Design Rationale Assignment 2

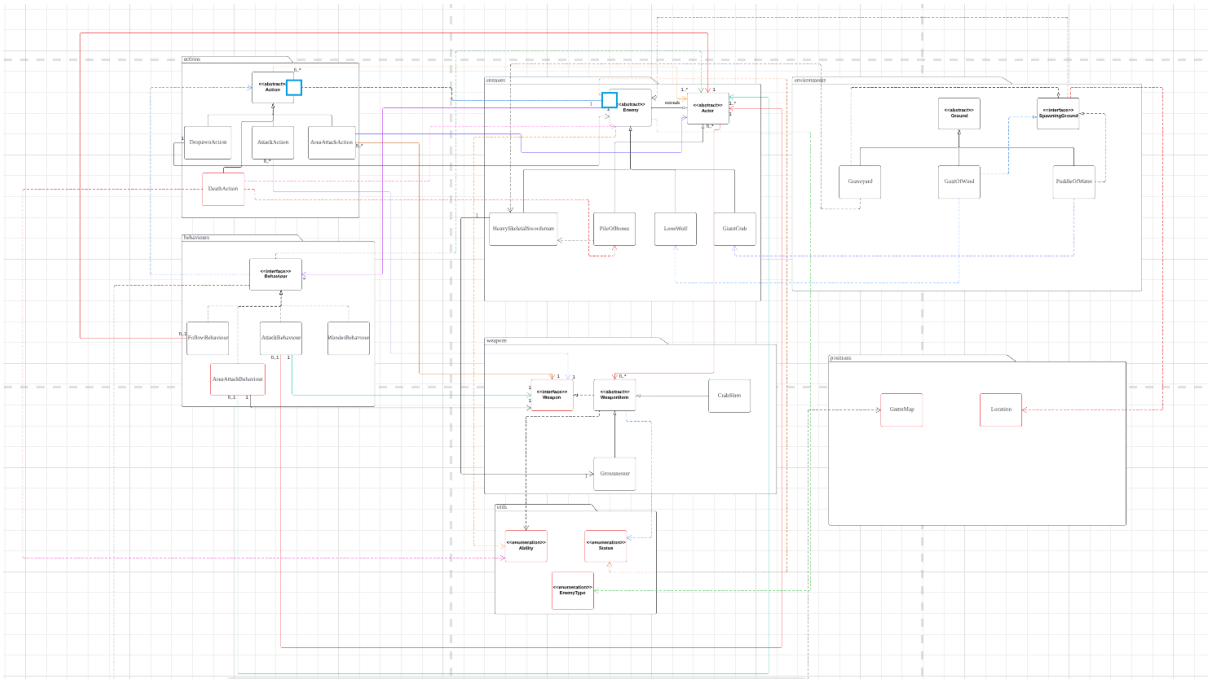
MA_Lab05_Group6

Lai Khairong 33271232
Lim Yu Ean 32619561
Tan Jian Kai 32560923

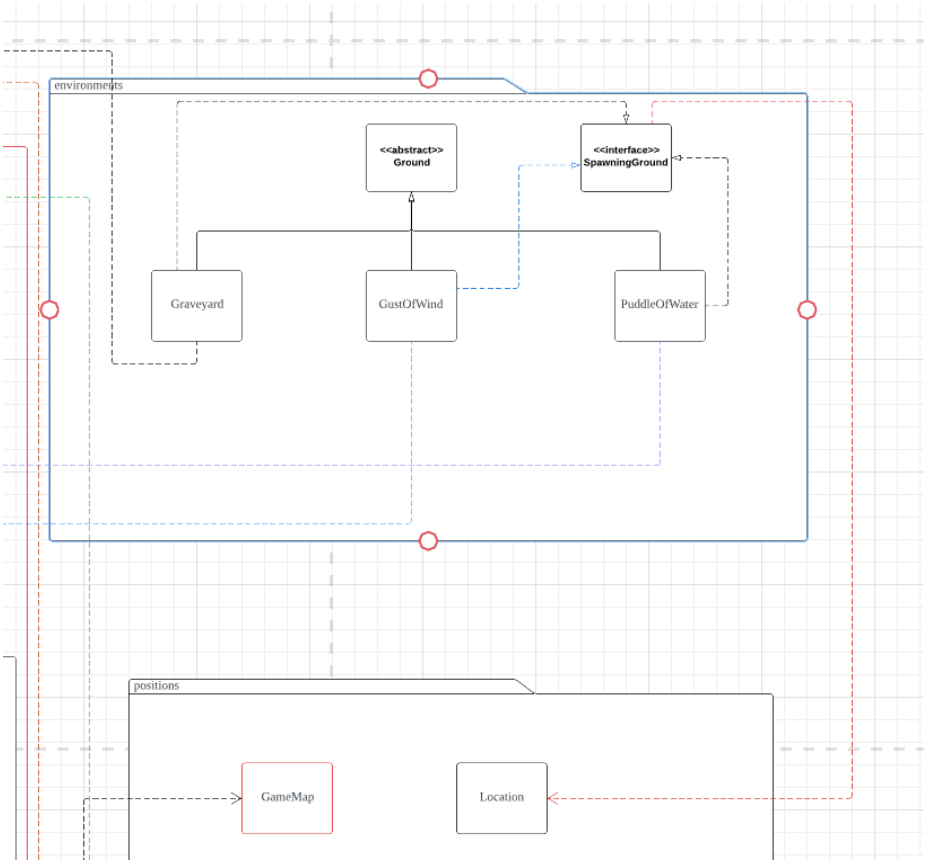
Black Color Font represents the design rationale which does not change in Assignment 1.

Red Color Font represents the extra explanations and modifications from Assignment 1.

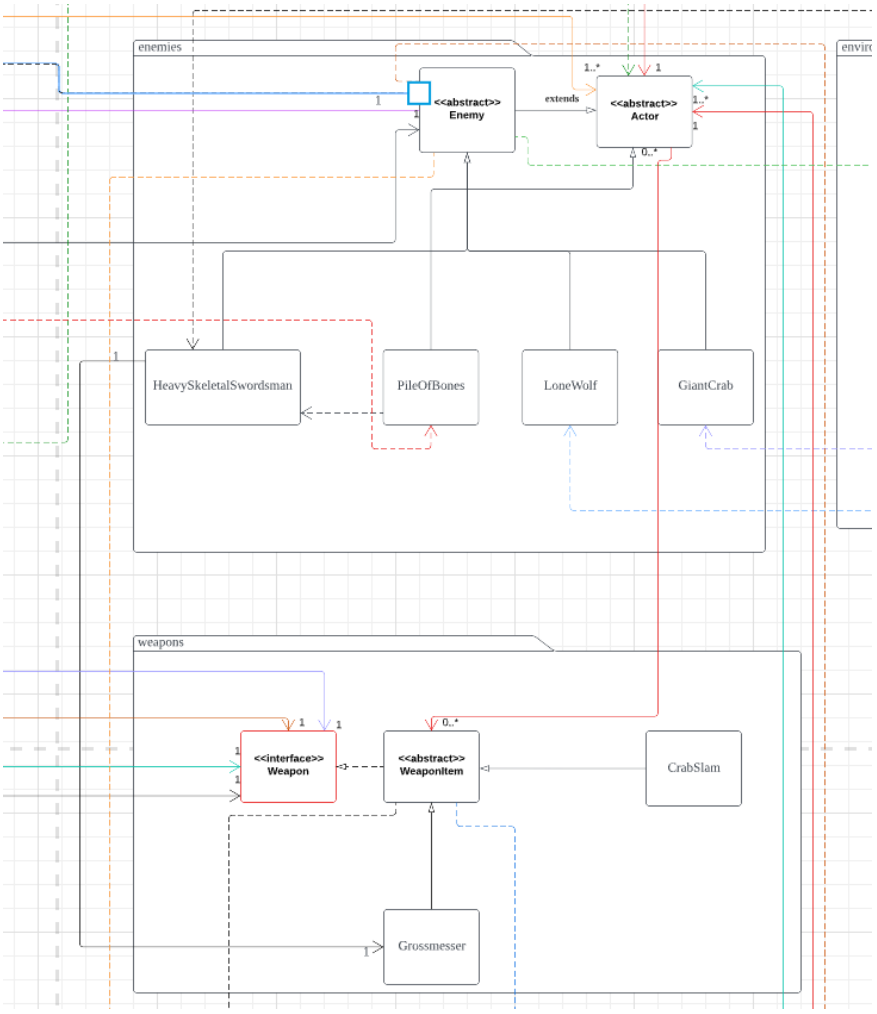
Requirement 1



The diagram represents an object-oriented system for different environments, enemies, weapons, actions and behaviour. The are 3 environment classes extend a base Ground class, 3 enemies class extend a base Enemy class, a weapon extend a base WeaponItem class, 4 behaviour class extend a base behaviour class and 4 actions class extend a base action class. The design goal of Req1 is to **reduce code repetition**, achieve the **SOLID principle** and as much as possible.



1) All three types of grounds extended the abstract Ground class. Since they share some common attributes and methods, it is logical to abstract these identities to avoid repetitions (DRY). When adding new features to each of the sub classes, such as each ground can spawn different enemies, we do not need to modify the base Ground class to adapt the changes (Open-Close Principle). Since Graveyard, Gust of Wind and Puddle of Water can spawn enemies, an interface named SpawningGround is created. Each of the grounds that can spawn enemies should implement this interface, so that the abstract Ground class would not be a God class which handles everything and is hard to maintain (Single Responsibility Principle). The drawback of this approach is that the SpawningGround interface will have a dependency relationship with the location to indicate which location should the ground spawn the enemies.

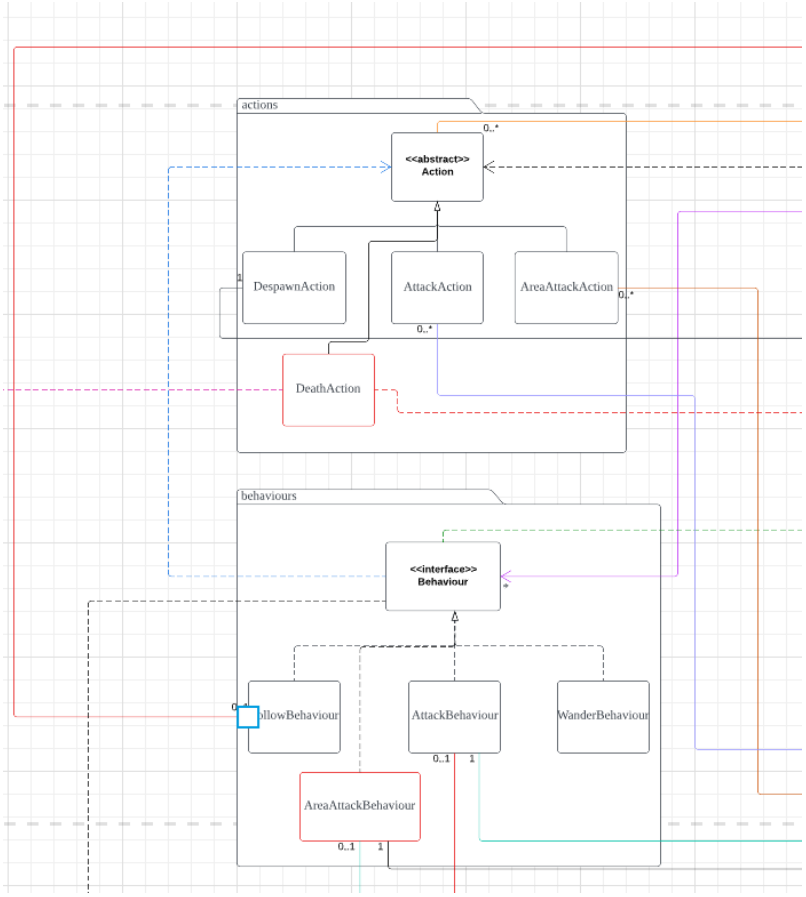


2) Create an abstract Enemy class which extends the actor class as base class for all enemies. Since Heavy Skeletal Swordsman, Lone Wolf and Giant Crab can attack the actors at their surroundings, we make them become concrete classes and extend Animal class to abstract the 3 concrete classes and avoid repetition of code (DRY). It also helps in improving the extensibility of code, for adding more enemy classes into the system. The Piles Of Bones class no longer extends the abstract Enemy class, as it cannot attack anyone and it will not drop runes when it is killed by the player, we instead make it to extend the abstract Actor class. The Piles of Bones will be revived to the Heavy Skeletal Swordsman with full health, if it is not hit within 3 turns, so we declare an integer class variable to 0, and the variable will be incremented by one at every play turn, after the variables has reached 3, Heavy Skeletal Swordsman will be revived. If the Piles of Bones is hit within 3 turns, then a Grossmesser will be dropped in the location where the Piles of Bones was hit. However, for this implementation, we would have duplication of codes in the Enemy abstract class and PileOfBones class in the allowableActions methods. This is the trade-off for making PilesOfBones not to extend the Enemy abstract class.

