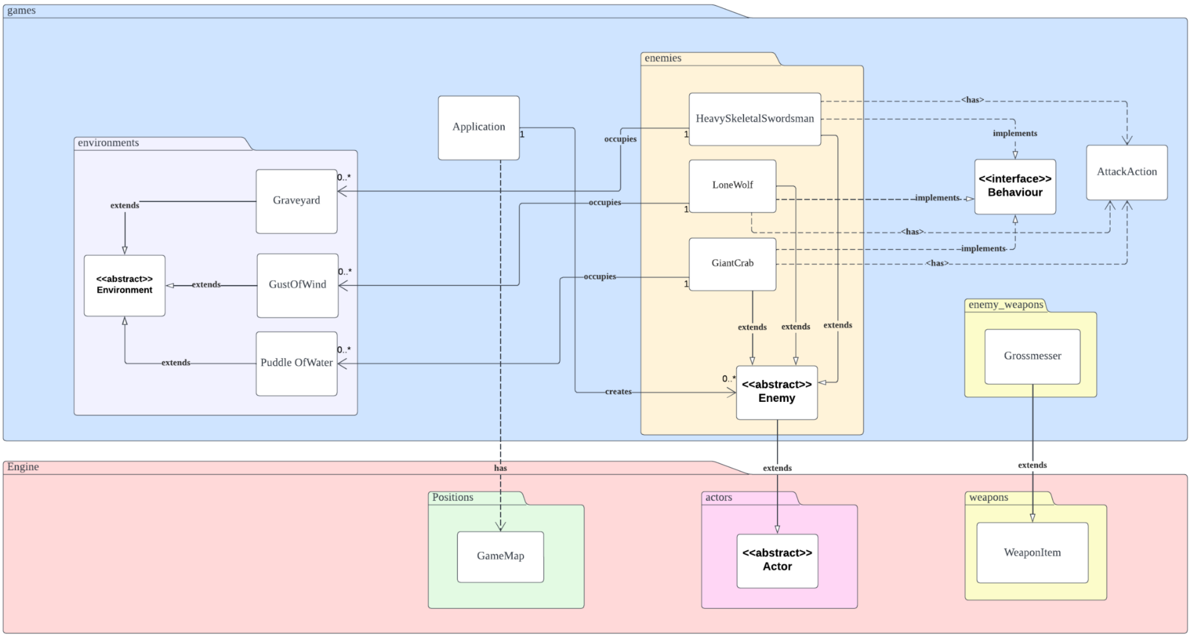
Lab06Team07’s Design Rationale

Team Members:
Chew Xin Ning, Foo Kai Yan, Ng Yu Mei

REQUIREMENT 1



Abstract class:

DRY (don’t repeat yourself) principle aimed to reduce repetition in codes, by replacing it with abstract classes to avoid redundancy. Changes to attributes and methods are easier as we can simply edit them in the abstract class instead of changing all individual classes with repeated codes.

Environment and Enemy abstract classes are created to be extended by individual concrete classes. Abstract classes are used for closely related objects. Concrete classes Graveyard, GustOfWind, and PuddleOfWater extend from Environment as they have common attributes such as spawn chance, and common functions to allow the corresponding creatures to be spawned from the environments at each turn. Next, we create another abstract Enemy class extended from Actor. Instead of having an Actor class as an abstraction of all actors, it is better to create an Enemy class to provide abstraction for the enemy, to separate “enemy actor” with other actors such as “trader actor” and “player actor” in the later part of the game. HeavySkeletalSwordsman, LoneWolf, and GiantCrab are extended from Enemy, which is also extended from Actor, to facilitate maintenance and implementation of common attributes/functions. Same goes for Grossmesser, which is a weapon and extends from WeaponItem.

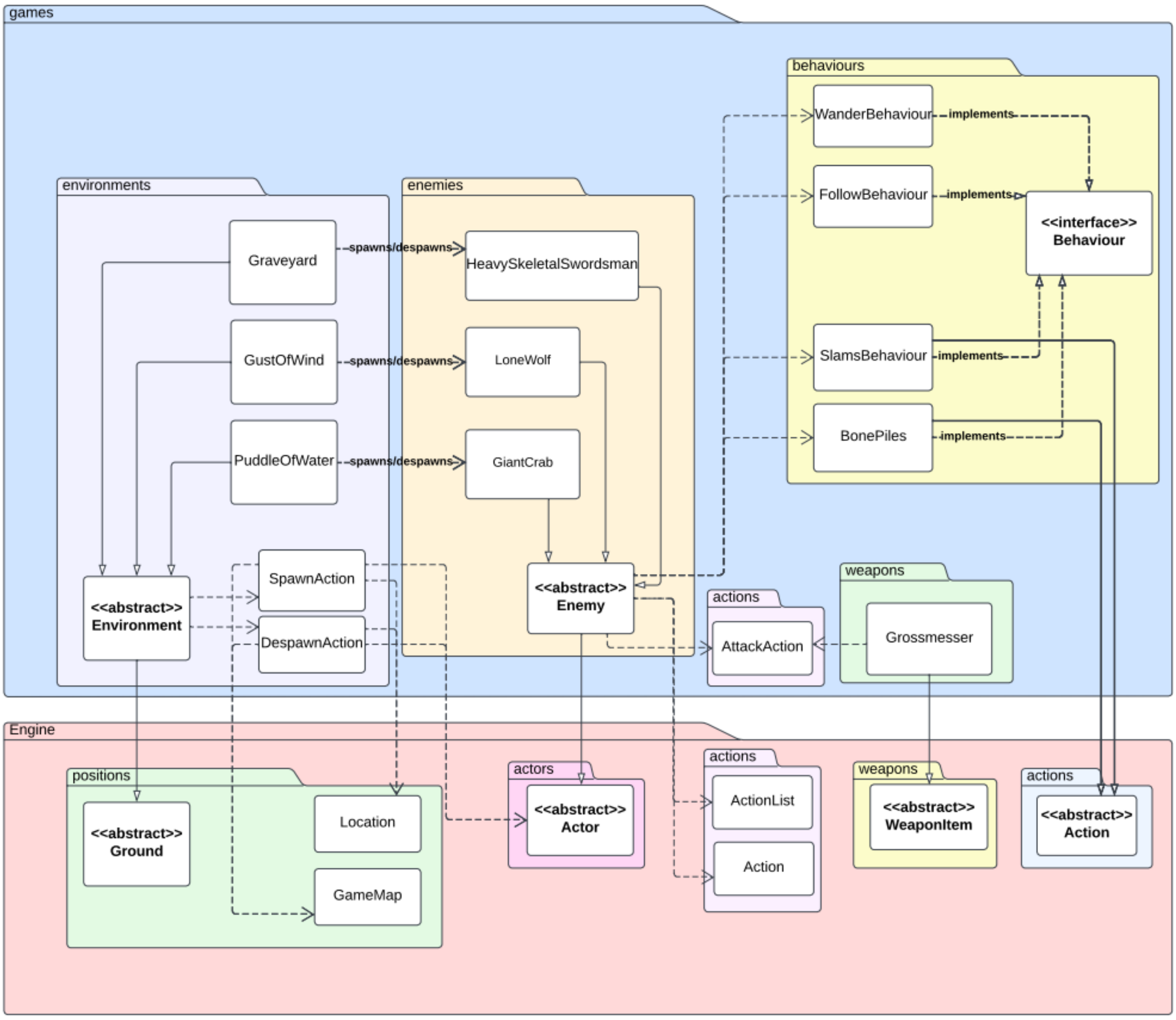
For easier navigation, Environment classes are regrouped under environments package; Grossmesser is added to a new package called enemy_weapons; whereas the three enemy classes are placed under enemies package. New classes with the similar type can be added easily into the package.

