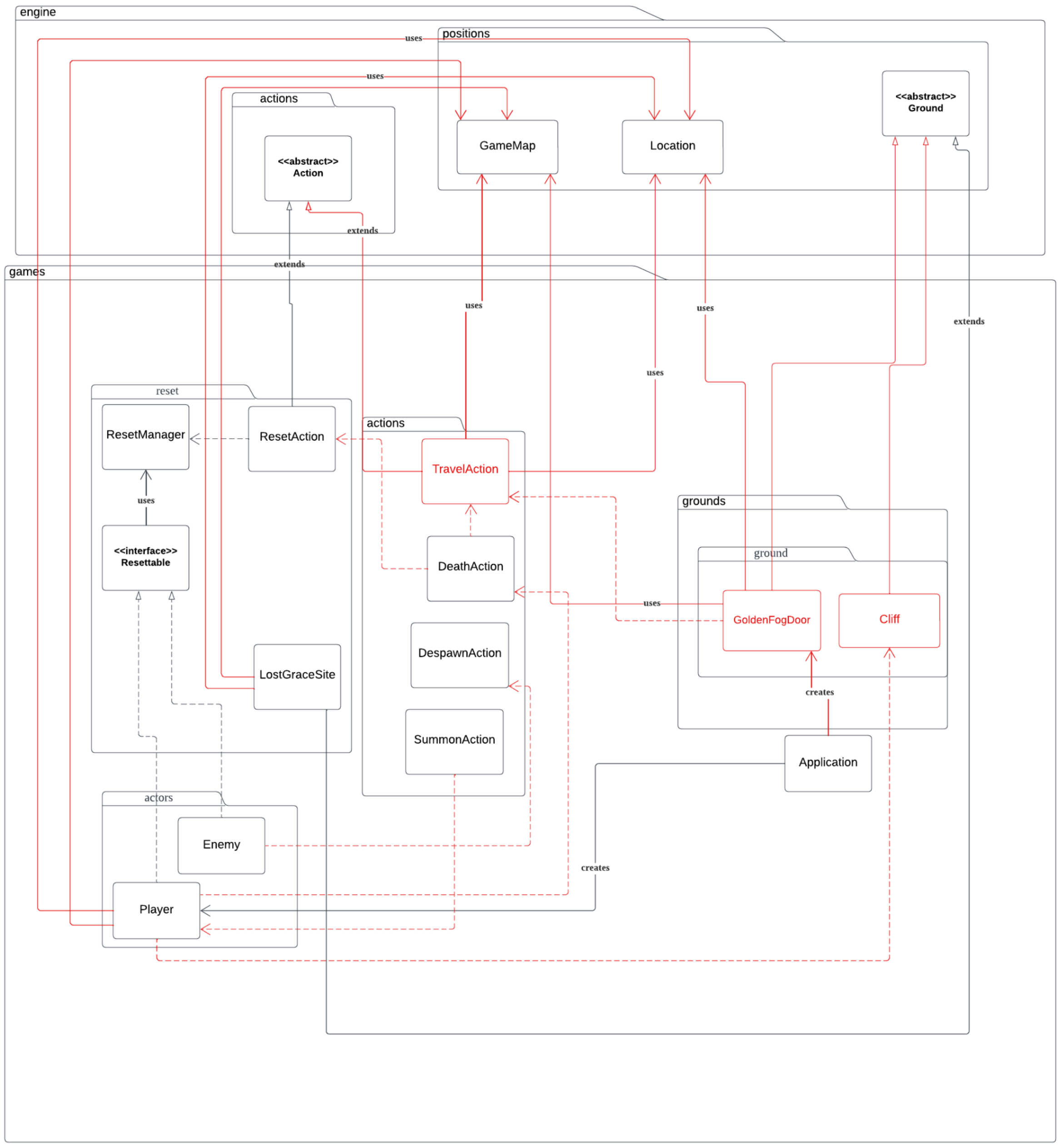
Lab06Team07’s Design Rationale

Team Members:

Chew Xin Ning, Foo Kai Yan, Ng Yu Mei

Table of Contents

REQUIREMENT 1



GoldenFogDoor class extends from **Ground abstract class** because it has a common attribute of display char, `canActorEnter` method to check if the actor that wants to step on GoldenFogDoor is Player and `allowableActions()` to return new ActionList that contains actions that will be performed when the Player enter the GoldenFogDoor(DRY principle). Its `allowableActions()` will return an ActionList that contains TravelAction if the Player enters the GoldenFogDoor so the dependency relationship is built between GoldenFogDoor class and TravelAction class. The **GoldenFogDoor need to pass the GameMap type instance and Location type instance to the constructor of TravelAction class** when TravelAction being built so that the association relationships are built between GoldenFogDoor class and GameMap class and GoldenFogDoor class and Location class because the GameMap type instance and Location type instance will be got from input.