3) Enemy will select the highest priority behaviour, when they want to perform any action on their turn. We created a tree map in the Enemy abstract class. The area attack action has the highest priority followed by the single attack action, follow and wander around. This means that the enemy will always attack the player when the player is near (1 block away) from the enemy. We create public static final variables to represent each of the behaviour

priority, so that we don't have magic numbers to represent the priority which can be confusing, thereby avoiding the connascence of meaning **(CoM)**, this will help us to prevent inconsistency of values such as magic numbers and magic strings.

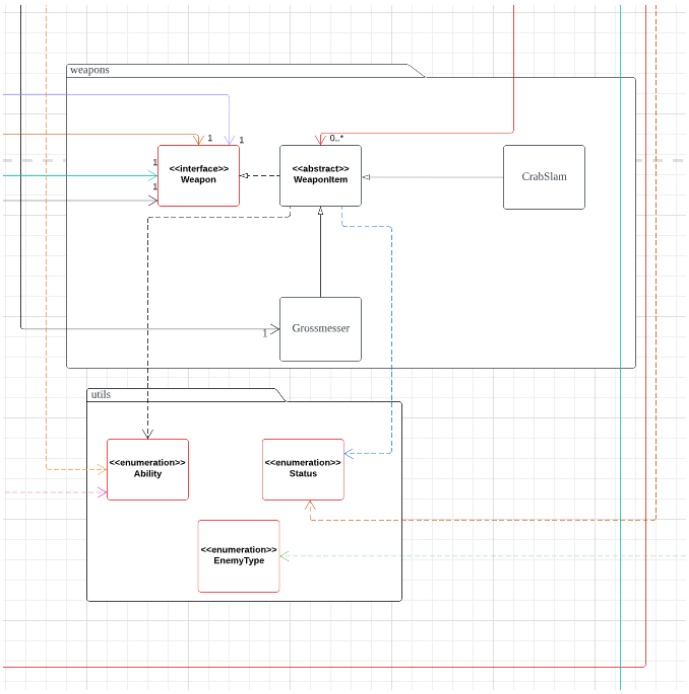
4) Grossmesser extends the abstract WeaponItem class as there can be many types of weapon instance in the future of the development of the system. Grossmesser <<uses>> ability. The Heavy Skeletal Swordsman has the Grossmesser as their weapon, so that Heavy Skeletal Swordsman also has the capability of Grossmesser's unique skill. As the Grossmesser can perform area attack, we create an enum class to store the Grossmesser's special skills which can perform an area attack, so that the enemy or player who owns the Grossmesser has the capability of performing an area attack. By doing so, we can avoid using if-else statements to check whether the weapon class has this unique skill. This design can help us to achieve the (Reduce Dependency Principle).



5) Each action extends the abstract base Action class as they share the common attributes and methods (DRY). They can override the execute method in the base class to implement their own logic. For the AreaAttackAction, since the area attack attacks all the actor in 8 exits, so it will first add all the actors in an ArrayList of target that can be attacked by the attacker, in the execute method, the probability that the attacker attacks each target is independent, so that target A may get hit while target B may not get hit by the attacker. For the death action, when the Heavy Skeletal Swordsman died, it will become Pile of Bones, so by using the enum and capability, we can check whether the actor in the death action has the ability to become bones when die, so that the Heavy Skeletal Swordsman can be replaced by the Piles of Bones in its location. However, this approach will introduce extra

dependencies between DeathAction and PileOfBones, but we can implement the logic of PilesOfBones correctly.

6) Each behaviour implements the Behaviour interface, so that each behaviour class should implement the method defined in the interface as each behaviour has different implementation logic (Single Responsibility Principle). Different behaviour has different input parameters, when creating the behaviour class, for example AreaAttackBehaviour and AttackBehaviour has two constructor, one requires the Weapon parameter, whereas the other does not require the Weapon parameter. We create two constructors that, as the enemy can have no weapon in its inventory, so that they will use the intrinsic weapon to attack the other actor, else the enemy will use its first weapon to attack the other actor. We separate the AreaAttackBehaviour and AttackBehaviour into different classes, because these two behaviours have different implementation logic, so that each class only has a **single responsibility (SRP).**

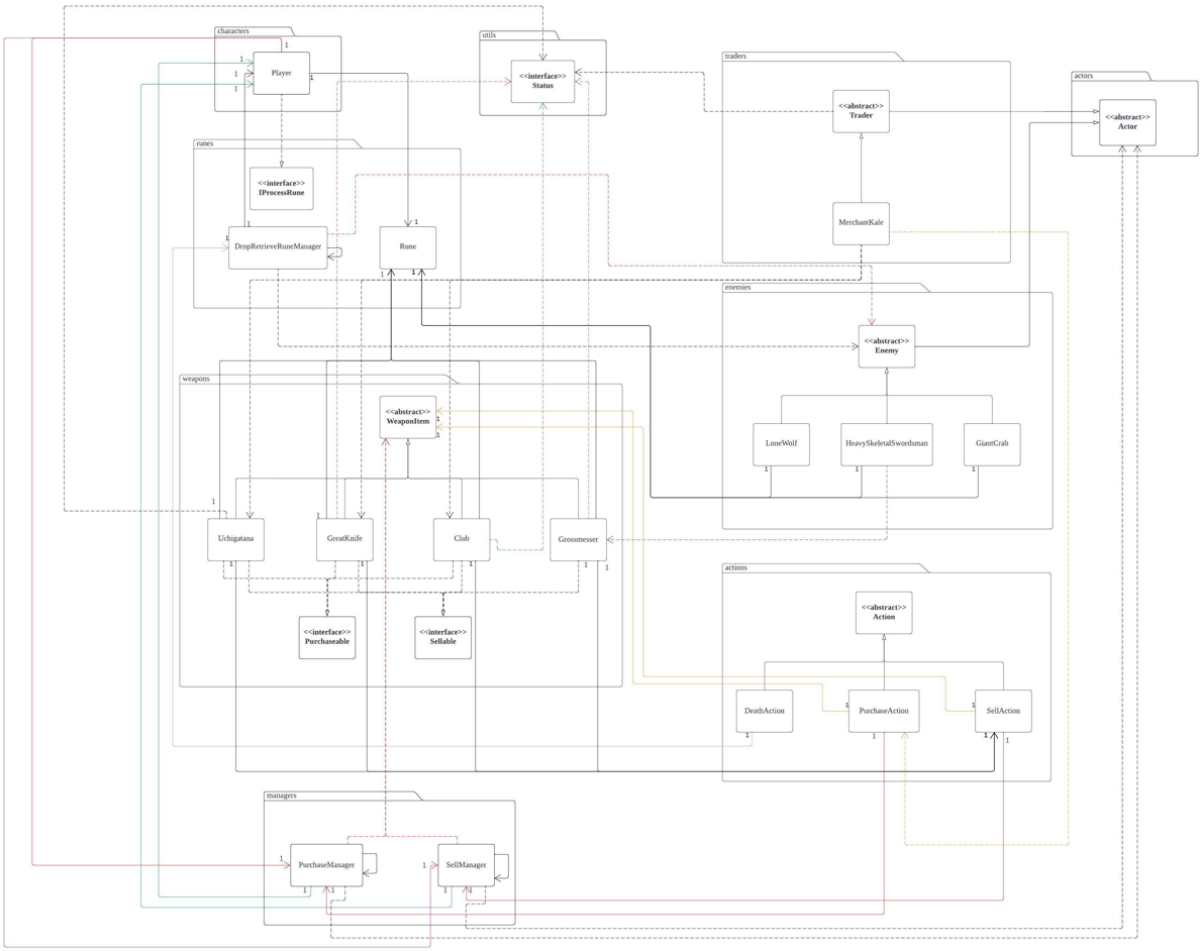


7) Two enum classes created namely Status and EnemyType. Status enum class is for checking for the status of the actor, for example, player the Status.HOSTILE_TO_ENEMY, which means that every enemy can attack the player. We can use this concept to implement that the enemy will not attack their type unless an area attack action is performed. Ability enum class is used for the unique skill such as the Grossmesser can perform an area attack action. By using this approach, we can achieve the Single Responsibility principle (SRP), where each class only has one responsibility.

