the same type.
  - A majority of existing behaviours will be implemented as a result for new Enemies that may be created.
  - Since enemies of the same types do not attack each other, this method allows us to identify and compare class types. Previously, this used enums containing Enemy types, however, that faced some issues with allowing for further implementation as some of the common behaviours between enemy types would need to be reimplemented.

- **Enemy Stores runes values instead of Storing the Rune class**
  - Defeating any enemy should award the player with Runes and thus should store a Rune class within the enemy so that it can still exist when the enemy class gets deleted.
  - Using numerical values to store Runes within the enemy will continue to rely on the enemy class to hold the Rune, even after death. Eg. having an integer class attribute called Runes.
  - Since Runes can remain in a location after the player's death, reusing the defeated enemy's class to hold the rune value would interfere with other functions such as spawning new healthy enemies,etc.
  - Thus, the abstract enemy class and runes have an association relationship.

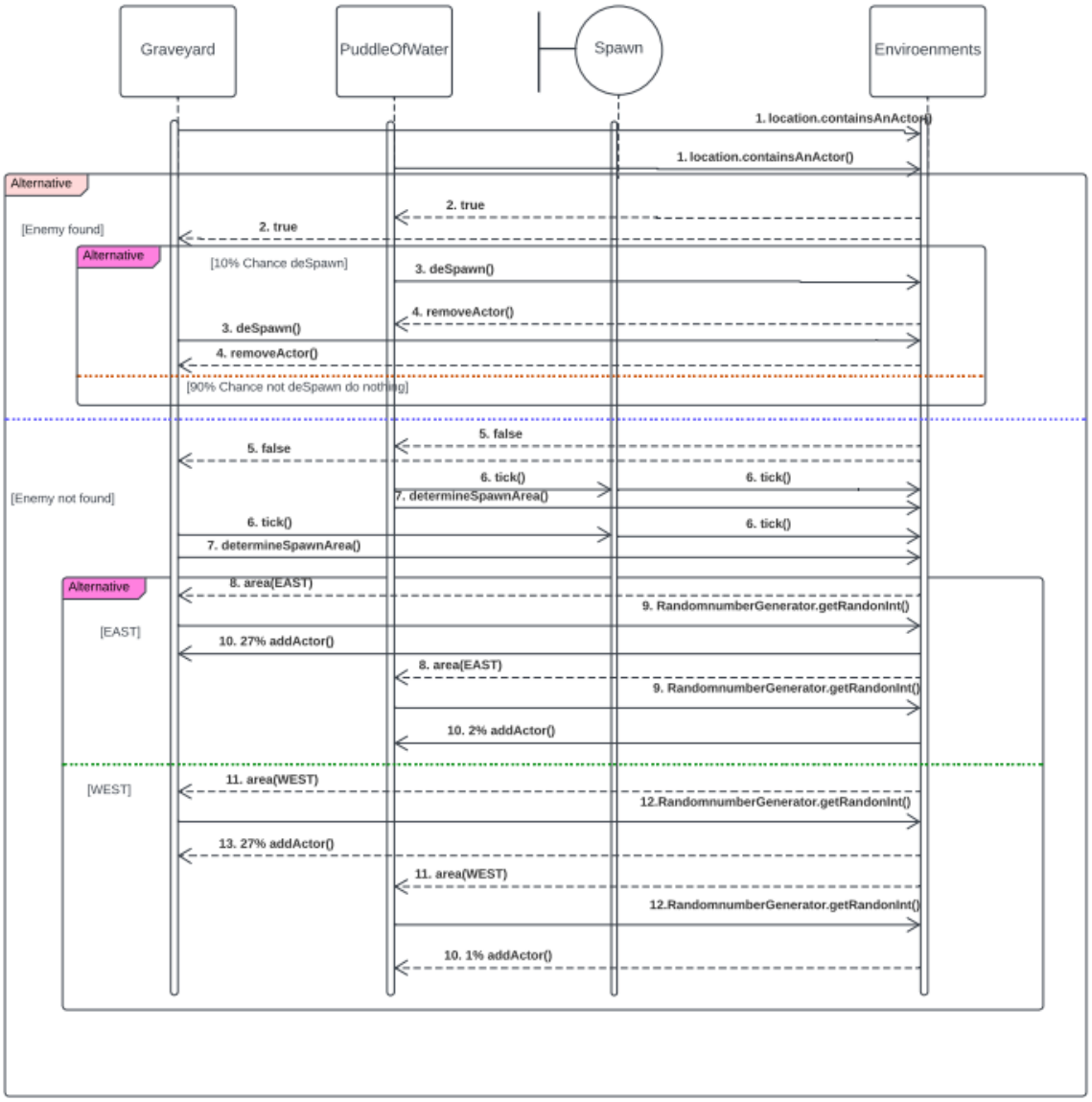
- **PileOfBoneActor, PileOfBoneBehaviour, PileOfBoneAction is used by Skeleton Enemies**