TravelAction extends from **Action** because it has common methods of `execute()` and `menuDescription()` being overridden and used to move actors from current map to another map or one location to another location (DRY principle). Its `execute()` method just uses the map.moveActor() to move Player from the current map to the particular location of another map. Therefore, the association relationships are built between TravelAction class and GameMap class and TravelAction class and Location class because we will use destination's map and destination's location from input for our implementation of `execute()`. Although the input type of constructor of TravelAction class and the input type of constructor of GoldenFogDoor class are the same, we still not directly pass GoldenFogDoor type instance to the constructor of TravelAction class because we want to make TravelAction class can be flexible used in the future. The creation of the TraveAction class also applied the Single Responsibility Principle (SRP) since the TravelAction class only focused on moving the Player from the current map to the destination map without other functionality. Therefore, if the movement of the Player between maps does not work well in the future, we can easily debug to fix the problem.

Cliff extends from **Ground abstract class** because it has a common attribute of display char and `canActorEnter` method to check if the actor can enter the Cliff (DRY principle). Since we want to make Cliff instantly kill the player without asking in the menu description, we will check if the player steps on the Cliff in the `playTurn()` of Player class by checking if the current location of the player is equal to the location of the Cliff. Therefore, the dependency relationship is built between Player class and Cliff class. **The DeathAction class will be returned directly if the Player steps on the Cliff** so that the dependency relationship between Player class and DeathAction class is built either. **The implementation of our DeathAction class is improved by moving the Player to the location of LostGraceSite and returning the ResetAction class at the same time** so that the reset() will be invoked at the next round automatically and the enemy can be removed from the map directly. Therefore, the dependency relationship is built between the DeathAction class and ResetAction class. **DeathAction will call TravelAction.execute() to move Player to the location of Lost of Grace Site** so that the dependency relationship between DeathAction and Travelction. **We use the TravelAction class instead of map.moveActor()** because we may need to move Player dead on another map to the location of Lost of Grace Site on another map.

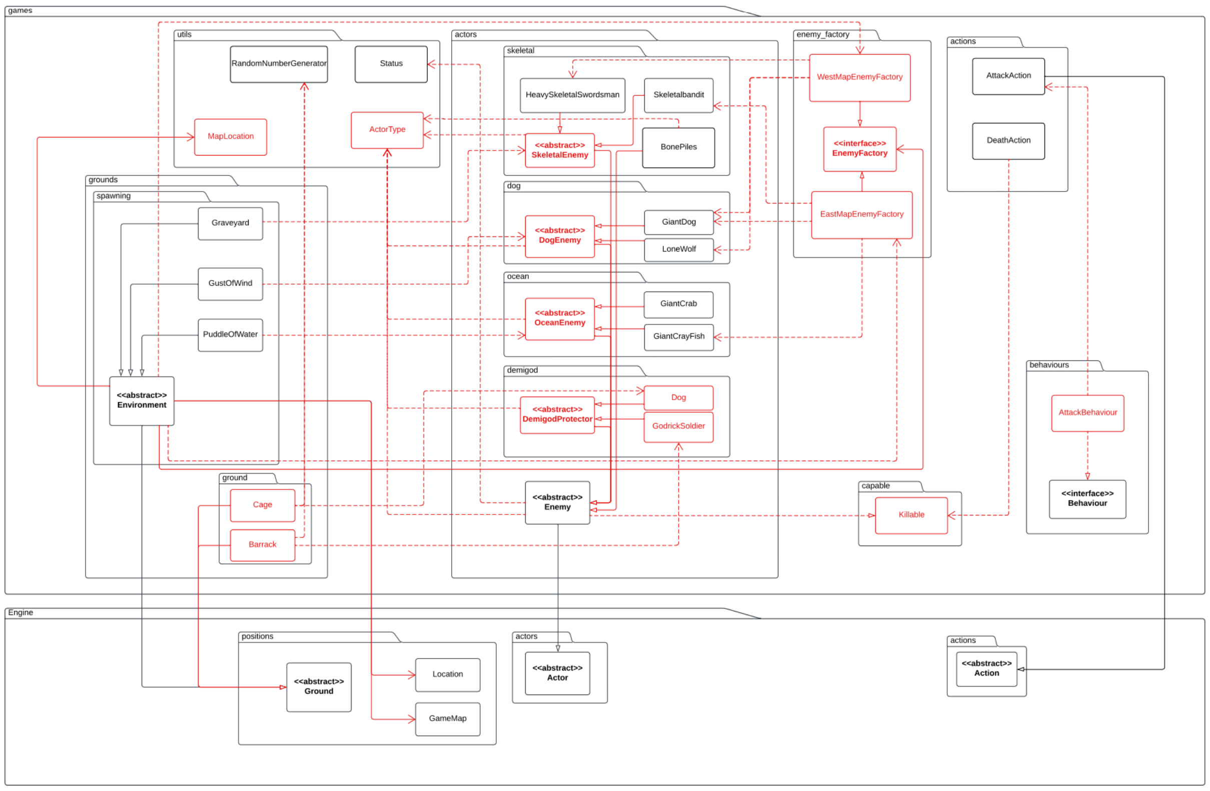
Besides, the association relationship is built between Enemy abstract class and GameMap class and the GameMap type instance will be obtained from the input. Therefore, the GameMap type instance can be used to remove enemies from the map at the next round after the player died in the reset(). **The method we use to remove enemies from map is DespwanAction.execute()** so that the dependency relationship between the Enemy abstract class and DespawnAction class is built.