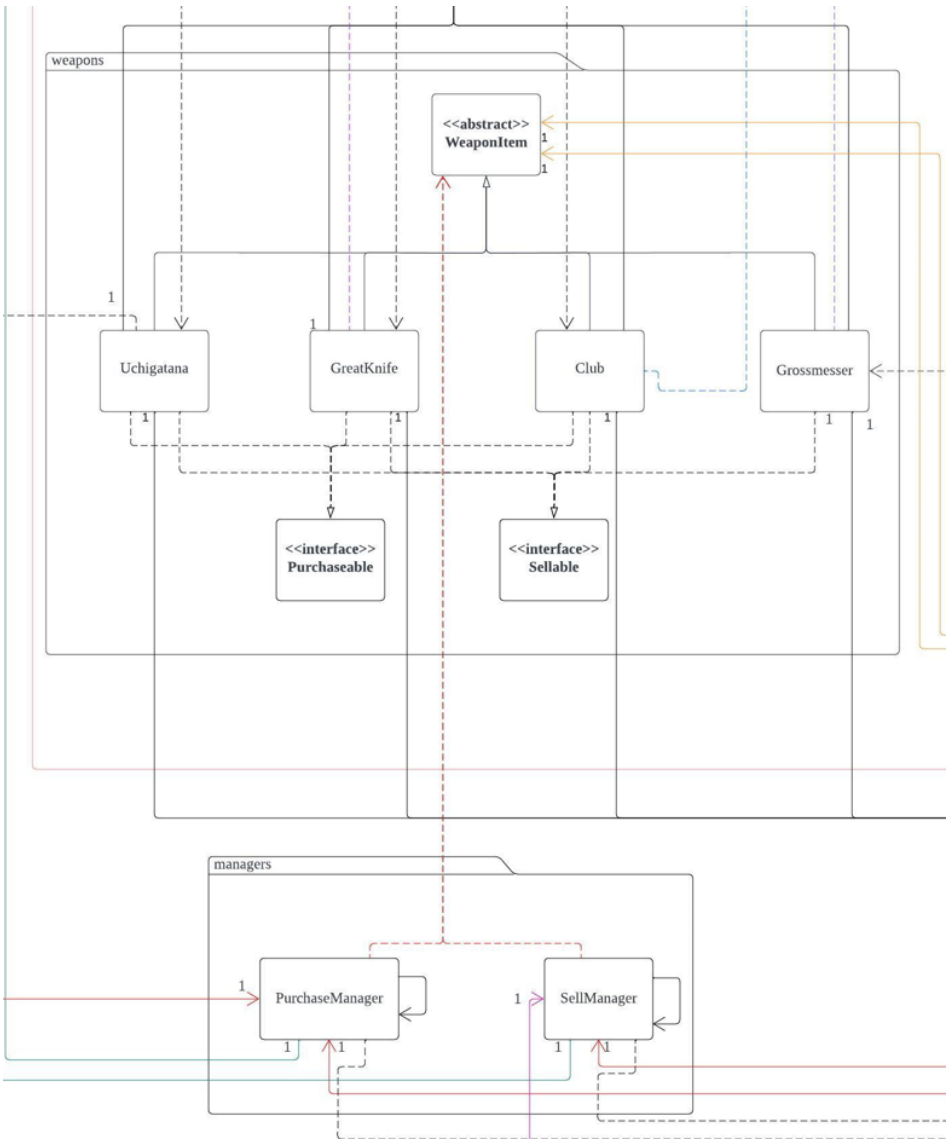
8) For the implementation of the enemy and player allowableActions method, we have used downcast to implement the behaviour of the enemy. We know that this does violate the SOLID principle as we have used downcast to add the behaviour to the particular enemies. However, before downcasting, we will check that the actor is an Enemy by using the enum class, so in this situation we can avoid any of the run-time errors, because we make sure we only downcast the actor to the enemy by checking the actor is essentially the type of enemy.

We have tried to avoid downcasting by implementing another logic, but the implementation does not work well for many situations. For example, when there is a LoneWolf in a gamemap, and it has GiantCrab on its east. So we first process the LoneWolf by getting the allowableActions in the GiantCrab, since they are not the same type of enemy, the LoneWolf should attack the GiantCrab. But in this implementation, we can't add directly to the behaviour of LoneWolf since we are getting the allowableActions in GiantCrab class, so there is no attack behaviour added in the list of behaviours in LoneWolf, as a result, the LoneWolf will not attack the GiantCrab. Since, we are in the allowableActions in GiantCrab class, we will just add the attack behaviours into the GiantCrab class as we know the surrounding of GiantCrab has a LoneWolf, so GiantCrab can attack LoneWolf. However, this implementation has a very serious problem. The list of behaviours of LoneWolf do not have the attack behaviour, so the LoneWolf will just wander around and it can move away from the surroundings (8exits) of the GiantCrab. For the turn of the GiantCrab, since it already has the behaviour of attacking the LoneWolf, it will just attack the LoneWolf even if the LoneWolf is not in one of its 8 exits. In short, we has come over a conclusion that we will prefer that the application is working well rather than the application that follow the SOLID principles but does not work well, so that we used downcast to implement the logic, but as a trade-off we violates the SOLID principles

Requirement 2

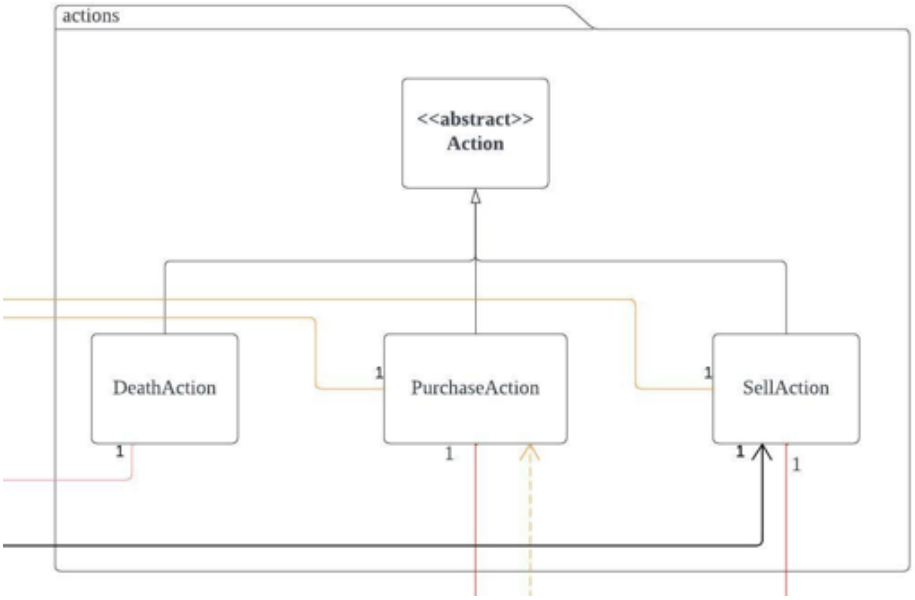


The diagram represents an object-oriented system for a rogueslike game. We created 4 types of weapons that extend an abstract WeaponItem class with Purchasable and Sellable interface as well as PurchaseManager and SellManager that will help in the transaction of weapon with trader, 3 types of enemies that extend an abstract Enemy class, 3 types of action that extend abstract Action class, a trader that extends abstract actor class and a merchant kale that extends trader class, lastly three classes that manages the rune for the game which are Rune, IProcessRune and DropRetrieveRuneManager classes. The design goal of REQ4 is to reduce code repetition, dependency and achieve SOLID principle and as much as possible.

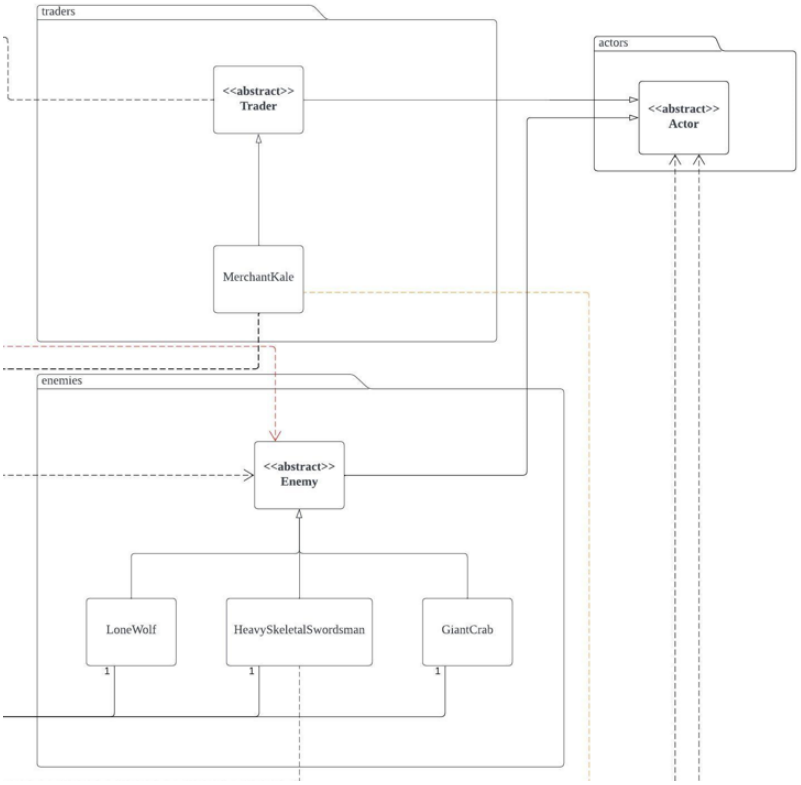


1. All four types of weapons namely, Uchigatana, GreatKnife, Club and Grossmesser extended the abstract WeaponItem class. Since they share some common attributes and methods, it is logical to abstract these identities to avoid repetitions (DRY). Grossmesser extends the WeaponItem class and is the only weapon that can be carried by a Heavy Skeletal Swordsman. For every weapon that can be purchased by player or sold to trader, the weapon will implement Purchasable interface and Sellable interface, thus the weapon isn't directly accessed by any actor classes. From the diagram we can also see that there are two managers which are PurchaseManager and SellManager, which manage all the buy and sell actions in the game. If the player intends to purchase a weapon, PurchaseManager will check if the player rune amount is sufficient to buy the weapon, otherwise it will return false. Both managers also will update the player's current rune amount and current weapon on hand. Besides, in both managers class, we create a static factory method and make the constructor private ensuring that there is only one instance of the PurchaseManager and SellManager classes as external code could not create an instance of both classes using the public constructors (singleton). Since we are not required to create a new SellManager and PurchaseManager object each time it is invoked, thus reducing the verbosity of creating parameterized type instances. Not all weapons are purchasable or sellable, hence by using

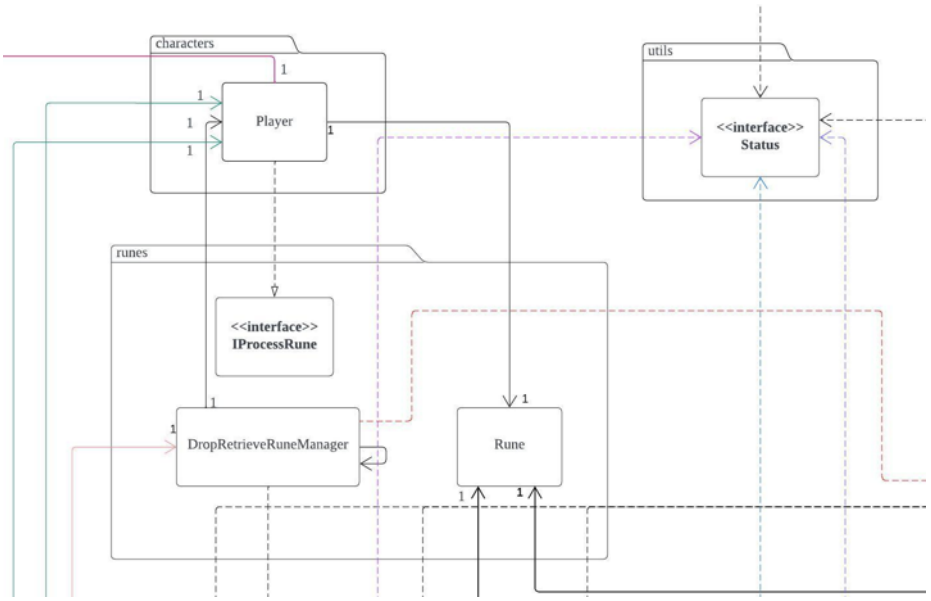
the Purchasable and Sellable interface, we can determine it easily through both interfaces, and visualise it instantly through UML diagram.



2. All three types of actions extended the abstract Action class. Since they share the common attributes and methods, it is logical to abstract these identities to avoid repetitions (DRY). There are two classes, namely PurchaseAction and SellAction which extend Action abstract class. These two classes are applicable to a trader class only. However, we do not include it into our trader class as we don't want to make the Trader class a GOD class. This will allow us to add more action features in Action class in future development without having to modify the Trader class (open-closed principle) as well as ensuring every class handles a single part of the program functionality (single responsibility principle). MerchantKale, which is a child class of Trader class, depends on PurchaseAction. MerchantKale creates a PurchaseAction for each weapon that is purchasable and added into the allowableActions, so that the player can purchase the weapon when the trader is in the surroundings of the player. Thus, in future development, if we intend to create a different trader that comes with different actions, we can decide whether to add the PurchaseAction, SellAction etc into it easily, making the code more flexible and extensible.