- ➢ By the original PileofBones Action into 3 different parts, this reduces the responsibilities and maintains SRP.
- ➢ PileOfBonesActor extends from the Actor class and is used to place an actor on the map to represent the passive ability (X) activated by the skeleton classes.
- ➢ PileOfBonesBehaviour implements the behaviour interface so that the activation conditions can be implemented into this class and also can be integrated into the current behaviour system. Previous UML diagrams utilised a new interface, however, the old idea could not be integrated as it would ignore the engine all together.
- ➢ PileOfBonesAction extends from the Action Class to contain the executable action for the PileOfBonesActor.
- ➢ As PileOfBonesBehaviour is placed into the Skeleton class's behaviour hashmap, this allows the game to check if the skeleton's last action was death and return the PileOfBones action.
- ➢ The behaviour creates an association relationship as the pile of bones behaviour will be stored in the behaviour map inside of the skeleton class.

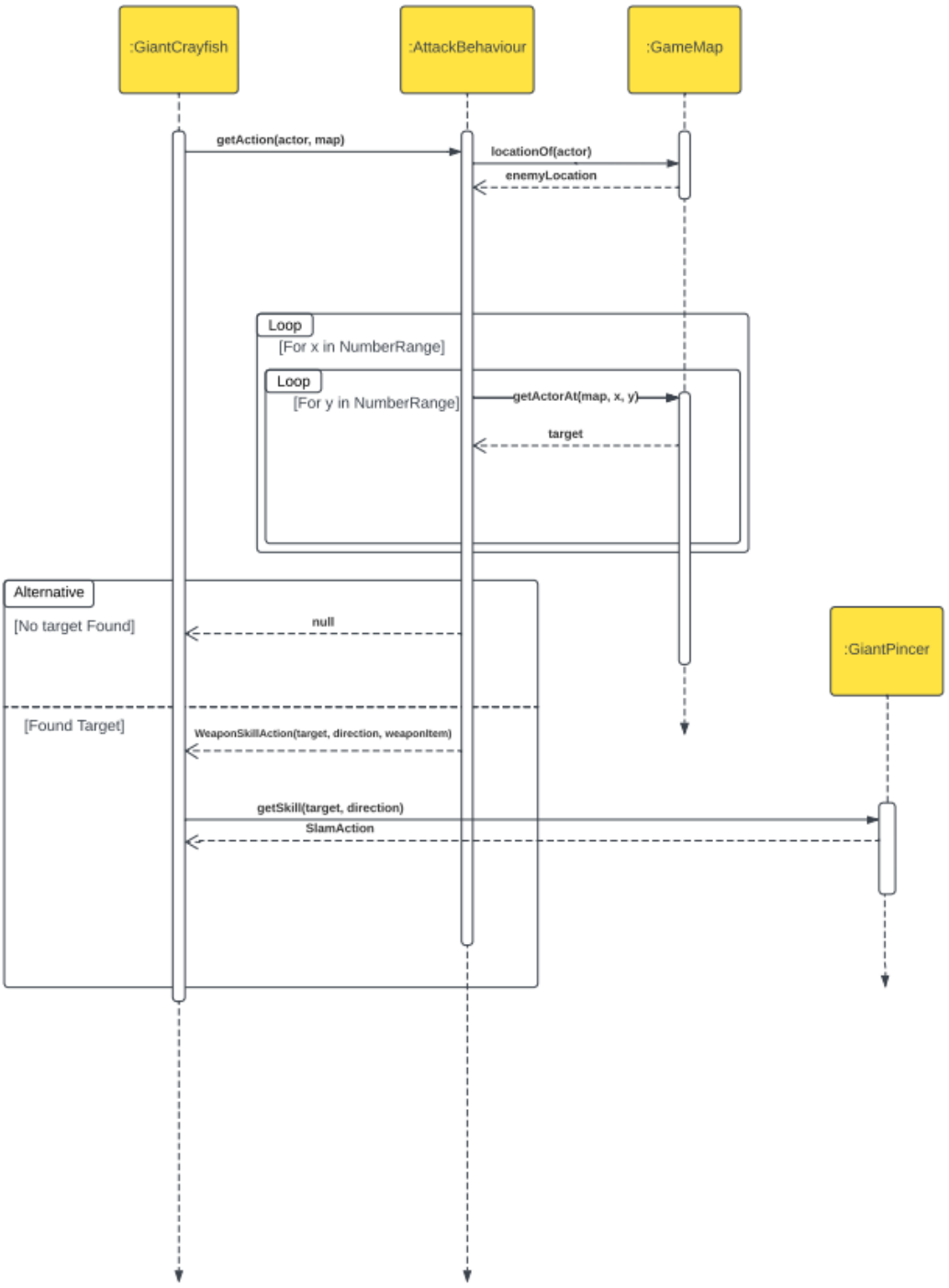
❖ **GiantPincer, GiantHead, GiantClaw inherit Slamable (WeaponItem) which depends on SlamAction.**