The new association relationships are built between LostGraceSite class and GameMap class and LostGraceSite class and Location Class. Therefore, the location and the game map of the Lost of Grace can be kept track of different Lost of Grace Sites. We can move the Player to the particular Lost of Grace site easily because we know their location and map the Lost of Grace Site on. The dependency relationship is also built between LostGraceSite class and RestAction class because we expect the Player being asked if Player wants to rest when The Player steps on the Lost of Grace Site.

Ally and Invader will be removed from the map if and only if the Player died. Therefore, **we use HashMap<GameMap, Actor> to store maps that spawn Ally and Invader as key sets and the actors who are Ally or Invader as the corresponding value**. Therefore, we can remove Ally and Invader from the corresponding map in the reset() of the Player when the Player dies. **We use addGuest() to store actors and map into HashMap in the SummonAction class** since this class will implement the spawning of the Ally of Invader. Therefore, the dependency relationship is built between Player class and SummonAction class. Besides, Ally and Invader should not implement the Resettable interface because they will not use reset(), the method in the interface, even though Invader are enemies and they will be removed from the map when the Player dies to apply Interface Segregation Principle (ISP). They are different from other resettables since they cannot be reset when the Player rests only.

The association relationship is also built between Application class and GoldenFogDoor because **we want to create new maps with the GoldenFogDoor being set to the map in the Application class**. We want all maps being processed in the background even though the Player does not stay on that map at the moment so that all maps will be added into the world before the world.run() is called.

REQUIREMENT 2 (Red = changes made)



Updated Feature:

Added AttackBehaviour. AttackBehaviour returns AttackAction in `getAction` during the `playTurn` of enemies that can perform attack on other actors.

Enemies have been refactored to extend from their enemy type class. **SkeletalEnemy, DogEnemy, OceanEnemy extend from Enemy abstract class.** Each concrete enemy then extends from their enemy type class. **Heavy Skeletal Swordsman and Skeletal Bandit extend from SkeletalEnemy. GiantDog and LoneWolf extend from DogEnemy. GiantCrab and GiantCrayFish extend from OceanEnemy.** Since the `playTurn` for enemies of the same type are similar, we put them into the enemy type class to reduce redundancy of code (DRY Principle).

Enemy creates dependency with ActorType. To select the actions that can be performed by other actors to the current enemy, `allowableAction` in Enemy abstract class uses ActorType enum to check for the type of actor then determine whether they can be attacked. Each type of enemy has a unique ActorType, and is assigned to every enemy in the Enemy Type class by using `addCapability` method. For example, LoneWolf is of type dog and inherits from DogEnemy which has the capability ActorType.DOG_TYPE. This creates dependency between ActorType with SkeletalEnemy, OceanEnemy and DogEnemy classes.

Enemy abstract class implements Killable interface, which contains `getRune` method to obtain the rune generated by enemies after they are defeated by Player. Since all enemies drop runes to Player after death, we make Enemy implement the interface instead of letting each individual enemy class implement the interface. This reduces the dependency between Killable and individual enemy, and complies with the Reduce Dependency (ReD) principle. When enemies are defeated, DeathAction will be called and uses the Killable interface to obtain the runes generated by the enemy. Dependency Inversion Principle (DIP) applies here where DeathAction no longer depends on the low-level modules (individual enemies), instead, it depends on the abstraction which is the Killable interface, to obtain the runes.