3. Abstract actor class is a base class for trader class and enemy class. Enemy as an abstract class extends actor abstract class. It's logical to abstract these identities to avoid repetitions because they have similar attributes and methods. Thus, making it more manageable and organised. Every enemy has dependencies on DeathAction when they are dead. Hence, we do not need to modify the base class to adapt the changes which obey the open-close principle. Besides, each enemy can drop runes when they are dead, hence we create several rune related classes to manage runes instead of cramming everything inside the enemy class, to prevent it from being a GOD class.

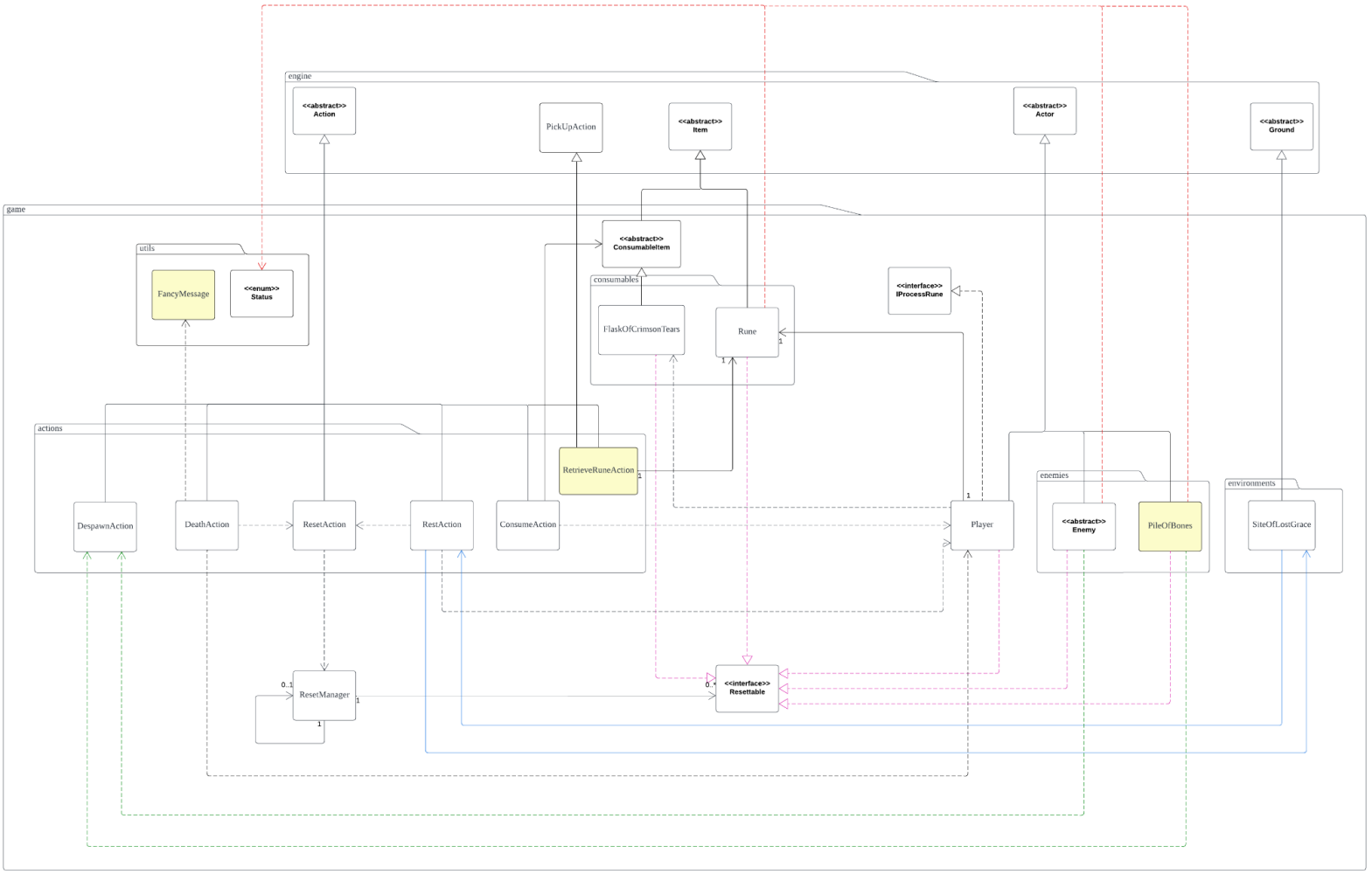


4. In our rune package, there are three classes which are IProcessRune interface, Rune and DropRetrieveRuneManager. IProcessRune is only used by the player, which has a method that can get and set the amount of rune, yet we make it interface so that it doesn't directly access the rune amount inside rune class. DropRetrieveRuneManager handles all the rune drop and retrieval when the enemy is dead. Since, the player has the ability to retrieve the runes dropped by the enemies, when the player kills them. Therefore, we create an interface class which is IProcessRune and make the player implement this interface. The DropRetrieveRuneManager is responsible for managing the runes dropped by the enemy when the player kills it which adds the runes dropped into the balance amount of the player. Besides, in DropRetrieveRuneManager we create a static factory method and make the constructor private ensuring that there is only one instance of the class as external code could not create an instance of the class using the public constructors (singleton). Since we are not required to create a new DropRetrieveRuneManager object each time it is invoked, thus reducing the verbosity of creating parameterized type instances. Lastly, Rune class is basically the one that records the display and amount of runes. Instead of cramming everything into a Rune class making it a GOD class, we split it into three relevant classes, which is much easier to maintain (Single Responsibility Principle).

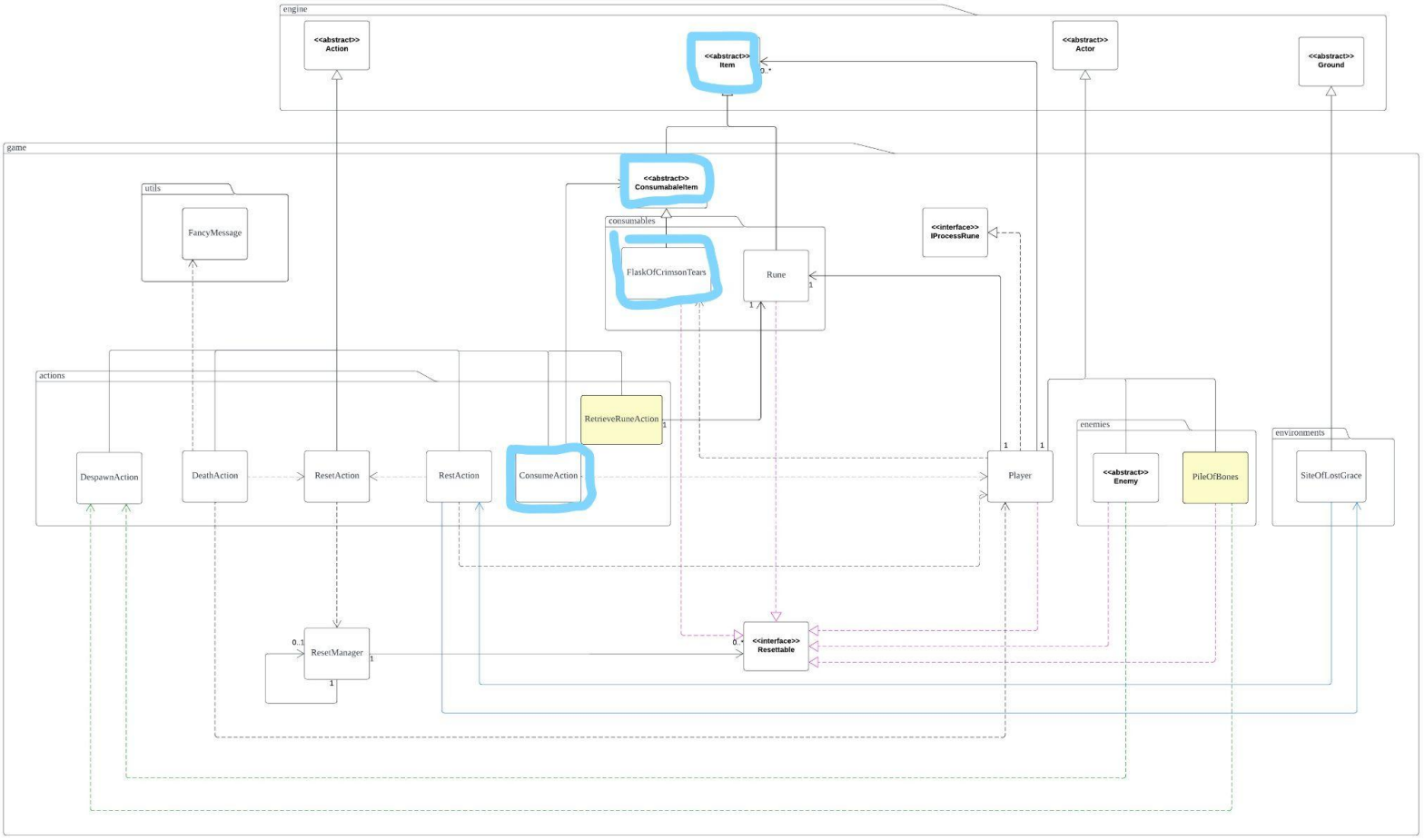
5) Since we do not want to use downcast in our implementation, so for each weapon that is sellable which implements the Sellable interface, we check if the surrounding of the actor holding this sellable weapon contains a trader or not. If a trader exists, we then add a sell action in the allowable actions in this weapon class, else we remove this action from the allowable actions. For this approach, we would have duplicate codes in the sellable weaponitem classes but this approach allows us to not use downcast to in the implementation logic, this approach is the trade-off between downcast and duplicate codes.

6) We create public static final constants to represent each of the purchase cost of the purchasable weapon in the MerchantKale class. In this way we can avoid using magic numbers to represent each purchase cost and eliminate any inconsistent values thus avoiding connascence of meaning (CoM).