- ➢ SlamAction is a special skill that the Giant type Enemies have, which consists of slamming a target and damaging actors in a certain radius.
- ➢ As each Actor had different damage outputs for the same skill, through implementing an undroppable WeaponItems, each enemy can have a way to track how much damage they output and this can be used by calling the getSkill method inside of the WeaponItem class.
- ➢ Another alternative solution was to have the enemy class hold a skill return method. However, not all skills involved would have damage attributes such as how the skeleton class had the Pile of Bones abilities. This as a result would potentially cause LSP to be broken as the child classes of the Enemy class would not be able to utilise this method.
- ➢ This implementation also allows us to maintain OCP as although it was possible to use Exit locations in the engine package. By having the Slam Action take in a target and radius, it can be further implemented in the future for Area of Effect attack actions. One example could be a bigger slam, or adding after-effects such as burning or freezing the actors.
- ➢ This however, provides some issues with flexibility as the way that the damage is delivered cannot be changed. Although there is some flexibility with the damage value and output descriptions, the implementation does not allow for modifications on how the damage is delivered. As a result, features such as a different damage output per tile may not be possible without modifying the file itself.

REQ 1 Sequence Diagram: Spawning Process



REQ 5 Sequence Diagram: Slam Skill Action on target



REQ 2:
Rune:

- **Runes Class extends Items**
  - Runes are the main form of currency used within the game.
  - Merchant classes will not require a rune class (See Trader Section for more detail)
  - Runes class extends from Item class to inherit the basic properties and functionalities as Runes is an in-game item.
  - In this way, enabling the player to pick up the runes on the location they died or from the enemies.
  - Since the only person who drops runes is the player (on death), the Runes class is a simple class that extends from the Items class and holds a value.
  - Any transaction methods are implemented in the Runes Manager class to maintain SRP.
  - By extending from Items, the Runes class can have the portability attribute which allows it to be picked up and dropped, allowing us to integrate it into the current Item system, especially when the player themselves have these actions available.
  - This class allows us to maintain the Open/Close principle as the Rune class will not affect any current functionality but also allows us to provide new rune related functionality without affecting the other classes.
  - On top of that, using factory method design patterns to obtain the benefit of easier future modifications.
  - Since the requirement for creating Rune changed, all we need to do is to update the Runes's instance from the factory method.
  - In this way, the impact on other parts of the code can be minimised.
  - However, the potential drawback can be when the Rune class is no longer needed in the future.
  - Which can be difficult to refactor the code as it requires modifying all the classes that use Runes class. Since right now Runes class's factory method is a singleton instance.
  - Despite this drawback, this approach still is viable and reasonable as this provides more flexibility and scalability for creating and managing Runes, especially when it is a vital part to the game.