Interface:

Interface is to provide a common functionality for classes that are unrelated. It allows the separation of definition of a method from the inheritance hierarchy. As not only the enemy can have behaviour, other objects that are able to get action have a different implementation of the method, therefore methods in the Behaviour interface should not be bound by inheritance.

All three enemy classes implement the Behaviour interface class, which also consists of method getAction() to retrieve Action objects that can be performed by enemies. Each enemy also executes an attack action and thus creates dependency between each enemy and AttackAction class.

Improving from A1:



Enemy extends Actor. Actor consists of attributes and methods necessary for a generic Enemy. **HeavySkeletalSwordsman, LoneWolf, GiantCrab extend from Enemy.** Enemy provides a more concrete implementation for each enemy where it selects a valid action to be performed by the enemy using `playTurn` function, and `allowableAction` function returns a collection of what other actors can perform on the enemy according to their capability.

To do this, **Enemy has dependencies to every specialised behaviour**. Each enemy has their own capability represented using `Status enum` such as `HAS_SPECIAL_SKILL` for Giant Crab which has the special skill of slamming all creatures in its surrounding. The `allowableAction` method then checks their status and adds their specialised behaviour into `ActionList`. In this case, Giant Crab will have SlamsBehaviour action to perform on other actors when it is its turn.

The alternative way would be to use `instanceOf` or `getClass` to identify the enemy class. This approach involves downcasting and thus violates the use of abstraction and encapsulation. Also instead of using a very specific Status that only identifies the specific class, we use a more flexible Status such as `HAS_SPECIAL_SKILL` rather than

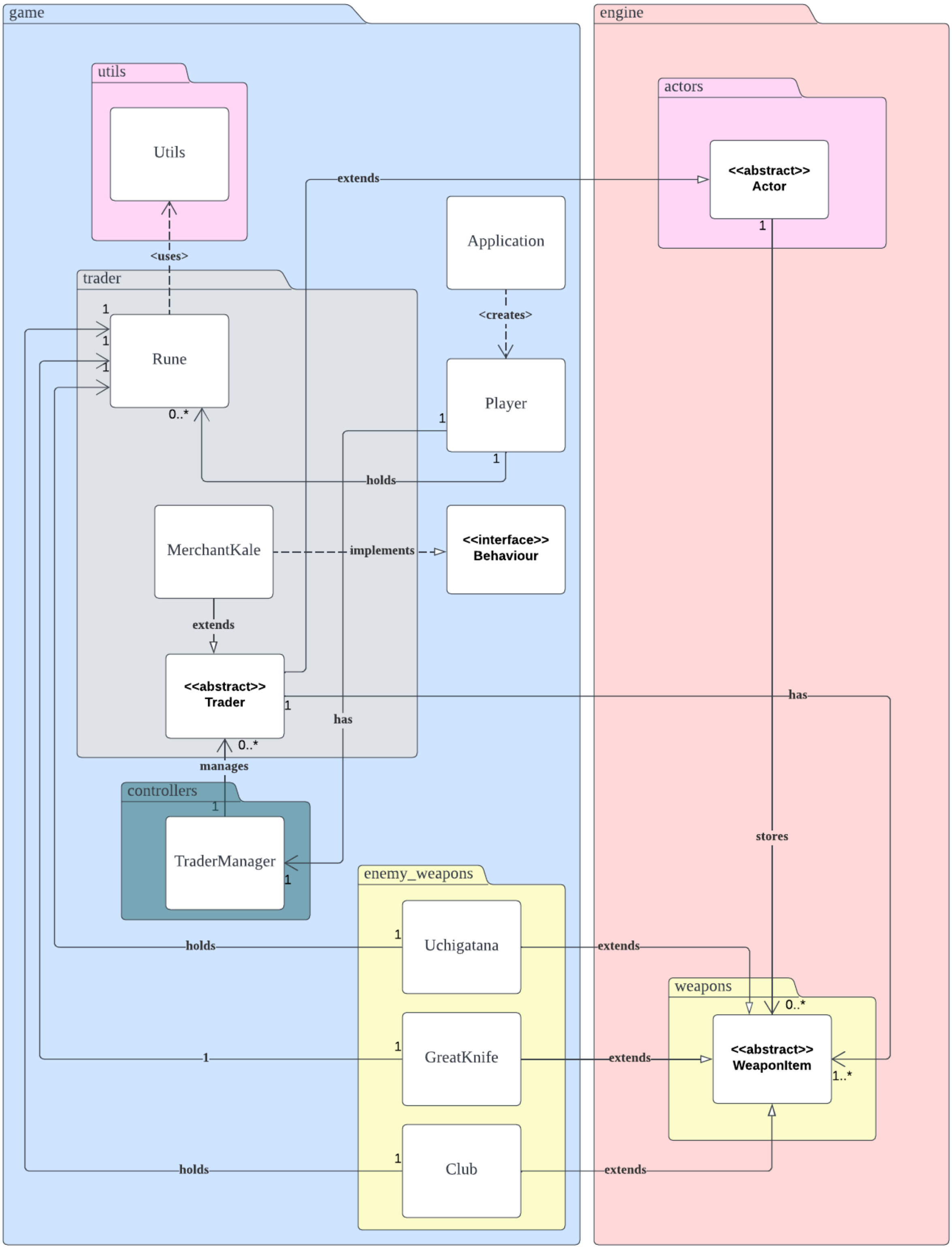
IS_GIANT_CRAB that only describes Giant Crab. This is not recommended as it creates extra dependency and is hard to extend in the future. If there is a new enemy, we have to add another if-else clause to check the identity of the class.