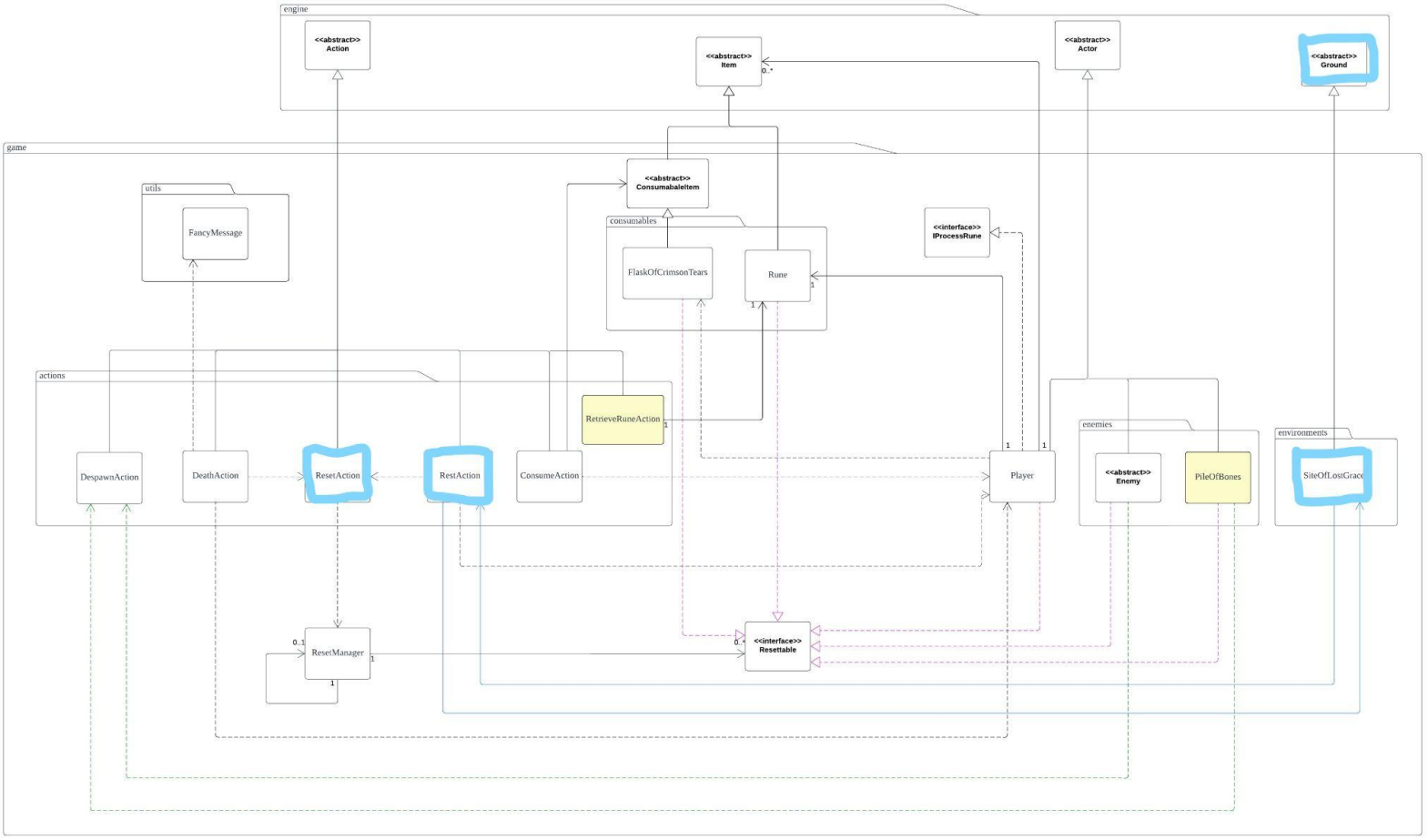
Requirement 3



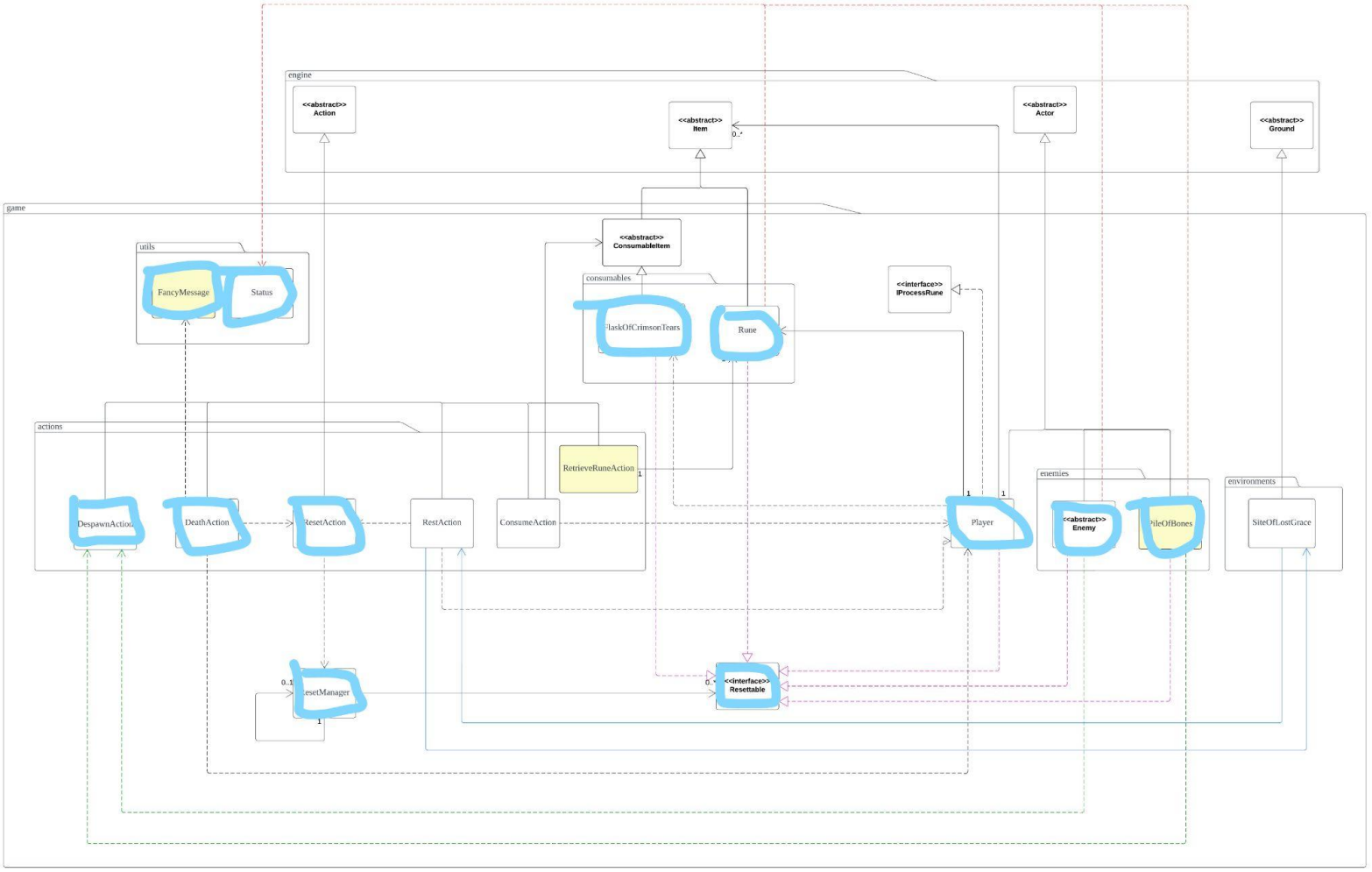
Requirement 3 includes a new item - Flask of Crimson Tears, a new ground - Site of Lost Grace, the game reset functionality and how the runes operate during reset. The diagram represents an object-oriented system that aims to fulfil all the requirements while achieving the SOLID principles and reduce code redundancy.



For the Flask of Crimson Tears, I created a new class for it which extends the abstract ConsumableItem class. The ConsumableItem is an abstract class which extends the item class where all consumable items will extend from. This fulfils the **don't repeat yourself principle** as all the attributes and methods of a consumable item can be extracted into a class. This design can be easily extended as if a new consumable item is added to the game, it just needs to extend the ConsumableItem class to work. I decide not to implement the Consumable interface as stated in assignment 1 as I think interfaces are good for providing a common functionality to unrelated classes but for this scenario an abstract class would be more appropriate as the consumable items can be considered as closely related classes. I also did not include the ConsumeManager class that was stated in assignment 1 as I decided to manage the consumable items using the ConsumeAction class. This design may be reasonable for now as there is only one consumable item but it may fall when more consumables are added to the game as different consumables might have different effects when being consumed. The better way to implement this while more consumables are added should be to create a Consumable interface and ConsumeManager class which manages all the consumables. This design fulfils the **open-closed principle** because the code can be easily extended if extra consumable items are added to the game but closed for modification. Overall, this fulfils the **single-responsibility principle** where each responsibility is distributed to different classes.

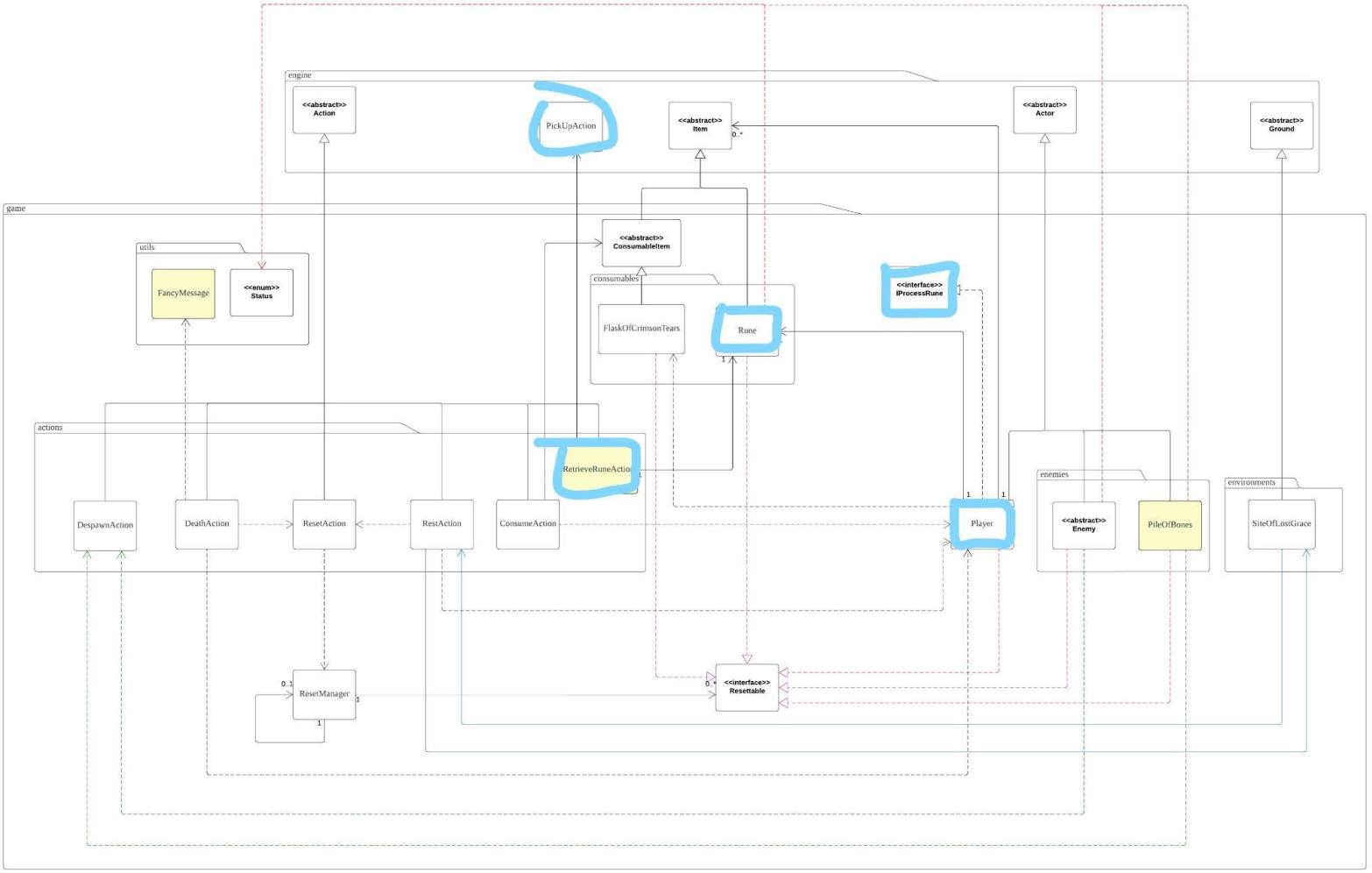


For the Site of Lost Grace, I created a new class for it that extends the abstract ground class to fulfil the **don't repeat yourself principle** as it uses all the ground class code so no repetition will occur but the rest functionality will have to be added to it. To implement the rest function, it has a dependency with the RestAction class which extends the Action class. This fulfils the **single-responsibility principle** where the rest function is separated from the SiteOfLostGrace and it doesn't become a god class with lots of code. And lastly, it has a dependency with the ResetAction which allows it to implement the reset functionality. The SiteOfLostGrace can be extended in the future where each site has its own name which is stored in an attribute. Besides, other actions other than resting might also be allowed in these sites, for example travelling to another map through these sites. The pros of this design is that everything is neatly separated into a different class and each class manages its own functionality while the cons is that it might be overwhelming for others as there are many classes created.