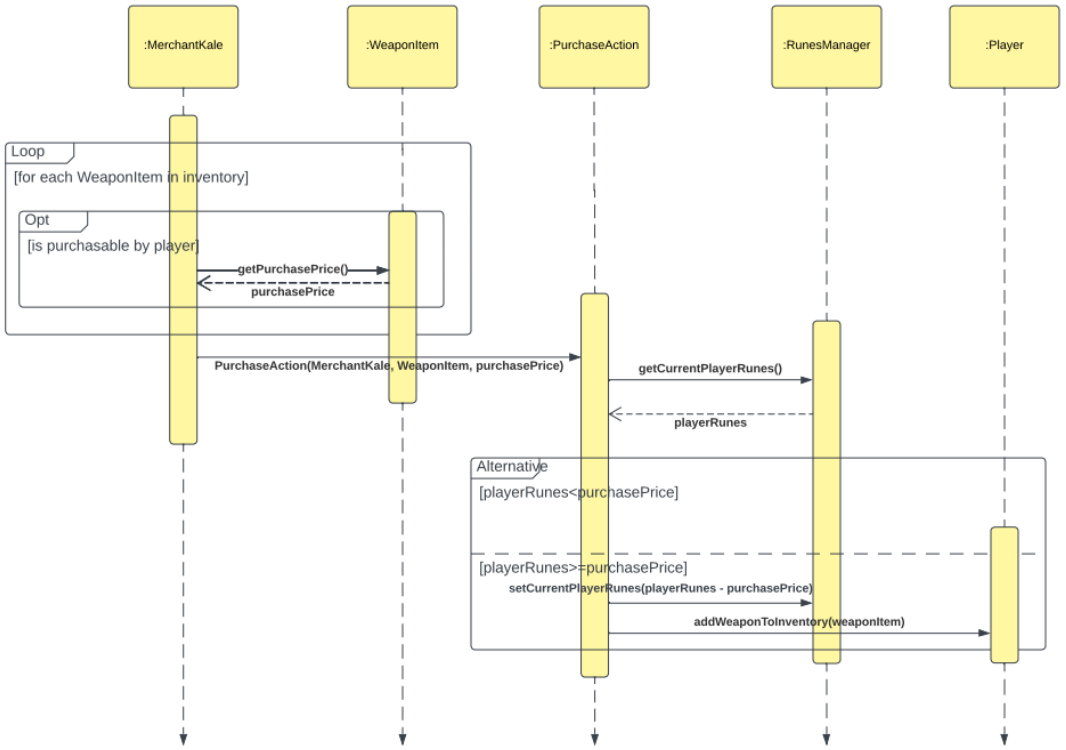
- **Runes Manager Class**
  - The purpose of the Runes Manager class is to manage the in-game currency (Runes) for the players. That is responsible for creating and maintaining a single instance of Runes object associated with the player.
  - Meanwhile providing essential methods for managing the runes value and interaction such as dropRunesFromEnemy(). The player will obtain the amount of Runes dropped from the enemy they defeated.

- ➢ By doing this approach, separating into Runes and RunesManager, created an interface DropRunes, preventing the breach of SRP.
- ➢ Otherwise, a singleton Runes class or RunesManager will have too many responsibilities.
- ➢ By following SRP it will decrease the difficulty to maintain the quality of the code and on a larger scale, overall code can be easier to modify if any change is needed.
- ➢ Despite the benefits of this approach, we also tried to avoid too many interfaces or small classes created.
- ➢ Since this will overcomplicate the overall design leads to increment of complexity.
- ➢ In this case we did avoid this to occur and carefully distributed the amount of the responsibilities that each class doesn't contain any extra responsibility.

Traders:

- ❖ **MerchantKale extends Actor implements Merchant. (Inheritance/Generalization)**
  - ➢ The purpose of this MerchantKale class is to represent the trader character in-game who trades weapons with the player. The class is responsible for creating and maintaining the merchant character's inventory and defining actions available when the player interacts with the marchant.
  - ➢ Which is the core reason why we extend from the Actor abstract class. Since an actor entity holds inventory that stores items. Making it perfect to extend from Actor class.
  - ➢ Which then creates a generalisation relationship between MerchantKale as now MerchantKale is able to access Actor class by calling Actor.
  - ➢ Therefore, we overriding the allowableACtion method from the Actor class to design the actions that the player can take when interacting with the merchant in-game.
  - ➢ Including checking if the player can trade, then creating SellAction and PurchaseAction instances for each sellable and purchasable item in both player's and merchant's inventory.
  - ➢ On top of that, extending from the abstract Actor class and implementing Merchant interface will prevent breaking open/close principles. Even if we add a new feature it won't break the code since the code is extended by interface and abstract Actor class.
  - ➢ Ensuring our code is easier to maintain and extend in the future.
  - ➢ Yet, from time to time this can make the system more complex. However, evaluating the assignment scope, this shouldn't be any issue.
  - ➢ Therefore, it is a viable and reasonable approach for us to implement games.