The drawbacks of making the allowableAction as such in the Enemy abstract class makes it a god class to determine what the others can perform on the current actor using multiple if-else statements. Although this avoids dependency from concrete enemy classes to their specialised behaviours, this approach violates the Open-Close principle (open for extension, close for modification). Whenever there is a new specialised behaviour for a new enemy, we are required to modify the allowableAction method by adding another if else clause for a new enemy, which would be tedious when many enemies are added.

Environment extends Ground. Graveyard, GustOfWind and PuddleOfWater extends from Environment. Environment has common attributes displayChar and CapabilitySet with Ground, tick function to allow the passage of time for the environment for each turn, and necessary function to return an ActionList of what the actor can do on the environment using allowableAction. Environment then provides a more detailed implementation of a concrete environment which includes spawnEnemy and despawnEnemy methods to spawn and despawn enemies. The ideal way to spawn and despawn an enemy in an environment is to attach the appropriate action to the corresponding object. Alternatively we can create this function inside each concrete environment class. However this approach would require the environment to know which enemy on the location to be spawned/despawned. With this, it creates more dependency between LoneWolf, HeavySkeletalSwordsman, GiantCrab with Actor, Location and GameMap to obtain the location and actor on that location in the in GameMap. This also violates the DRY (Don't Repeat Yourself) principle as we are repeating the procedure for spawning and despawning enemies. Therefore we create a despawnAction and spawnAction class to be used by Environment spawnEnemy and despawnEnemy functions, to adhere to DRY and also ReD (Reduce Dependency) principle. Now that the SpawnAction and DespawnAction only have dependencies with Actor, Location and GameMap classes.

Specialised behaviours (BitesBehaviour, SlamsBehaviour, BonePiles) are actions that can be performed by Enemy, therefore extending from Action abstract class. execute method performs the action to the target and provides a description of the action in menuDescription. These behaviours also implement the Behaviour interface and override getAction method to return the valid action that the other actor can perform on the enemy in Enemy playTurn method.

REQUIREMENT 2



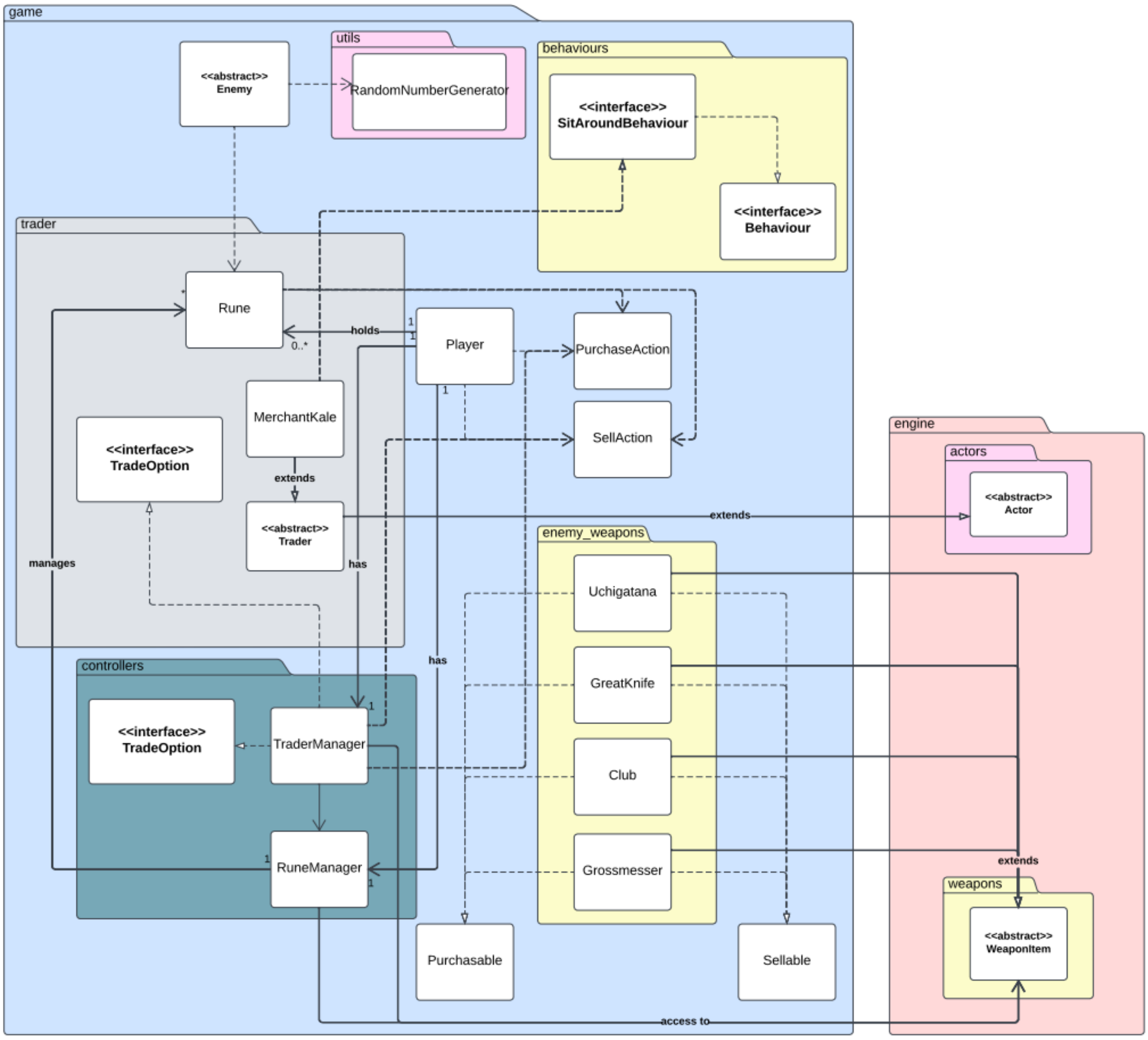
Utils class is created to generate the random number of runes within a specific range for the corresponding weapons. Utility methods such as validating inputs or generating random numbers in the future can be placed under this class to avoid repetition of codes in multiple classes. The Rune class has a dependency relationship with the Utils class since the number generated depends on the Utils class. Alternatively, we can remove the Rune class and Uchigatana, GreatKnife and Club use Utils class directly. However, doing so will create additional dependencies between them which is not recommended. Hence, we created the Rune class to align with the ReD principle and break coupling between Utils, Uchigatana, GreatKnife and Club.