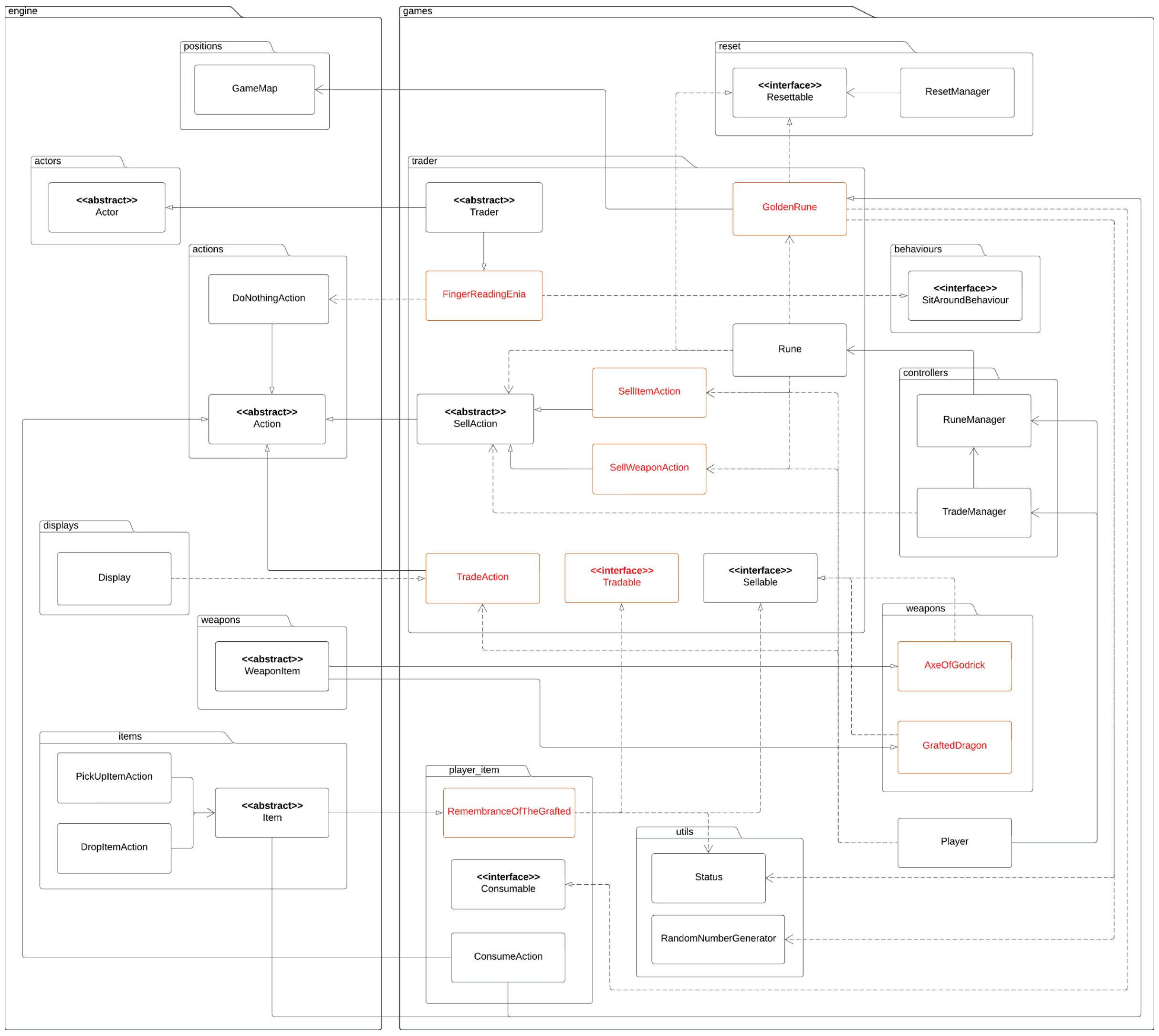
Environment abstract class uses EnemyFactory interface to spawn enemies on each turn, by using EastMapEnemyFactory and WestMapEnemyFactory. To achieve the optimal implementation that is extensible, the environment has an attribute called mapLocation which is a type of MapLocation enumeration. This attribute indicates which side the environment is on the game map - East or West. As we do not know the map location at first, we use polymorphism to wrap the two enemy factories together, and create an EnemyFactory attribute in Environment to serve as a level of abstraction. In each turn, the tick function will be called and it checks the map location of the current environment using `getMapLocation` method, this function will then initialise the enemy factory to be east or west map enemy factory. With this, we wrap both factories and use the EnemyFactory interface to spawn enemies on top of the environment ground. This approach does not require us to know the exact location of the environment as it performs the checking itself and utilises the appropriate enemy factory for spawning. Each environment has a method called `spawnEnemy` which uses the enemy factory to spawn their respective enemies depending on their location on map. East and West Enemy Factory implements the spawning enemy methods to spawn different (Ocean, Dog and Skeletal type) enemies. For example, Gust of Wind spawns DogEnemy on the map. When the `tick` function is called in each turn, it checks the map location of Gust of Wind and initialises the suitable enemy factory, spawnEnemy function then calls enemyFactory.spawnDogEnemy(). If GustOfWind is currently on the east map, it uses the east factory to spawn GiantDog, else if it is on the west map, it spawns LoneWolf. If a new type of map side is added such as North map, NorthMapEnemyFactory can be added and implements EnemyFactory to spawn the respective enemy.