REQ 2 Sequence Diagram: Purchasing a WeaponItem from Merchant Kale



REQ 3:

Grace & Game Reset:

- **LostSiteOfGrace extends Ground (inheritance / generalisation)**
  - Lost site of grace is a checkpoint of the game which has a specific location on the game map like other environments.
  - Which holds similar properties like other environments, but has different behaviour, only for this specific area.
  - And since the Ground class is used to represent a ground object it makes logical sense for us to inherit from the Ground class.
  - Since we want to have a generalisation relationship as LostSiteOfgrace should be depended on the Ground class. And we want to treat LostSiteOfgrace as object Ground.
  - Making this relationship an essential part of our design.
  - Meanwhile using ActionList to ensure if the player decided to rest the player is able to choose from the actionList to perform rest action.
  - This entire class is designed to provide a rest action for the player that is standing on the location by using allowableAction().
  - As such, if we did not extend Ground, whenever there is an update related to LostSiteOfGrace, the area will not be updated accordingly. Resulting in insufficient game reset, further causing issues within the entirety of the game and code structure.
  - On top of that, this design complies with SRP. As LostSiteOfGrace is designed for single responsibility. We only perform restAction if it's needed. (more explanation under GameReset dependant on PlayerDeath part)
  - Thus, making the class itself very easy to maintain, and free from potential refactoring issues in the future.

- **RestManager**
  - The purpose of this class is very crucial for the entire game. Since it is used to manage various game elements that need to be reset under certain conditions. Such as, player death or player interacts with the SiteOfLostGrace.
  - This creates a dependency relationship between LostOfSiteGrace as we pass LostOfSiteGrace as a parameter.
  - Yet, this might result in modifications in between these two classes if any changes are needed in the future. However, we still designed in this way due to the use of manager class and LostOfSiterace is a single responsibility class.
  - Therefore, this wouldn’t cause us any major issues if we are required to modify anything.
  - Even for the extension this should be very manageable. Since the purpose of this class is to control all the relevant operations of GameRest.
  - By managing these resettable elements, the ResetManger class ensures the game mechanics would function as it is required.

- ➢ Including two methods for the reset triggers, runWithPlayerDeath() / runWith LostGrace() helps to distinguish between reset processes depending on the scenario.
- ➢ Also, using factory methods to achieve a singleton design pattern to ensure that only one instance of the Resetmanager can be created throughout the game's lifecycle and it will be used for the entire game.
- ➢ As such, this design provides us more benefits such as scalability, flexibility and easier management than the cons.
- ➢ Making this approach a reasonable and logical approach.

❖ **ResetManager is dependent on deathAction and LostSiteOfGrace.**

- ➢ We could have implemented PlayerDeath and LostSiteOfGrace as one class, GameReset. However we found out that even though they both trigger the game reset, there are some distinctive behavioural differences.
- ➢ The PlayerDeath will cause the player to lose all Runes and the player can die anywhere which is why there is a dependency relationship between Environment and PlayerDeath. However, LostSiteOfGrace has a specific location.
- ➢ By separating PlayerDeath and LostSiteOfGrace, it allows us to maintain the single-responsibility principle as we want the PlayerDeath to manage Death related behaviours (such as dropping runes, reviving, etc) whilst having the LostSiteOfGrace manage the functions of the environment as a spawn location.

❖ **ResetManager resets FlaskOfCrimsonTears, Player, and Enemies**

- ➢ GameReset resets the FlaskOfCrimsonTears back to the maximum the player can hold, which is two for now and it also removes all the enemies from the map.
- ➢ This is achieved by creating a realisation between resettable interfaces. Since we want to treat the CrimsonTears as a resstable object. So we can keep track of the usage of the Tears.
- ➢ Though ResetManager will be depending on the resettable interface. It still still allows plenty of extension potential in the future.
- ➢ RestManager also resets the player when the player dies, by calling the reset method inside the player class. The reason why resting players are not implemented inside the rest manager is because we want to avoid creating a god class.
- ➢ The reason why we chose the relationship to be dependency, is because every time when we reset the FlaskOfCrimsonTears, we reset it to a specific number, to do this does not require to create a class attribute of FlaskOfCrimosonTears inside the GameReset class.
- ➢ We can just call a setter method inside FlaskOfCrimsonTears. Same with removing Enemies from map, a method from Enemies class will be efficient enough to achieve this.

- **FlaskOfCrimsonTears extends Item Implements resettable.**
  - FlaskOfCrimsonTears will be in a consumable package as it is likely there will be other consumable items that will be introduced to the game, but some of them may not be resettable by game reset.
  - Having the FlaskOfCrimsonTears class allows us to maintain the single-responsibility principle as this class will be responsible for keeping all the functionality of itself as well as being able to store it inside the player's inventory.