There could be more traders/merchants in the future, therefore Trader abstract class is created to handle this issue. This enhances reusability of code, new traders can be created by just inheriting from Trader rather than having duplicated code for the common attributes and methods, such as the display character, trading methods, list of weapons for sales.

WeaponItem class initialises damage, hit rate, verb for weapons held by enemy, to be extended by Uchigatana class, GreatKnife class and Club class. Player holds a Rune object to buy weapons, it then makes a deal with the Trader class by specifying the weapon that it is intended to buy. Trader will accept the Weapon object as type WeaponItem as a parameter in the makeNewTrade method, and pass it to the TraderManager to perform the actual trading. Manager then has access to the corresponding weapon runes through public getters (Encapsulation) to perform trading. In this way, these weapons are managed by the manager through the WeaponItem interface rather than directly connecting to the concrete weapon class to reduce dependency. This implementation also adheres to the Single Responsibility Principle (SRP), Trader class focus on defining the attributes and function to make new trades, whereas TraderManager manages trades passed by Trader. Therefore, Player has one TraderManager, and TraderManager manages 0 to many Traders.

MerchantKale is the concrete trader extended from the Trader abstract class. We define Trader as abstract class instead of interface because interface will force all traders to implement all its methods. Most of the time, traders have high chances of having the similar or exactly the same implementation for making a new trade, but with just a slight difference in display character or actions to be performed. Therefore, an interface is not suitable for our situation, which is why we want to create a class that contains methods that most traders will do, but also with flexibility to override methods.

Improving from A1:



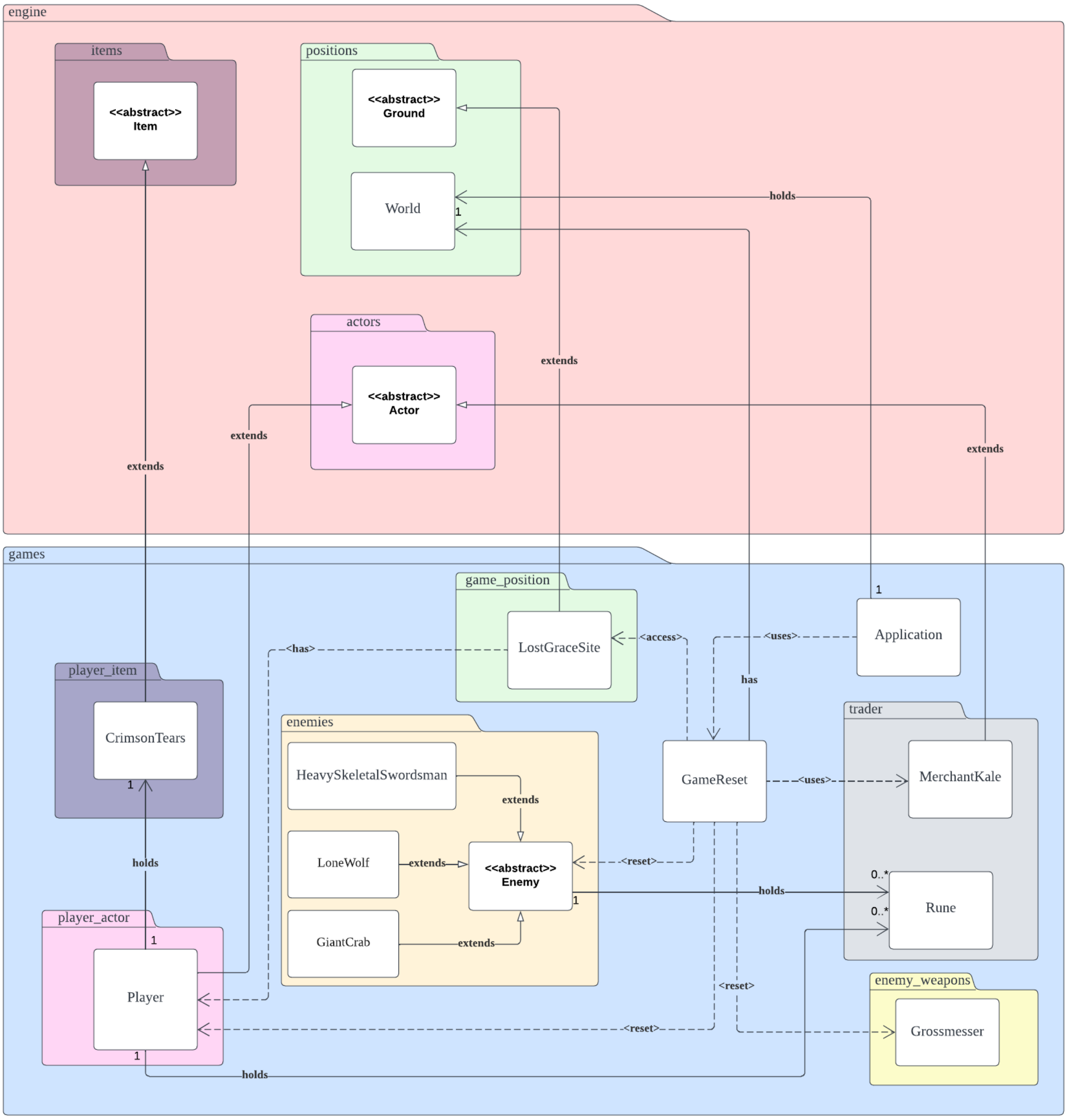
Trader extends from Actor. Actor consists of certain attributes and methods necessary for a generic Trader as a Trader is also considered as an Actor. **MerchantKale extends from Trader. MerchantKale implements SitAroundBehaviour.** Since it was assumed that all Traders have different behaviours and for now, there is only 1 Trader present so only MerchantKale implements SitAroundBehaviour. If more traders were to be added and all these traders have the same behaviour then the abstract class Trader will directly implement SitAroundBehaviour to adhere to DRY principle to avoid all individual trader classes implementing the SitAroundBehaviour interface one by one repeatedly.

Interface Segregation Principle (ISP) states that a class should not implement an interface if it doesn't use it. Purchasable and Sellable interface provides the selling/purchase price, as well as getter to obtain the price. In this case, **weapons will only implement the interface if they can be traded with traders.** Weapons that are not sellable/purchasable should not implement these interfaces.