New Feature (optional weapon Heavy Crossbow is not implemented):

Dog and Godrick Soldier extend from DemigodProtector, which extends from Enemy. `playTurn` method is defined in DemigodProtector class to avoid repetition of code for similar behaviour. All enemies' allowableAction was defined in Enemy class as well as they have the same implementation of the `allowableAction` (DRY Principle).

Cage and Barrack extend from Ground abstract class as they have the common attribute of displayChar and also have the `tick` method to spawn enemies on each round. Cage and Barrack spawn Dog and Godrick Soldier respectively in each turn. They spawn them in the `tick` method by first checking the spawn and despawn chance using RandomNumberGenerator class and therefore create dependencies. Cage and Barrack are not inheriting from Environment abstract class because they do not spawn enemies depending on the map location and do not use enemy factory, therefore have a different implementation of `tick` function. Hence, we make them to inherit from Ground which we can override the `tick` method to spawn enemies.

REQUIREMENT 3



GoldenRune implements the Consumable and Resettable interface as GoldenRune dropped randomly across the map can be reset once the game reset and Player can consume GoldenRune to obtain a random number of runes from a specific range. `generateRune` method from Rune class will be used to generate a random number of runes from a given range. `getRandomInt` method from RandomNumberGenerator class and `getXRange` and `getYRange` method from GameMap class will be used to generate random x and y coordinates to spawn and drop the GoldenRunes generated randomly across the game map.

When GoldenRune is picked up, PickUpItemAction will be used and `addItemToInventory` method from abstract actor class will also be used to add the picked up GoldenRune to player's inventory whereas if GoldenRune is consumed with ConsumeAction, `removeItemFromInventory` from abstract actor class method will be called and used to remove GoldenRune from player's item inventory. On the other hand, if players like to drop their GoldenRune, DropItemAction will be used to place the GoldenRune onto the current position of the player on the game map.

FingerReadingEnia class extends from Trader abstract class and implements the SitAroundBehaviour interface. FingerReadingEnia implements SitAroundBehaviour as Finger Reading Enia just sits around in the Roundtable Hold which is somewhere within the game map. For now, both Merchant Kale and Finger Reading Enia implements SitAroundBehaviour interface with their own class as both these traders have the same behaviour throughout the game play but if more traders were to be added and there is a chance that all these traders would have the same behaviour as Merchant Kale and Finger Reading Enia then the abstract class Trader will directly implement SitAroundBehaviour to adhere to DRY principle to all individual trader classes to avoid implementing the SitAroundBehaviour interface one by one repeatedly.

In adherence to the Interface Segregation Principle (ISP) where all classes should not implement an interface if the said class doesn't use it, **items and weapons like RemembranceOfTheGrafted, AxeOfGodrick and GraftedDragon will only implement the Sellable interface** and not the Purchasable interface as these item and weapons can be sold to Finger Reading Enia for runes but can't be bought from any traders as of the current design of the game.

Remembrance of the Grafted is dropped by Godrick the Grafted after defeat and it is an item that can only be traded with Finger Reading Enia in the current version of the game for runes or weapons. Hence, **RemembranceOfTheGrafted implements Tradable interface** and is of tradable type. But due to insufficient time and that Req3A was optional, Godrick the Grafted was not done and hence, players will start with 1 Remembrance of the Grafted in their item inventory.

TradeAction extends from the Action class as TradeAction is an action, so TradeAction has all characteristics that the Action has. This has applied and adhered to DRY principles. `execute` method performs the exact action of trading Remembrance of the Grafted while `menuDescription` is used to display to the player if Remembrance of the Grafted was traded successfully or not.