- **UseFlaskOfCrimsonTears extends from Action**
  - Having the UseFlaskOfCrimeTears separate from FlaskOfCrimsonTears , it follows the single-responsibility principle as this is the class that will only be incharge of the action, use FlaskOfCrimsonTears.

- **PlayerDeath loses Runes.**
  - Since only PlayerDeath loses Runes, but LostSiteOfGrace doesn't, we only have dependency relations between Runes and PlayerDeath.
  - The reason we chose dependency is because PlayerDeath will not store an instance of Runes but rather make the Runes drop on the map.

REQ 4:

Classes:

- **Combat Archetypes via composition instead of generalisation (inheritance)**
  - Instead of having Samurai, Bandit and Wretch classes inherit the player class, using composition allows us to maintain OCP in a much more efficient manner without affecting other classes.
  - Through having the 3 Action classes: ChooseBanditAction, ChooseSamuraiAction and ChooseWretchAction, the player class becomes more flexible as it can be reused in comparison to generalisation where the player class would need to be replaced especially in the given engine's code.
  - This is especially useful since the starting CombatArchetypes only affect the player's beginning state. Each combat archetype does not provide special skills (except for the ones in the weapons) nor does it provide any unique mechanics.
  - As a result, each Action class has a dependency relationship with the player class, since the player class does not store each action class and only calls them to modify itself.

Weapons (And Weapon Skills):

- **Uchigatana, GreatKnife and Club —--<<implements>>-----> Sellable and Purchasable**
  - The two interfaces Sellable and Purchasable are implemented into each WeaponItem that can be sold or purchased to the Merchant classes.
  - This is because there are weapons like Grossmesser, which the player cannot purchase from the Merchant but has the ability to sell after picking it up from an enemy.
  - Through having two separate Interfaces for Sellable and Purchasable, this prevents us from breaking the Interface Segregation Principle for the cases as specified in the previous point.

- **Uchigatana, GreatKnife, Club, Scimitar and Grossmesser extends WeaponItem.**
  - These are all the weapons in the game, they all share some common traits, such as can be used by the enemy or player for attack. Therefore, we created an abstract class of WeaponItem to categorise them.
  - This follows the Open/Close Principle as we may add new weapons to the game that have similar properties.

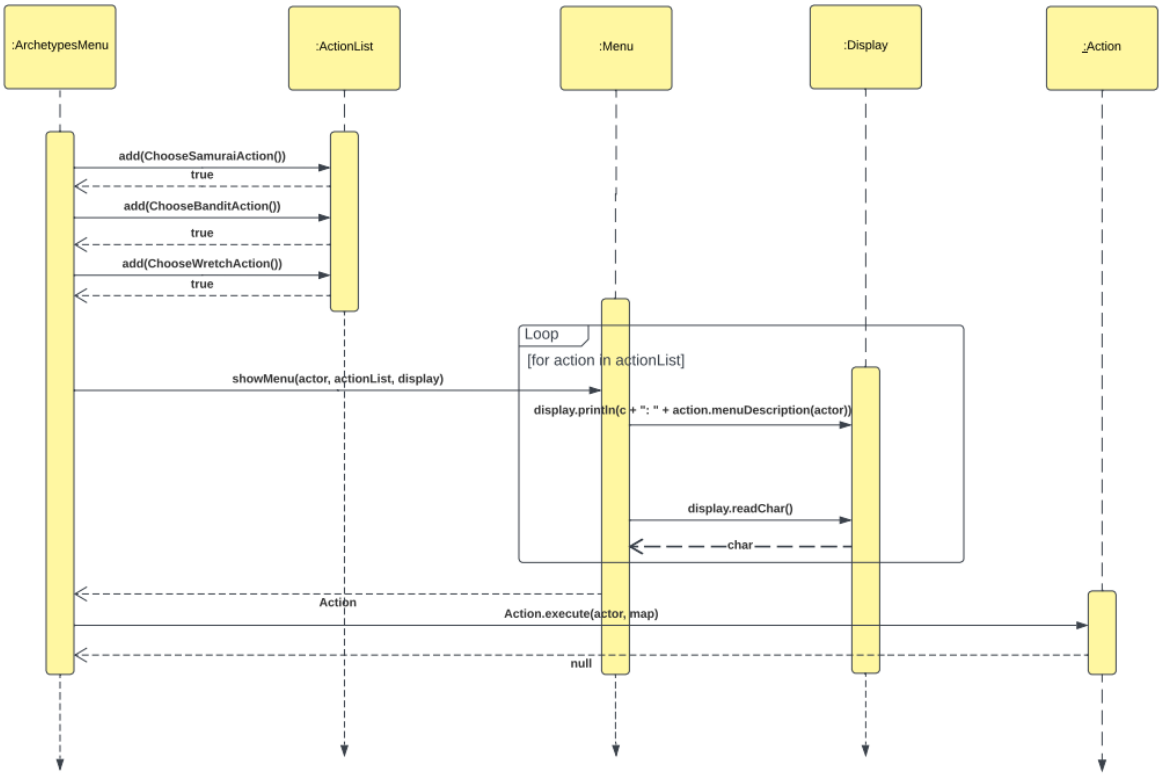
- **Uchigatana —-<<Performs>>----> Unsheathe, GreatKnife—-<<Performs>>---> QuickStep, etc. WeaponItem and WeaponSkillAction.**
  - In this game some weapons will have unique skills that the player can use if they hold that weapon. For example, Uchigatana and GreatKnife have unique skills that the player can use during a fight which are Unsheathe and Quickstep respectively.