Enemy abstract class contains `generateRune` method to generate random number of rune drops by enemies when they are defeated. It uses the `getRandomInt` method in the

RandomNumberGenerator class. Since every enemy drops rune when they die, we place this method in the parent class for all other enemies to use. This approach reduces dependency between enemies and RandomNumberGenerator class and always avoids repeated code. However, passing in magic number (the upper and lower bound for random generated rune) introduce connascence of meaning, but since the upper and lower bound for the random generation of drop is different for all enemy and it is only used once in the class, and not used in any other class, it is still acceptable.

REQUIREMENT 3



DRY principle is applied by extending CrimsonTears class from Item abstract class. Crimson Tears is an item as it has common attributes and behaviours such as item name, item display char and item allowable action list. We create CrimsonTears class as a new class because it has its own unique actions that other items do not and it only be held by the player. Same goes for LostGraceSite class that inherits from Ground class, having common implementations such as ground capability set, ground display char, ability to check which actor can enter and others.

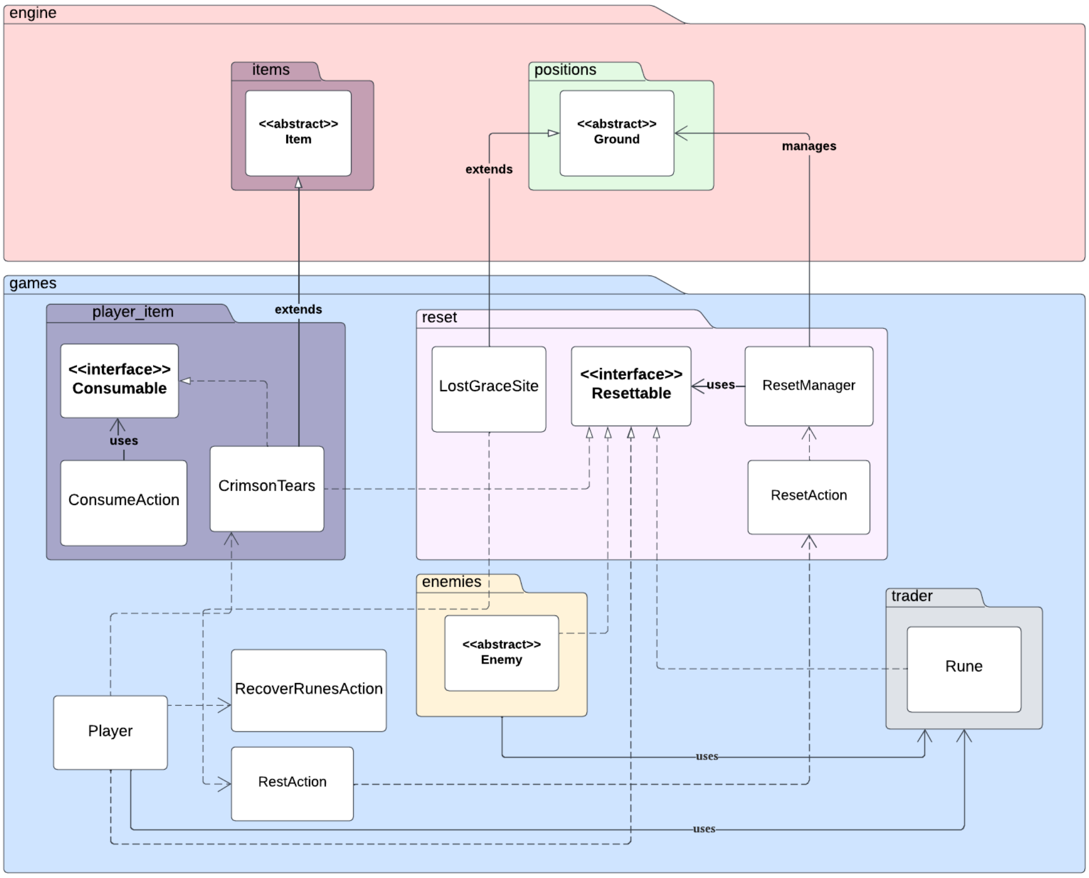
The player has the association relationship with the CrimsonTears class because the player will hold the Crimson Tears at the start of the game or when game reset. The Player class owns a CrimsonTears object and will be initialised under its constructor, hence creating an association relationship between the Player and CrimsonTears. New method will be created in the LostGraceSite class that holds the Player class type parameter for returning the fixed action that only the player can do in the LostGraceSite class.

Both abstract Enemy class and Player class have an association relationship with the Rune class as the Rune type variable should always be initialised when the Enemy or Player type object is instantiated.

The principle of ReD is applied by building the association relationship between GameReset class and Enemy abstract class instead of building the relationship between GameReset class and the individual enemy classes.

The GameReset class will reset the attributes such as hit point (HP) of the player and enemies. To reset the game, the GameReset class will use a newly added function in World class to create a new world, and also uses Grossmesser class to know the location of the dropped weapon. The GameReset class also has to know about the location of the last site of lost grace that they visited using methods in LostGraceSite. Hence, GameReset creates dependencies with Player, Enemy, World, LostGraceSite and Grossmesser classes. Application class, which the main class creates dependency with the GameReset class to instantiate the start of the game as resetting the game is almost equivalent to having a new game brought up.

Improving from A1:



A Consumable interface is created for all items that can be consumed. A consumable must implement the method under the Consumable interface. The Consumable interface can reduce the number of dependency relationships between consumable and the ConsumeAction since all consumables can be consumed and invoke ConsumeAction and apply the ReD principle. The Consumable interface will contain the method that the ConsumeAction will use to apply ReD principles.

A Resettable interface is created for all items and actors that can be reset. The ResetManager is created to manage all the items and actors that implement the Resettable interface. `registerResettable()` will be invoked by all classes that implement the Resettable interface. We make the ResetManager as a single instance by calling the `resetManager()` which will always return one same instance of ResetManager throughout the game. This is because the ResetManager class is just used to manage the resettable and nothing will be changed within the ResetManager throughout the game. This makes it convenient to the ResetAction class that used to reset every resettable since the `run()` under ResetManager will make all resettable invoke `reset()` method. The ResetAction class can invoke the ResetManager instance that has already been created and store all items and actors that can be reset.

The ConsumeAction constructor contains a Consumable type item, and it will be invoked under the class that implements the Consumable interface. The player can only consume one item every time. The ConsumeAction will only be invoked when the item is consumed.

ConsumeAction will extend from the Action class because it is an action and it has all characteristics that the Action has. This has applied DRY principles. This can also make the ConsumeAction class to compulsorily implement the `execute(Actor actor, GameMap map)` to perform action and `menuDescription(Actor actor)` to display option description on menu.