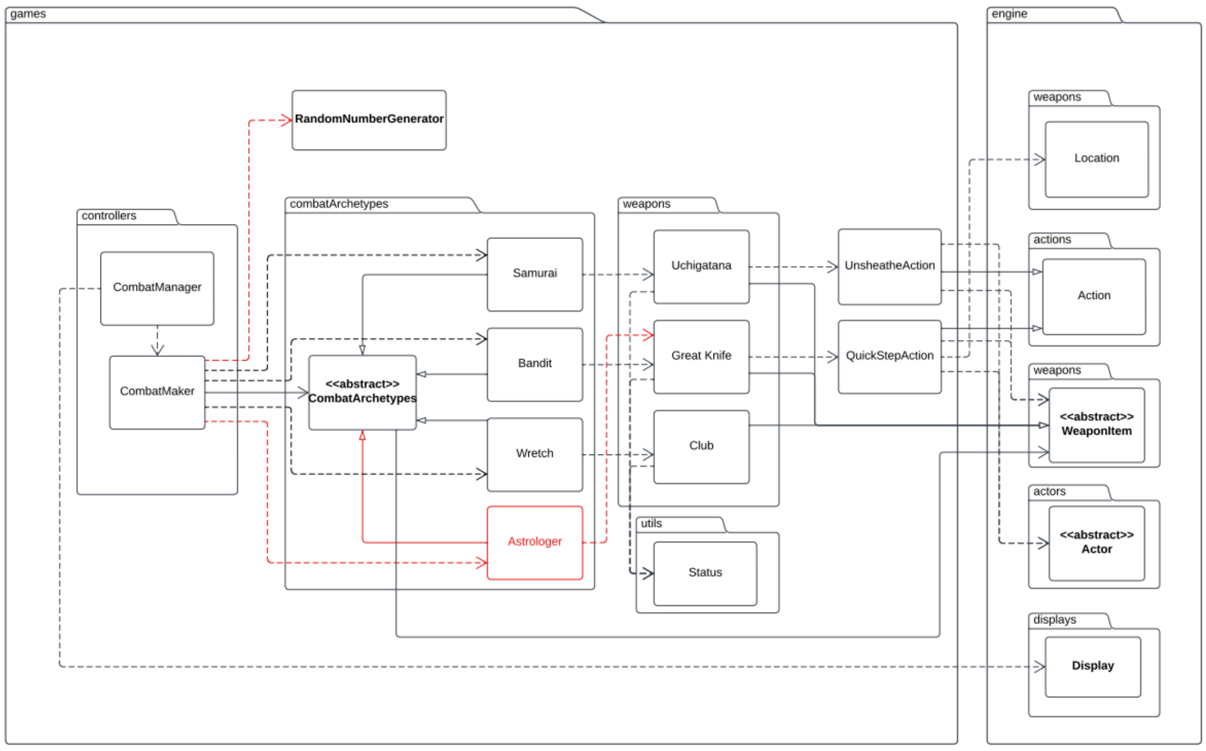
SellAction is updated to be an abstract class and SellItemAction and SellWeaponAction extends from SellAction. SellItemAction is used when Items are sold from players to traders whereas SellWeaponAction is used when Weapons are sold from players to traders. TradeAction is used when trading of weapons or items is done between players and traders.

SellItemAction extends from the Action class as SellItemAction is an action, so SellItemAction has all characteristics that the Action has. This has applied and adhered to DRY principles. `execute` method performs the exact action of selling Remembrance of the Grafted while `menuDescription` is used to display to the player if Remembrance of the Grafted was sold successfully or not. SellItemAction is a class used whenever an item is sold from player to any trader and in return, players will earn runes while SellWeaponAction is a class used whenever a weapon is bought from any trader by player with the player's runes.

In the future if Remembrance of the Grafted was to be given to players or dropped at random location on the game map then players can pick up Remembrance of the Grafted spawned randomly across the map with `PickUpItemAction`. Other than that, methods like `randomXDrop` and `randomYDrop` could be added to spawn Remembrance of the Grafted at random locations across the game map.

REQUIREMENT 4 (Red = changes made)