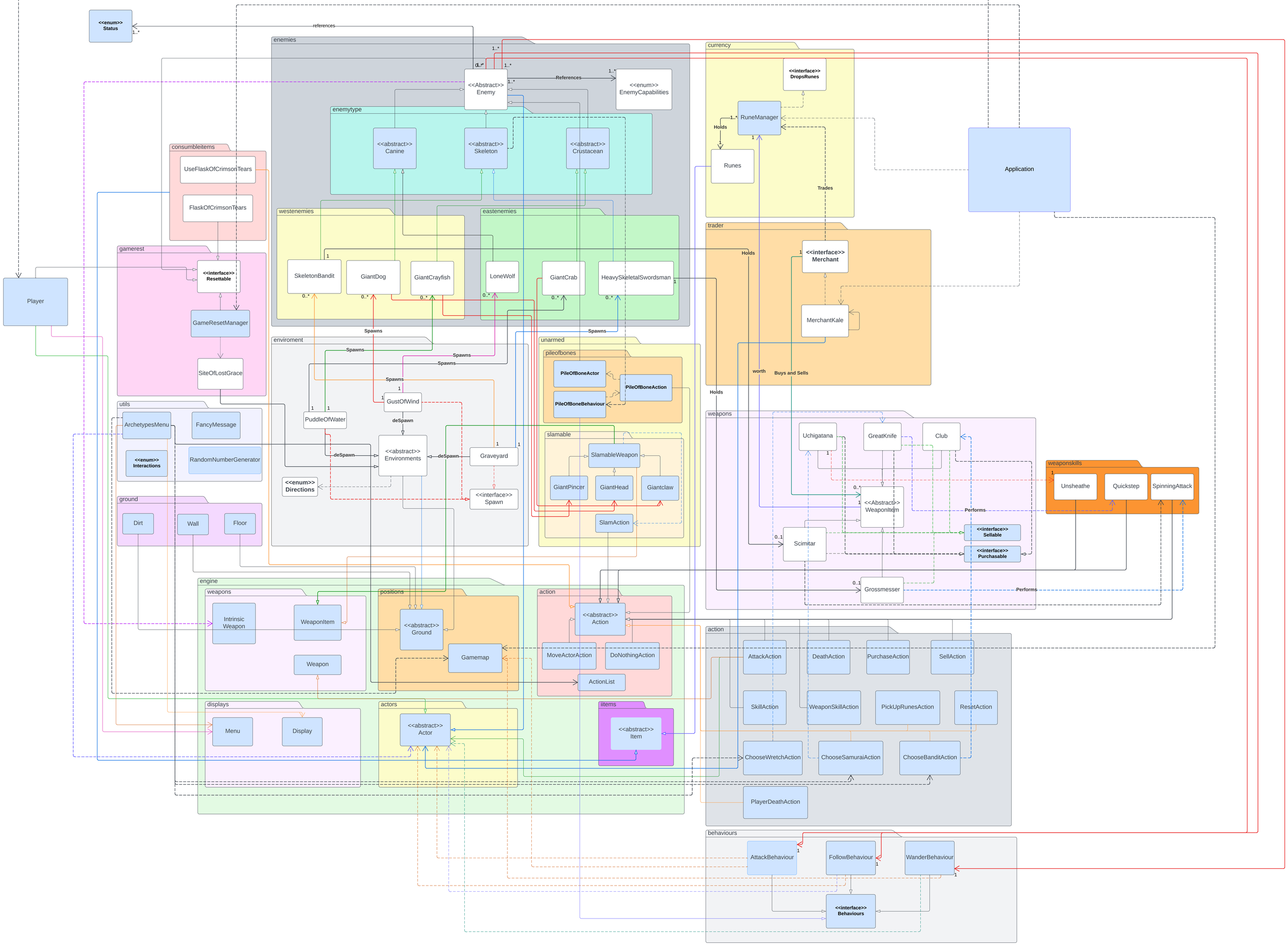
- ➢ These Skill Actions are called via defensive copying and this causes the relationship between a WeaponItem and the WeaponSkillAction to be a dependency.
- ➢ By using defensive copying, this prevents the Skills from being altered from the outside, which can potentially change the way the game is played out.
- ➢ Unsheathe and QuickStep extends from Action and are called via getSkill(), this allows these skills to utilise the game's engine and provide their usage through the allowable actions method.