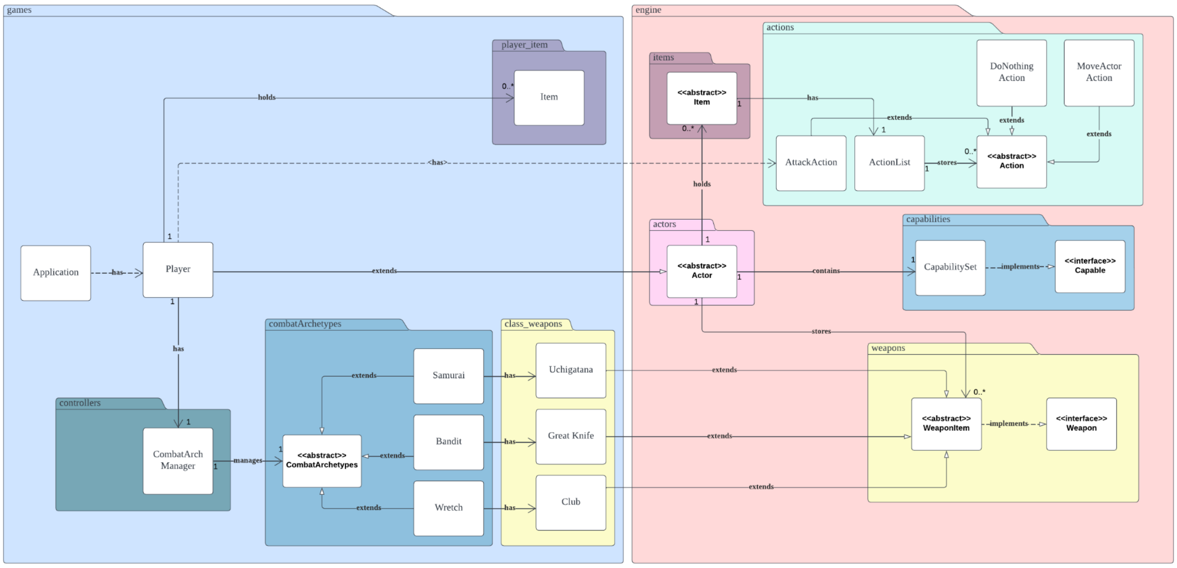
CrimsonTear class extends from item abstract class and implements the Consumable interface and Resettable interface since it can be consumed and reset and it has common attributes and behaviours such as item name, item display character and item allowable action list. This has applied both DRY and ReD principles. We make CrimsonTear class as a single instance by invoking the getInstance() because it can only be held by the player and one of its attributes, maximumUse will change its value throughout the game within only one single CrimsonTear class.

Three class variables, playerRune, runeValue, and runesDroppedLocation will be created under the Player class. We make them as class variables because these variables are only created under the Player class and their value will be changed from another class which contains only the Actor class type variable. Therefore, these class variables can be used by the player in another class through getter and setter of class variables.

LostGraceSite class will be made as a single instance variable since it will not be changed and the game will only create one throughout it and allow other classes to get the value of the variable of one LostGraceSite. Since the player will be respawned at the LostGraceSite, the location of LostGraceSite class will be recorded.

The RestAction and RecoverRunesAction extend from the Action abstract class to apply DRY principle since it has common attributes and behaviours such as e`xecute(Actor actor, GameMap map)` and `menuDescription(Actor actor)` that Action class has. However, different implementations of these two functions will be given for different classes.

REQUIREMENT 4



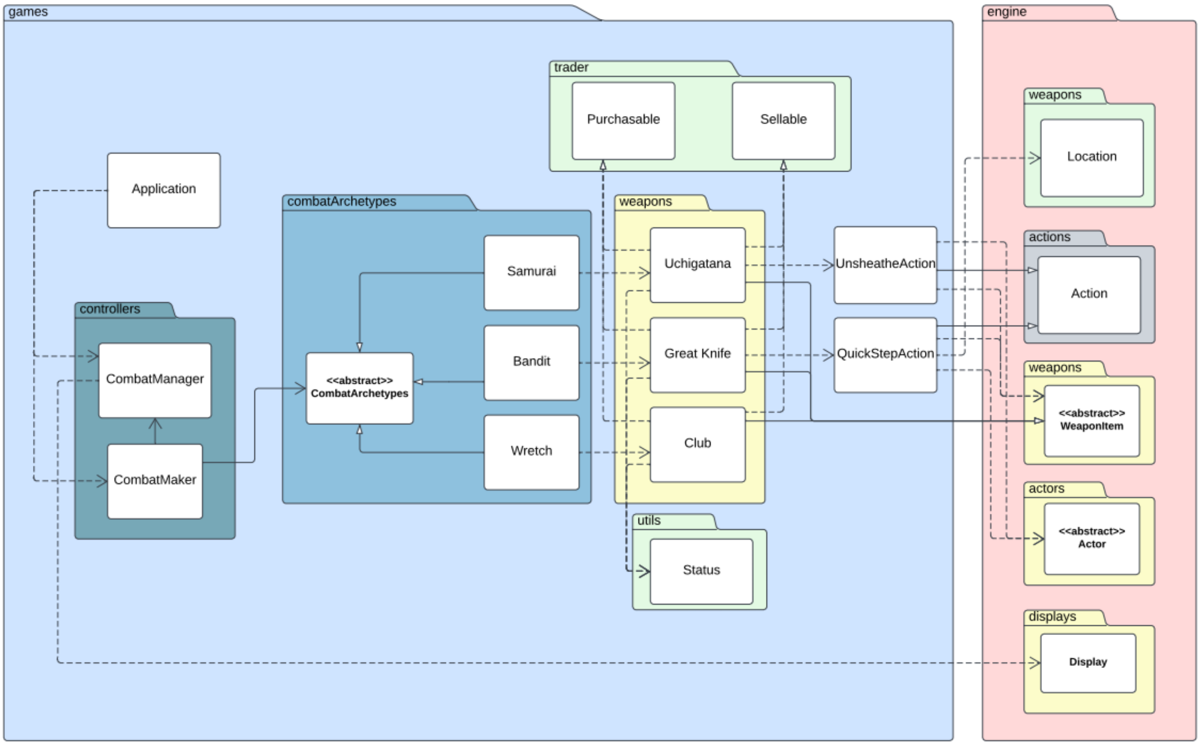
Each Player has a CombatArchManager which helps to manage CombatArchetypes for the player. For example, if the player choses Samurai as its starting class, the manager class will then accept this argument in its methods and create the corresponding CombatArchetypes, which is Samurai object in this case. With this, Player can entrust the creation of the starting class to the manager. CombatArchetypes will initialise common attributes like HP and the fact that each class has their own unique weapon (DRY Principle).