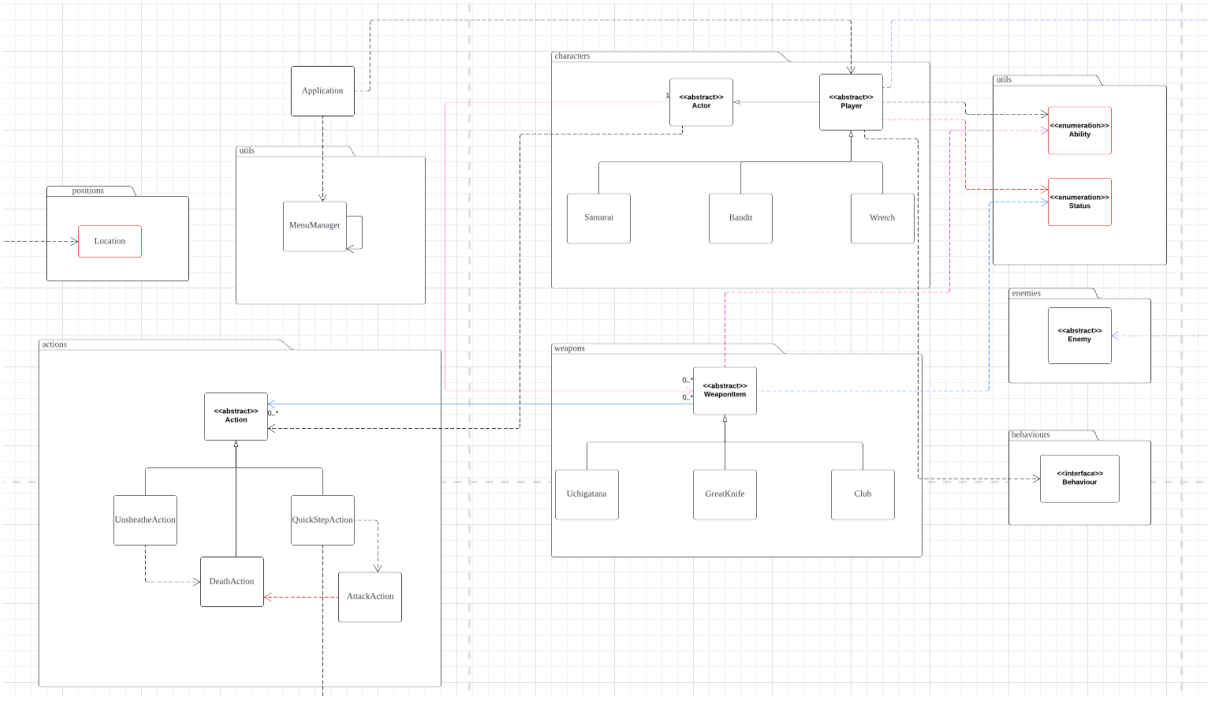
For the game reset functionality, a Reset Action class which extends the action class was created to reduce repetition and fulfil the **don't repeat yourself principle**. A resettable interface where all resettable things such as FlaskOfCrimsonTears, Player, Enemy, Rune and PileOfBones implements is created. This design fulfils the **interface segregation principle** where only resettable things implement the interface. A ResetManager is then created to store and manage the resettable things. The ResetManager has an attribute which stores the instance of itself. This is called the **singleton** pattern which ensures that a class has only one instance, while providing a global access point to this instance. The pros to this design is that the class will only have a single instance that we can gain global access point to and the singleton object is only initialised once when it is requested for the first time. While the cons are that it violates the single responsibility principle as it solves two problems at the same time. This design fulfils the **open-closed principle** where it can be easily extended if there are more resettables added to the game, it just needs to implement the resettable interface and apply the logic in the reset method. For the despawning of enemies, I initially (assignment 1) wanted to implement a despawnable interface and despawnable manager. After coding it, I realised that it is unnecessary and will just overcomplicate things. To implement the despawning, I created a despawn action which will be run if the enemies and pile of bones have the DESPAWNABLE status in the status enum. The DESPAWNABLE status will be added to them when the game resets. This can be easily extended if additional actors require despawning, they just have to add the DESPAWNABLE status during the

reset and implement it in their playTurn method. Lastly, the DeathAction has a dependency with the ResetAction and player to implement the game reset functionality when the player dies. It also uses the FancyMessage class in utils package to print the YOU DIED message when a player dies.

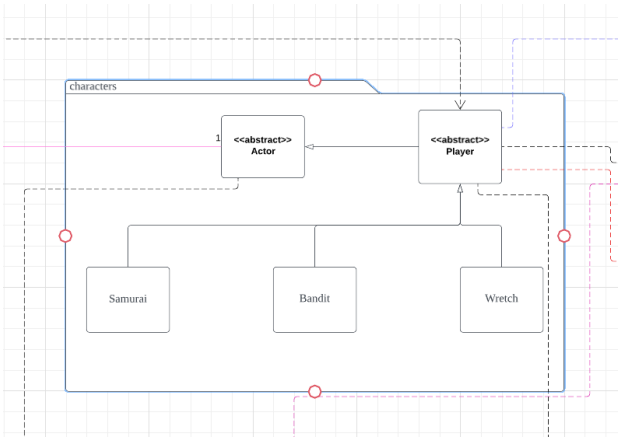


To implement the dropping of runes I implemented the logic inside the Rune class using the reset and tick method. This is done because the rune is not in the player's inventory and has to be removed from the map using this way. To get the dropped rune's amount, the player class implements the IProcessRune which has the method to get and set the rune's amount. For picking up the dropped runes, I created a RetrieveRuneAction that extends the PickUpAction and is used to override the PickUpAction method in the Rune class. This is done to ensure that the rune is picked up by the player but not added to the player's inventory because if it is in the player's inventory, due to it being portable we will be prompted each time whether we want to remove it from our inventory. The logic of adding the amount of runes to our current amount is implemented in the RetrieveRuneAction. The pros of this design is that most of the logic is implemented internally in the Rune class while the cons is that it may become a hassle to debug if it has errors. I initially (assignment 1) wanted to create a RuneManager class that stores and manages the runes but I feel like it is unnecessary as I can implement it inside the rune class and reduce the complexity of the game by excluding extra classes.

Requirement 4

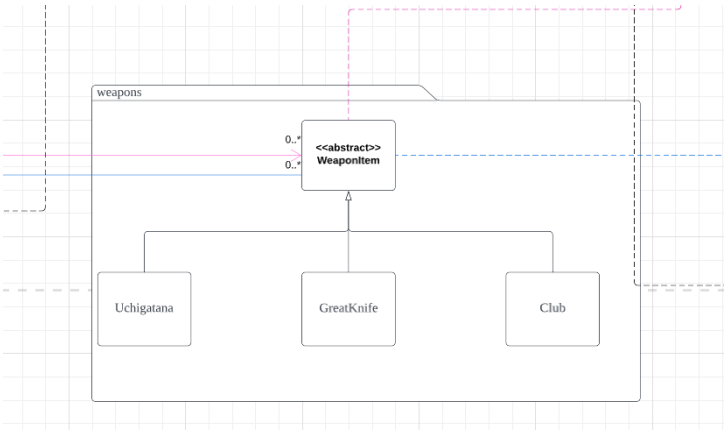


The diagram represents an object-oriented system for different characters and weapons. The 3 character classes extend a Player class and 3 weapon classes extend a base WeaponItem class. The design goal of REQ4 is to reduce code repetition, dependency and achieve SOLID principle and as much as possible.

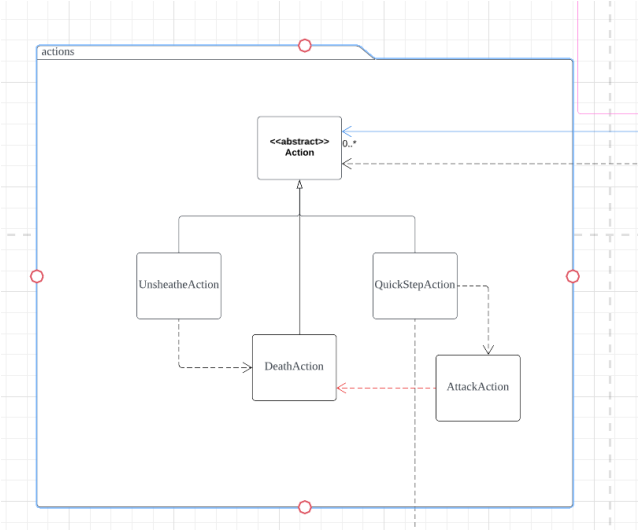


1) All three types of characters extended the abstract Player class. Since they share some common attributes and methods, it is logical to abstract these identities to avoid repetitions (**DRY**). When adding new features to each of the sub classes, we do not need to modify the base Player class to adapt the changes, we can extend the feature of each subclass by adding new methods without affecting the base class (**Open-Close Principle**). By making the Player as the abstract class which extends the abstract Actor class, we make each character extend the Player class, so that we comply with the **Open-Closed Principle**, by

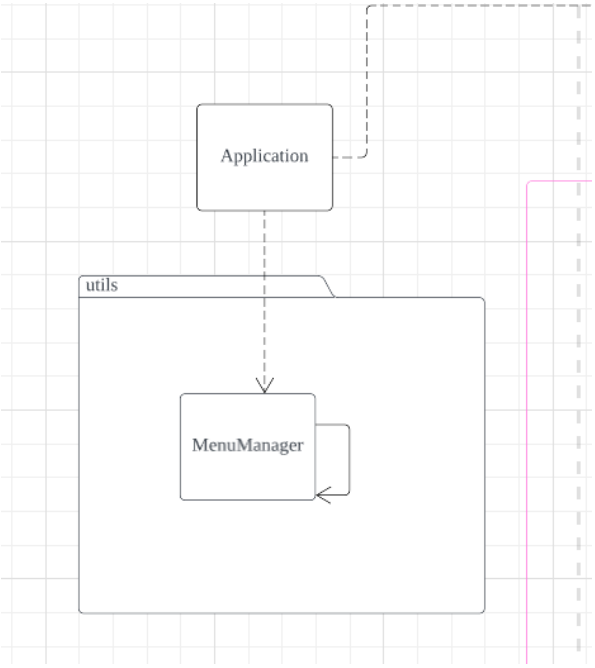
following the design we make our code easier to extend and maintain. We achieved this by using abstraction which is the abstract Player class.