❖ **Enemies and their weapons (and Intrinsic Weapons)**

- ➢ Each enemy can only hold up to one weapon only.
- ➢ Enemies who do not hold weapons at all will be able to use their intrinsic weapons.
- ➢ Intrinsic Weapons are a fully implemented class of its own and can be used by any character with attacking functions without a weapon. Thus, characters who do not hold a weapon can use their intrinsic weapon with their own initialised values.
- ➢ Enemies with weapons have association relationships with the given weapon they equip, having 1 enemy to 1 weapon.
- ➢ Since it can be possible for the player to equip dropped weapons, Enemy weapons follow the same structure as every other WeaponItem subclass, inheriting the WeaponItem abstract class.
- ➢ This allows us to maintain the single-responsibility principle as by separating the weapons and enemy implementation, we prevent putting extra responsibility on the enemy to contain behaviours of the weapons they utilise as well.
- ➢ Through implementing weapons via the WeaponItem class with generalisation, weapons can be picked up and dropped whilst also providing the player with allowable actions related to itself such as using weapon skills.

REQ 4 Sequence Diagram: Choosing a starting character Archetype




<<enum>>
Status
references
1..*
enemies
<<Abstract>>
Enemy
References
<<enum>>
EnemyCapabilities
enemytype
<<abstract>>
Canine
<<abstract>>
Skeleton
<<abstract>>
Crustacean
consumbleitems
UseFlaskOfCrimsonTears
FlaskOfCrimsonTears
westenemies
eastenemies
SkeletonBandit
GiantDog
GiantCrayfish
LoneWolf
GiantCrab
HeavySkeletalSwordsman
currency
<<interface>>
DropsRunes
RuneManager
Holds
Runes
Trades
Application
gamerest
<<interface>>
Resettable
GameResetManager
SiteOfLostGrace
Player
trader
Holds
<<interface>>
Merchant
MerchantKale
worth
Buys and Sells
Spawns
enviroment
GustOfWind
deSpawn
PuddleOfWater
<<abstract>>
Environments
Graveyard
<<interface>>
Spawn
<<enum>>
Directions
unarmed
pileofbones
PileOfBoneActor
PileOfBoneAction
PileOfBoneBehaviour
slamable
SlamableWeapon
GiantPincer
GiantHead
Giantclaw
SlamAction
weapons
Uchigatana
GreatKnife
Club
<<Abstract>>
WeaponItem
Scimitar
Grossmesser
Performs
weaponskills
Unsheathe
Quickstep
SpinningAttack
<<interface>>
Sellable
<<interface>>
Purchasable
utils
ArchetypesMenu
FancyMessage
<<enum>>
Interactions
RandomNumberGenerator
ground
Dirt
Wall
Floor
engine
weapons
Intrinsic
Weapon
WeaponItem
Weapon
positions
<<abstract>>
Ground
Gamemap
action
<<abstract>>
Action
MoveActorAction
DoNothingAction
ActionList
displays
Menu
Display
actors
<<abstract>>
Actor
items
<<abstract>>
Item
action
AttackAction
DeathAction
PurchaseAction
SellAction
SkillAction
WeaponSkillAction
PickUpRunesAction
ResetAction
ChooseWretchAction
ChooseSamuraiAction
ChooseBanditAction
PlayerDeathAction
behaviours
AttackBehaviour
FollowBehaviour
WanderBehaviour
<<interface>>
Behaviours