Same as Requirement 1 and 2, the starting weapons held by each starting class also inherit from WeaponItem abstract class that implements a weapon interface. These unique starting weapons also inherit from abstract classes found within the game engine package to reduce repeated codes as these weapons have common attributes like their own fixed damage and hit rate or also known as accuracy. Some weapons also allow players to perform unique skills. Because of that, now that each weapon is its own class, it will only be responsible for the specific weapon's attributes and use. And also, in this case, the players of special Combat Archetypes could only have access to their unique weapons which are special to their chosen Combat Archetypes. If the player has chosen to be a Samurai then they can only have access to Uchigatana. The player would not have access to unnecessary weapons such as Club as their starting weapon. This will also allow us to apply the ReD.

For CombatArchetypes abstract class there are three different subclasses which are Samurai, Bandit and Wretch. The aim is to have the CombatArchetypes abstract class to serve as a blueprint for creating Samurai, Bandit and Wretch which are different Combat Archetypes present in the game. This allows us to define common properties and methods that are shared among all different Combat Archetypes, such as HP. Each subclass which are Samurai, Bandit and Wretch would implement the specific characteristics and behaviours of its respective Combat Archetypes, such as unique skills and weapons. As mentioned before, the sole advantage of doing this is to reduce the dependencies between Player class and each Combat Archetypes class. With the same reasoning as CombatArchetypes abstract class and how it is linked to each different Combat Archetypes classes, each unique weapon of each different Combat Archetypes classes will inherit from the WeaponItem abstract class found in engine package class as this too allows less dependencies to occur for each weapons.

By doing so each player would have a dependency relationship with CombatArchManager which is a class that helps manage a player's chosen class and also an additional dependency with their starting weapon of their chosen class. Players extend Actors which allows each player to have certain Capabilities from the CapabilitySet that implements the Capable interface from the game engine package. The capabilities are unique to the class chosen by the player at the start of the game. Players hold items in-game and each item inherits the abstract item class from the game engine package which allows players to take the item out from their inventory with an action from their ActionList. Players have an ActionList which extends from the Action abstract class which shows that players can use these actions. And since Players do attack enemies of different environments, players would have an AttackAction within their ActionList which extends from the Action abstract class.

Improving from A1:



Bandit, Samurai and Wretch extends CombatArchetypes. CombatArchetypes is an abstract class that includes name, the starting weapon, and starting hp for the role to adhere to DRY principle.

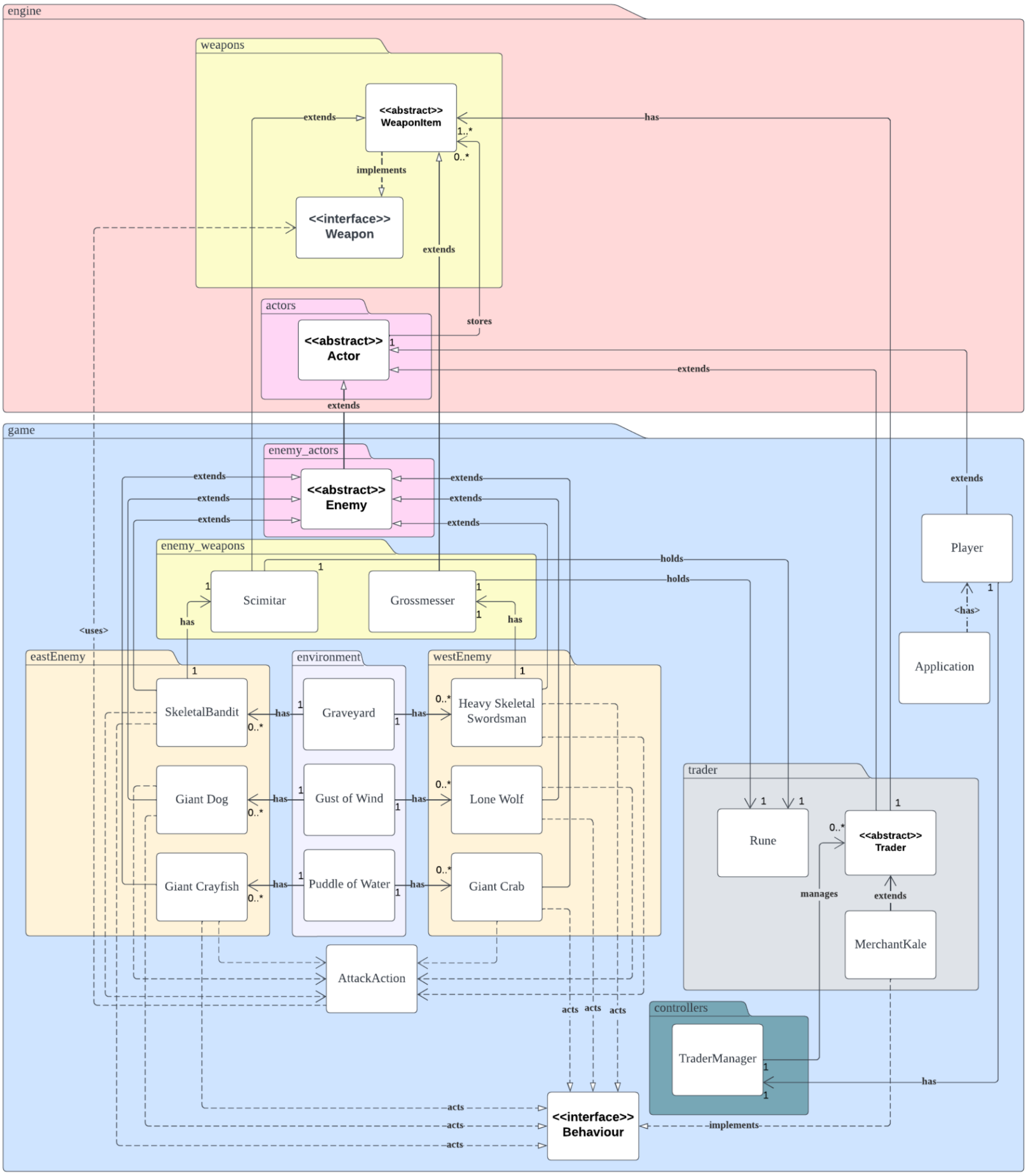
CombatManager calls CombatMaker to initialise the starting role for the player. To avoid the manager class to be a god class for both managing the player choice and also the creation of the archetype, we separate this responsibility into two classes which are CombatManager and CombatMaker. CombatManager is used to display the console menu and accept player choice for their starting class role and weapon by calling the `menuItem` function. Then the maker class creates the respective archetype based on user input by calling `createCombat` function.