2) All three types of weapons extended the abstract WeaponItem class to avoid repetition of code (**DRY**). It also helps in improving the extensibility of code, for adding more weapon classes into the system. Each weapon has a different unique skill, so we can create an Enum class to store the valid skills. We can then easily store these unique skills in the actor class as capabilities. By doing so, we can avoid using if-else statements to check whether the weapon class has this unique skill. This design can help us to reduce the dependencies among classes. Some of the weapons have different unique skills, so that we create an enum to store all the valid skills. This approach allows us to avoid **connascence of meaning**, because we would not have any magic strings that represent the unique skill of the weapon. We create an enum class to store each of the special skills of the weapon **(Single Responsibility Principle)**. The weapon Uchigatana has a unique skill which deals 2 times damage with 60% accuracy, so that we add the unique skill in the Ability enum class for Uchigatana, so that when a player is holding this weapon, the player has the option to use this unique skill. This works the same for Great Knife which also has the special skill. For the Great Knife unique skill which is to move the user away from the enemy, evading their attack. The user will move to one of its 8 exits, if the exit has no actor on it, so that we cant have 2 actors standing on the same location as per the requirements. We create this enum class to store the valid special skill of each weapon so that we can avoid using instanceof which add more dependency between classes, thereby achieving the **Reduce Dependency Principle.**



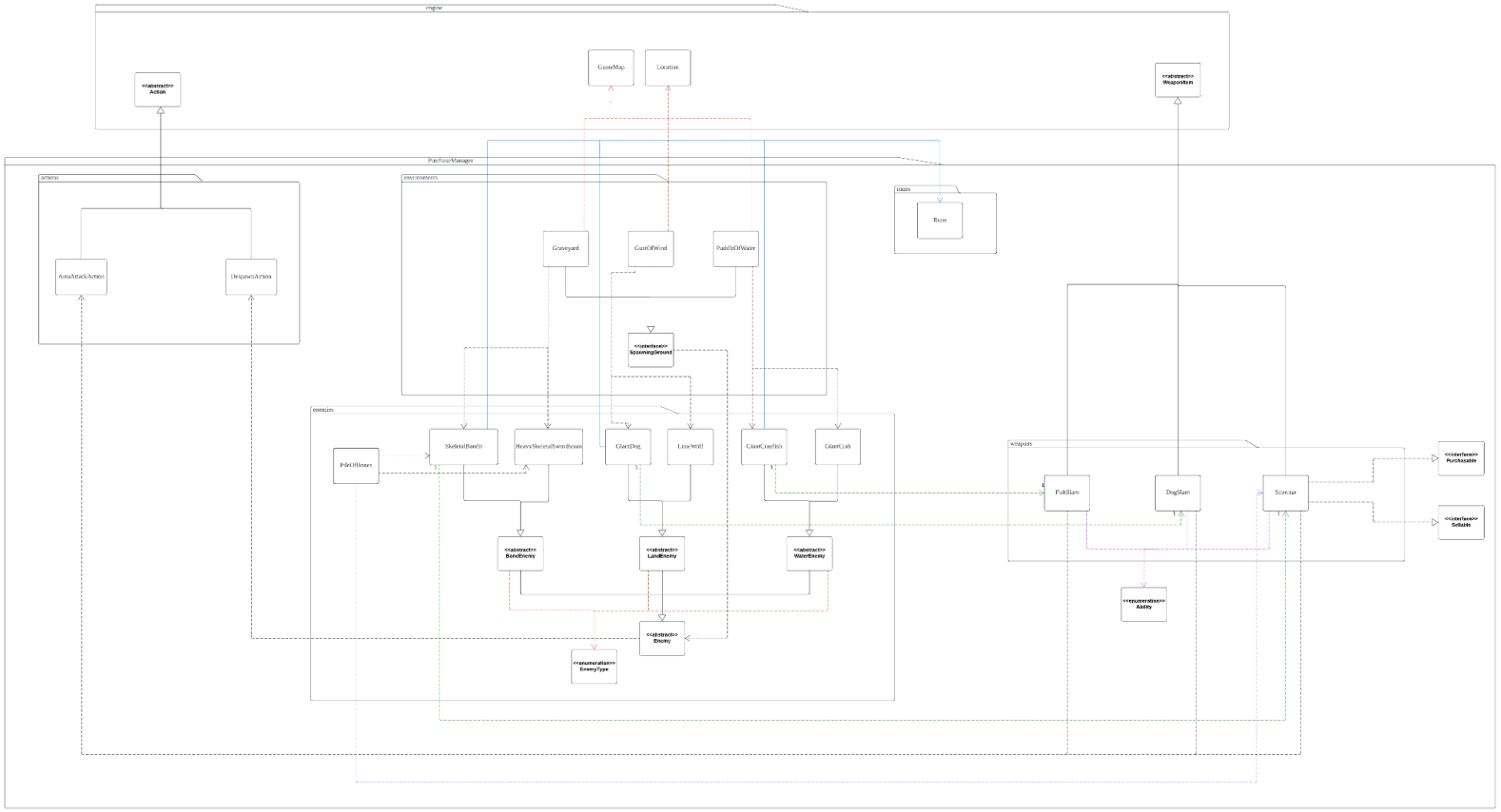
3) For the QuickStep action which is the special skill of a great knife, the actor holding this weapon can attack the other actor and move to one of its 8 exits if there is no actor on that particular exit. Since QuickStep action also deals damage to the other actor, we will reuse the AttackAction to perform the attack logic, so that we do not need to repeat the same code in QuickStep class, so that we follow the Don't repeat yourself principle. The disadvantage of this approach is that it will add dependency between QuickStepAction and AttackAction.



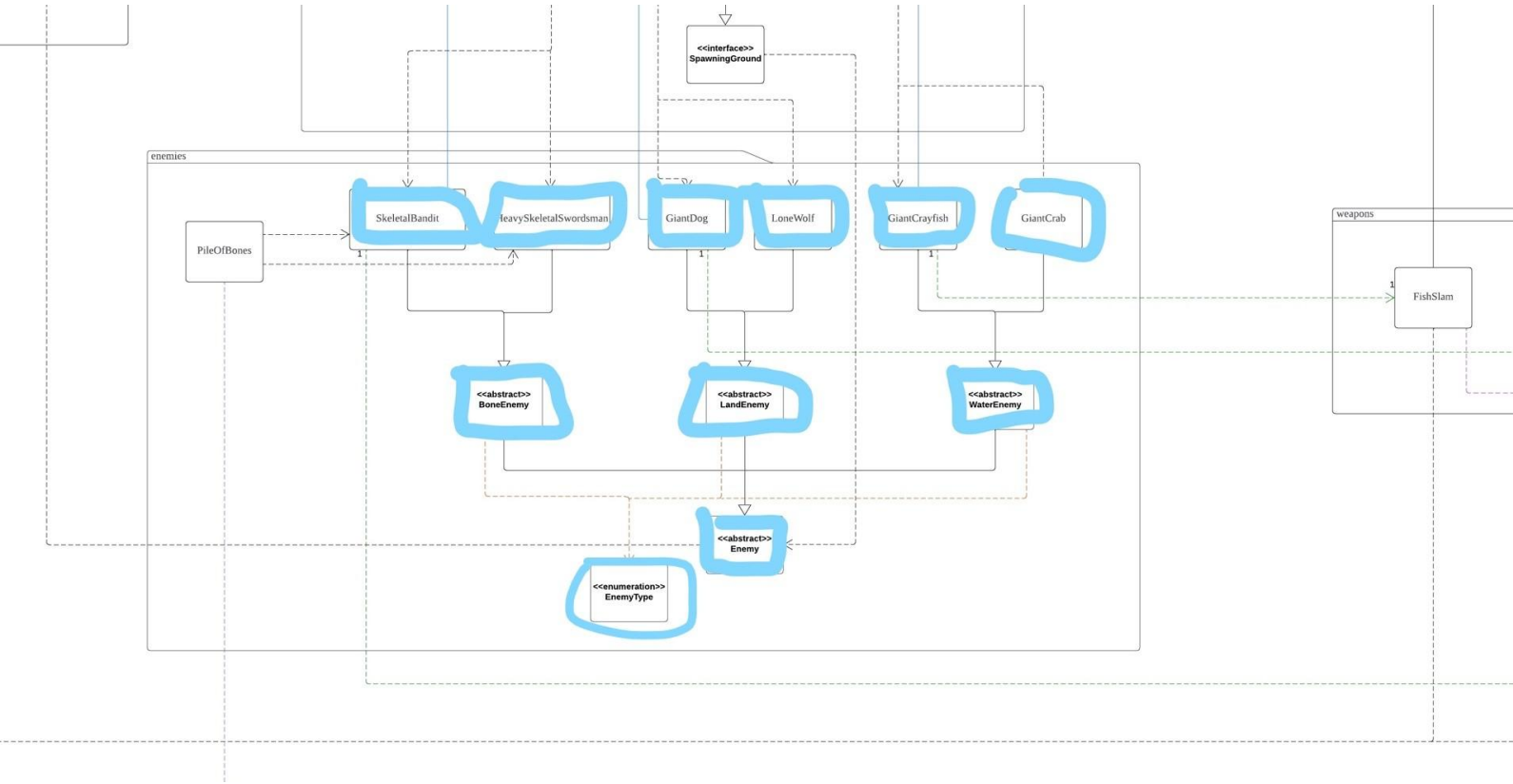
4) We create a MenuManager class which has only a method which allows the user to select the characters in the game engine. We make this a separate class from the World class because we do not want the World class to be the GOD class which handles anything and makes the engine hard to maintain. Therefore, each class should have one responsibility which comply with the (**Single Responsibility Principle**). For the MenuManager class which is used for the user to select the character, we create a static factory method and we make the constructor private. This would prevent any external code from creating an instance of the MenuManager class using the public constructors **(Singleton)**. We are not

required to create a new MenuManager object each time it is invoked, thus reducing the verbosity of creating parameterized type instances.

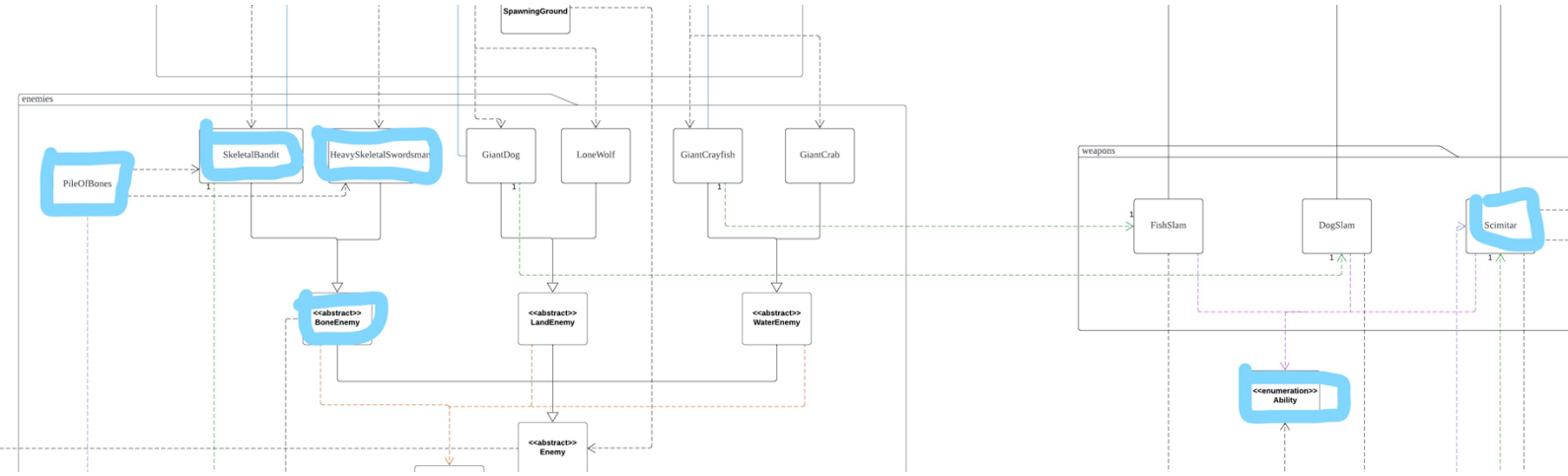
Requirement 5



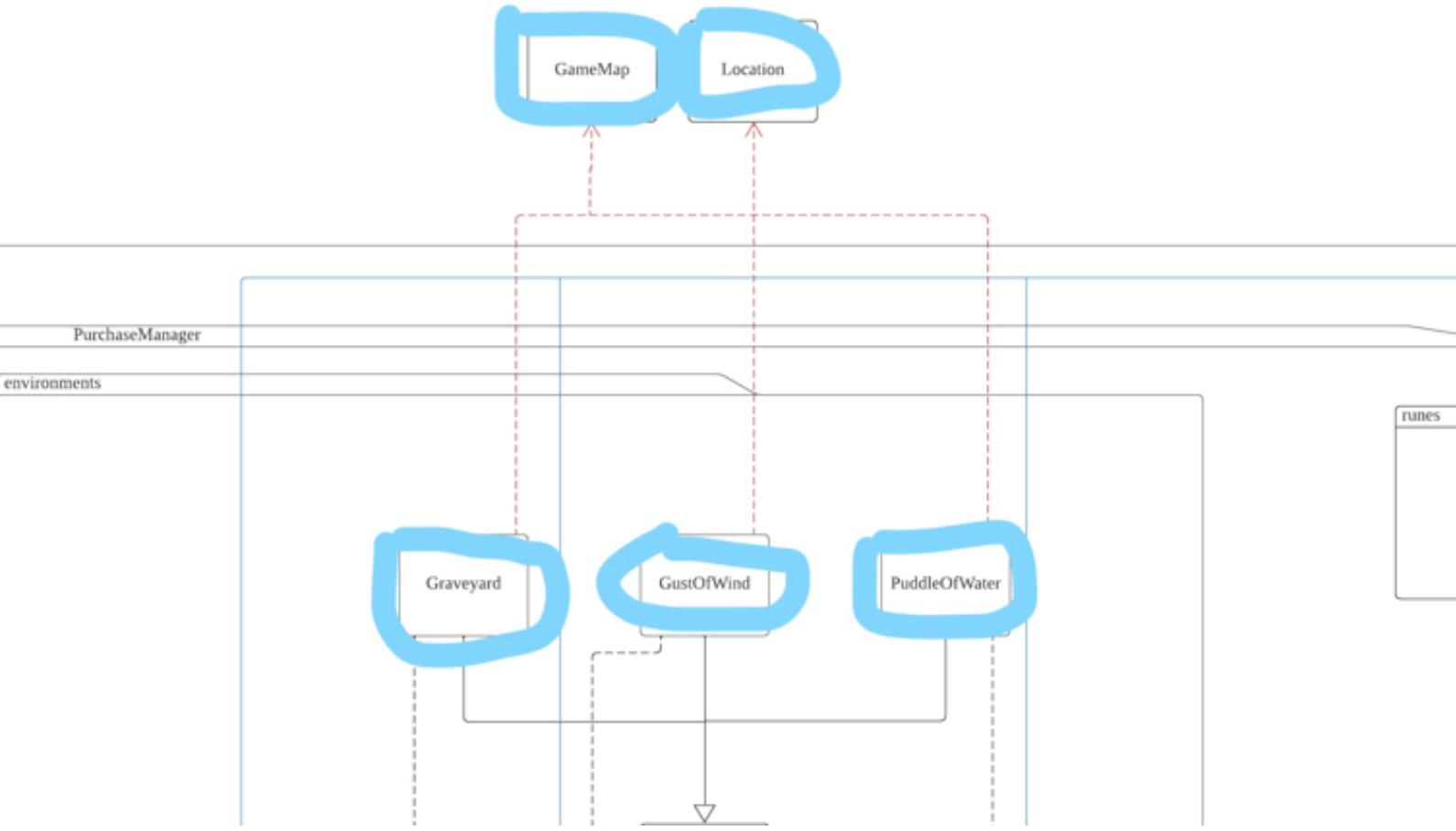
Requirement 5 includes some extra enemies (Skeletal Bandit, Giant Dog, Giant Crayfish) by splitting the map in half and setting specific enemies to spawn at each sides of the map and a new weapon - Scimitar which is held by the Skeletal Bandit and can also be purchased from Merchant Kale. The diagram represents an object-oriented system that aims to fulfil all the requirements while achieving the SOLID principles and reduce code redundancy.



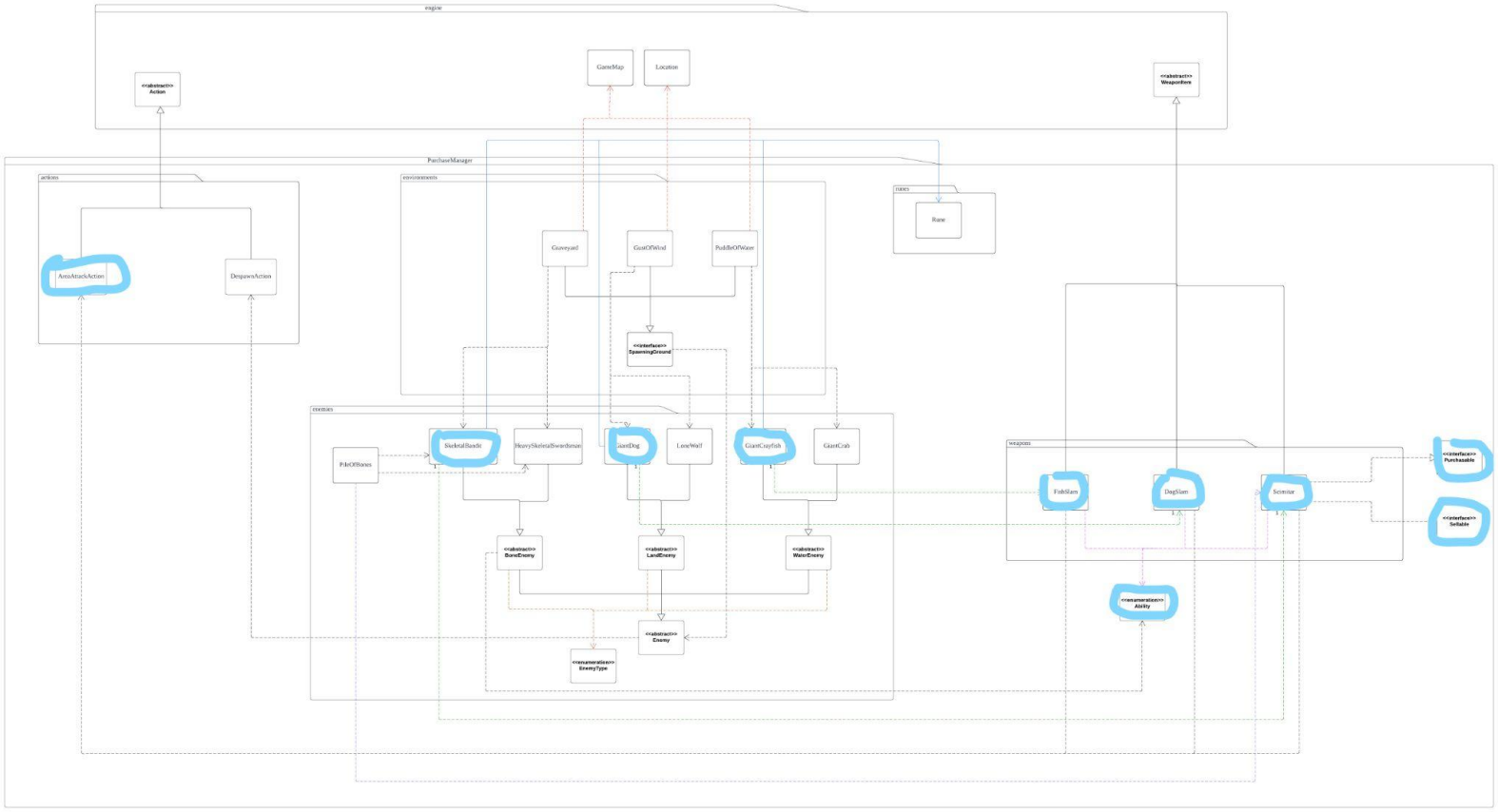
Due to the additional enemies, some abstract classes (BoneEnemy, LandEnemy, WaterEnemy) were created to better differentiate and group the enemies by type. These classes extend the Enemy class. The SkeletalBandit and HeavySkeletalSwordsman extends the BoneEnemy, GiantDog and LoneWolf extends the LandEnemy while the GiantCrayfish and GiantCrab extends the WaterEnemy. This design reduces code redundancy and fulfils the **don't repeat yourself principle.** With this design, we can set the enemy type through the abstract classes. The enemy type is used to ensure that enemies of the same type do not attack each other. This implementation can be easily extended if other enemies are added to the game, they just have to extend the abstract class of their specific type for it to work. The pros of this design is that it can be easily extendable if enemies of the similar type are added but the cons is that if enemies of different types are added it can be difficult to extend.



To implement the SkeletalBandit and PileOfBones relationship, the PileOfBones has a dependency with both the SkeletalBandit and HeavySkeletalSwordsman. It then checks which side of the map it is on and spawn the respective enemy if it is destroyed. The PileOfBones also has a dependency with Scimitar to implement the dropping of it when the SkeletalBandit's pile of bones is destroyed. The abstract BoneEnemy class uses the Ability enum in order for both its child classes to turn into PileOfBones when killed.



To separate the map into east and west, we do it in a different way such that we do not create an abstract EnemyFactory class and make two child class to extends the EnemyFactory class which are EastMapEnemyFactory and WestMapEnemyFactory class as suggested in the feedback for A1. Instead we only use the getXRange().max() method in the engine package which is in the GameMap class. In this way we can get the maximum x coordinates of the GameMap and just divide by 2 to determine the east and west of the GameMap so that we can easily spawn different types of enemies on different sides of the game map. In this implementation, we can avoid using extra additional classes, so that we can reduce the complexity of the applications by removing extra classes. The cons of this design is that we have to manually check which side of the map the ground is on and spawn the respective enemy which results in more if-else checks in our code.



To implement the additional enemies' weapons three additional classes (FishSlam, DogSlam, Scimitar) that extend the WeaponItem class were created. All of the weapons have an area attack hence each of them has a dependency with the AreaAttackAction class. Each of them also has a dependency with the Ability enum where they add their respective special skill. Lastly, for Scimitar to be sellable and purchasable, it implements the Sellable and Purchasable interface. This fulfils the interface segregation principle where each interface serves its own responsibilities and are only implemented by essential classes. This design can be easily extended if extra weapons were added, they just have to add their own special skill using the Ability enum and use the AreaAttackAction if it has an area attack. The pros of this design is that each class serves its own responsibility which fulfils the **single responsibility principle** but the cons are there will be many classes in the application.