This complies with the SRP (Single Responsibility Principle) to ensure a class only has a single, clear purpose. If there is more than one functionality in one class, it introduces coupling between them and hence when one function is changed in the future, we might break the coupled functionality and be required to test and debug for the other functionality as well. Therefore by separating the functionalities into two classes, it reduces coupling between individual archetype classes with the manager class.

However, this approach violates OCP (Open-Close Principle) which opens for extension but is closed for modification. For example when there is a new combat archetype to be added, we need to add another option for the user to choose and also modify the `createCombat` function with another switch case to create the new archetype.

The CombatManager and CombatMaker class has only one instance at all times. We maintain them as a single class instance throughout the game. This is done by the `getInstance` method in the classes where the same instance of the class will be returned instead of creating a new instance. This is called the singleton pattern, it ensures reusability by avoiding the same singleton object being used again and again. CombatManager and CombatMaker will always have the same behaviour and therefore there is no need to create another instance of them when they always have the same functionality.

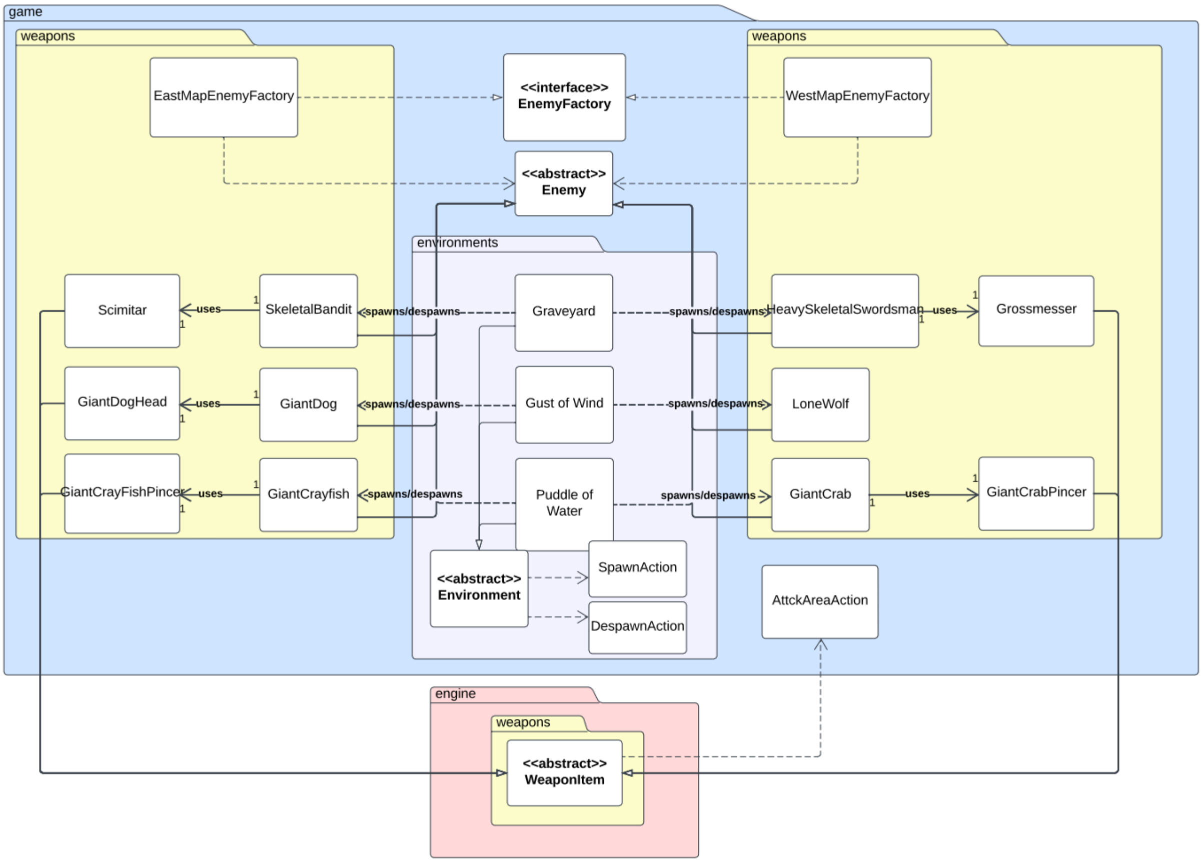
REQ5



More enemies are added. These more enemies found within the GameMap are now split to 2 packages which are enemies that spawn in the East and West side of the environment within the GameMap. In adherence to DRY and ReD principles, these enemies inherit their common attributes like their HP, spawn chance, unique abilities and runes dropped after defeat. One of the new enemies, Skeletal Bandit, also has a unique weapon, Scimitar, which too inherits from WeaponItem abstract class that implements a Weapon interface. As mentioned beforehand, the WeaponItem abstract class serves as a blueprint to create different weapon classes which includes both enemies' and players' weapons as well as different unique weapons of each Combat Archetypes class. With the WeaponItem abstract class, there will be less dependencies present which in return allows less things to be managed by a single class. It is better to have different weapons to have its own class and just inherit common attributes from an abstract class than having a general weapon class where all weapons are managed by it. This adheres to the SRP to avoid having a GOD class managing every weapons’ methods. Hence, now that each weapon is its own class, it will only be responsible for the specific weapon and let's say the weapon is Scimitar then the class will only have attributes related to the Scimitar which is uniquely only used by the Skeletal Bandit.

The behaviour of these additional enemies like the act of wandering and following players at certain distances and occasions are from the implementation of the Behaviour interface. When players are close in distance to the enemies, the enemies will follow and attack the Players. Hence, these enemies would have an AttackAction within their ActionList which extends from the Action abstract class as the enemies will be the object that players will conduct an action against the object and the action would be AttackAction.

Improving From A1:



EastMapEnemyfactory and WestMapEnemyFactory implement EnemyFactory interface. This interface provides a `newEnemy` method to spawn enemies on East and West. The ground is separated into east and west indicated by using `MapLocation.EAST` or `MapLocation.WEST` in `MapLocation` enum class as an attribute in the ground. The two enemy factory classes serve as a level of abstraction to spawn enemies. During spawning, we use polymorphism to accept the East and West enemy factory classes as attributes and there is no need to know the exact information carried by the attribute (thus providing abstraction) and spawn enemies on the respective east/west side of the ground.