Part A

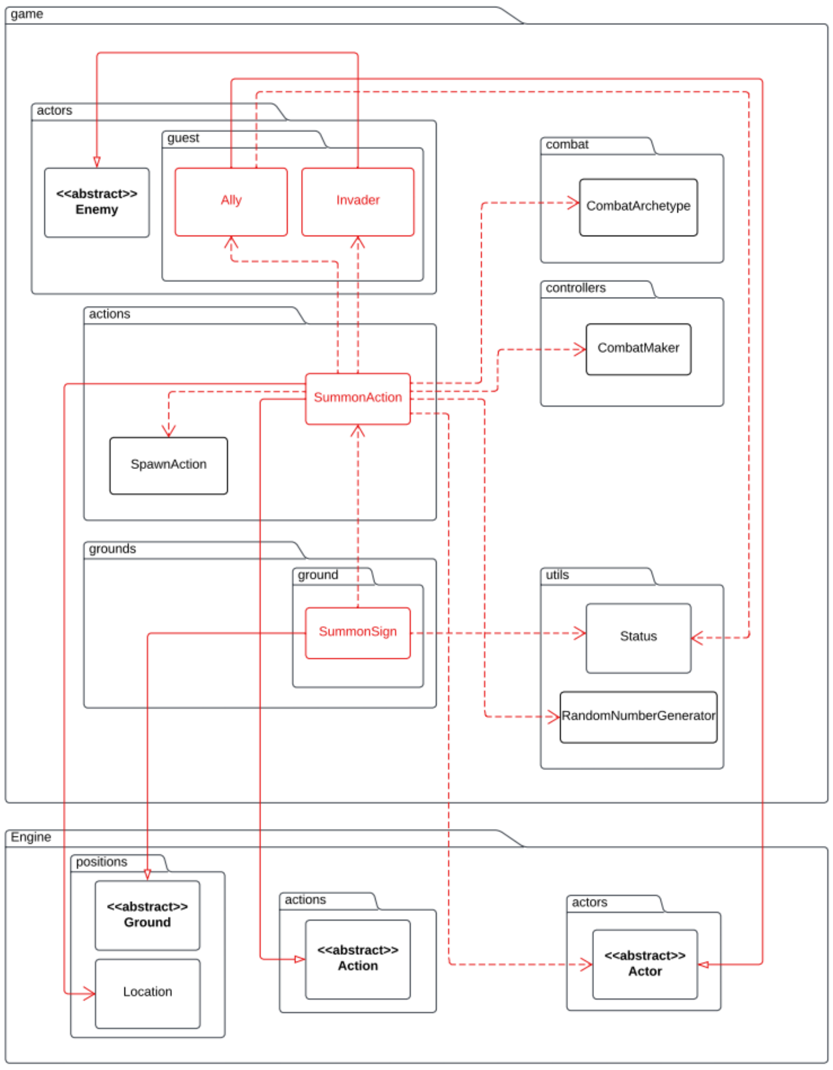


New Feature (optional weapon Astrologer's Staff is not implemented):

Astrologer extends CombatArchetype, and has dependency with GreatKnife. We did not implement the optional weapon (Astrologer's Staff) for Astrologer, therefore we initialise GreatKnife as the starting weapon for Astrologer. CombatMaker creates dependency to Astrologer in order to make the starting class for Player. This approach ensures that future new roles can be added easily by extending CombatArchetype and has dependency with CombatMaker. New method to create Astrologer will be added into CombatMaker. However, CombatMaker has to add another switch case in the `createCombat` function to accommodate for new roles. This would not be efficient if there are large amounts of new roles added, hence violating the Open Closed Principle.

New method `makeRandomCombat` added in CombatMaker. CombatMaker uses RandomNumberGenerator class to select random combat archetypes for Invader's and Ally's starting class in Part B by calling `makeRandomCombat` method. This function uses an arraylist that stores the character representing each role and passes it into `getRandomElement` in RandomNumberGenerator and it will return the randomly selected combat character. Then `createCombat` will be called to create the random role selected.

Part B



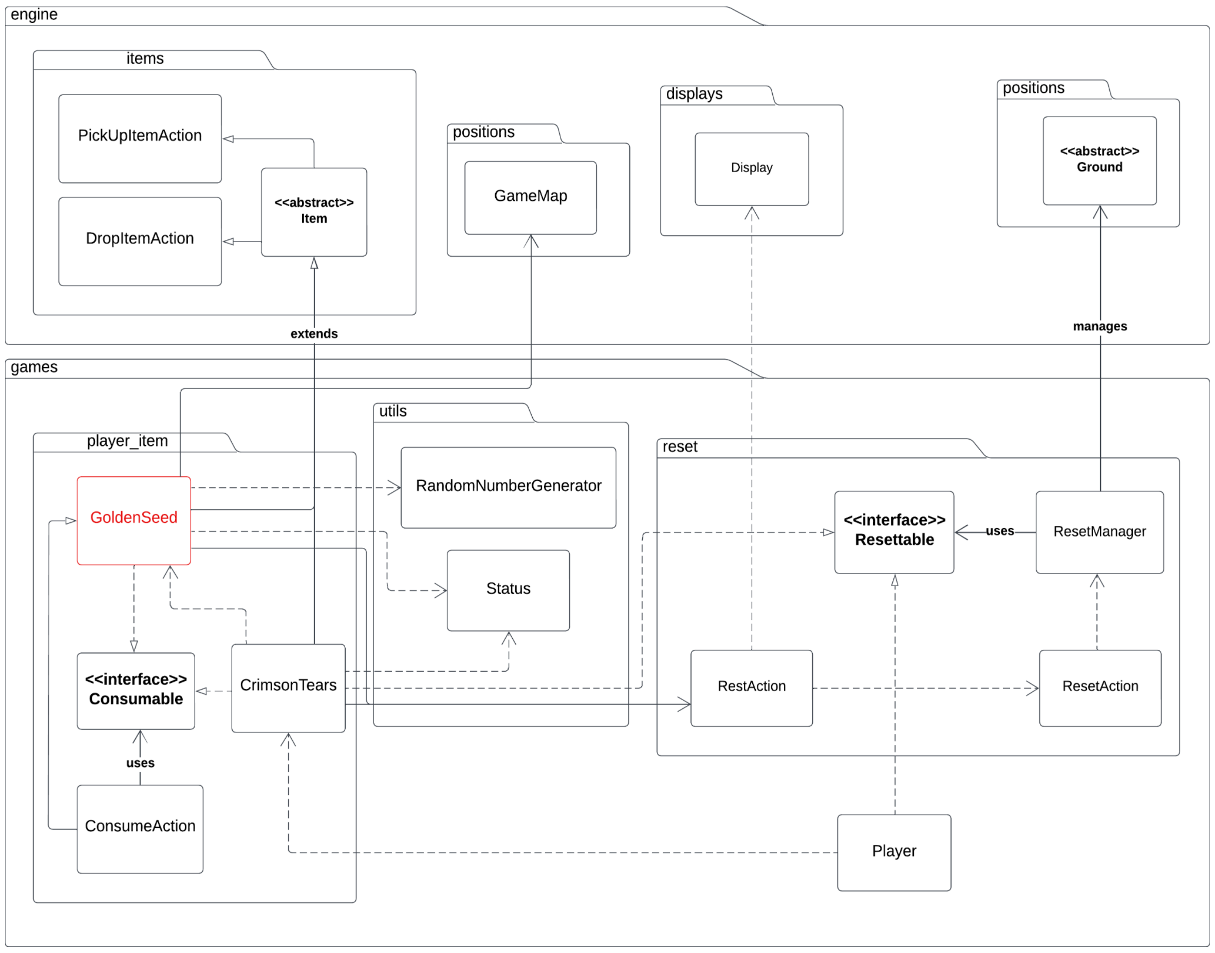
New Feature:

SummonSign is a new ground that extends from Ground. SummonSign creates dependency with SummonAction to spawn a guest from another realm. In the `allowableAction` method of SummonSign, it checks whether the actor standing on its surroundings is the Player by using **Status enum**. It then calls SummonAction to summon a guest (invader or ally), by using RandomNumberGenerator to obtain the chance for spawning Invader or Ally (50% chance for both). To initialise the starting class for invader and ally, SummonAction uses CombatMaker to initialise the CombatArchetype. Finally, it uses SpawnAction to spawn the guest on the location.

This approach adheres to the Single Responsibility Principle (SRP) where each class only has a single responsibility that does not interfere with each other. SummonSign does not summon the guest straight away in its class, instead, it calls SummonAction which has the clear definition responsible for summoning the guest. Whereas SummonAction is only responsible to determine the guest to be summoned, whether it is the invader or the ally, and initialise the guest with the starting class. Finally it calls SpawnAction to spawn the guest. None of the classes take on extra responsibilities, and therefore adhere to SRP, avoiding having a GOD class that does all the jobs for summoning.

Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules. Both should depend on abstractions. SummonAction depends on CombatArchetype instead of the concrete combat classes when initialising the starting class. The details are hidden by abstractions and SummonAction only needs to depend on the abstractions provided by CombatArchetype. Thus, this approach complies with DIP.

REQUIREMENT 5



GoldenSeed class extends from Item abstract class and implements the Consumable 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. GoldenSeed class was made as a single instance by invoking the `getInstance()` because although it can be spawned randomly across the map, it can also only be held by the player themselves. How many GoldenSeed a player has will change throughout the game as it depends on how many GoldenSeed players have picked up from the game map by themselves. Player starts with no GoldenSeed in hand and can pick up GoldenSeed spawned randomly across the map with `PickUpItemAction` and GoldenSeed is of Consumable item type so for players to consume GoldenSeed, players will use ConsumeAction and that the player can only consume one GoldenSeed every time ConsumeAction is used.

RestAction and ConsumeAction is updated to fit in GoldenSeed's implementation. ConsumeAction extends from the Action class as ConsumeAction is an action, so ConsumeAction has all characteristics that the Action has. This has applied and adhered to DRY principles. `execute` method performs the exact action of consuming the GoldenSeed while `menuDescription` is used to display to the player if GoldenSeed was consumed or not. RestAction is updated to display if the player would like to consume their GoldenSeed once the player steps into Lost Grace of Site.

Once GoldenSeed is consumed, players can have their number of Flask of Crimson Tears increased. One GoldenSeed used, players will have an increase of 2 more Flask of Crimson Tears. Players can use GoldenSeed anywhere throughout the game and is not limited to being only used in Lost Grace of Site but once player steps into the Lost Grace of Site, player will be asked if they would like to consume GoldenSeed.

GoldenRune class has only one instance at all times. This is to maintain GoldenSeed 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 which ensures reusability of the GoldenSeed class.

Players are able to drop GoldenSeed as an item onto the map whenever the player desires to do so, hence GoldenSeed would have the status of `PLAYER_DROP_ITEM` and would execute `DropItemAction` with the `execute` method and `menuDescription` method to display is GoldenSeed was successfully dropped onto the GameMap or not.

In the future if GoldenSeed were to be given to players permanently whereby players could not pick up GoldenSeed randomly across the map and players will have a fixed amount of GoldenSeed in their inventory then `goldenSeedAmount` would be the fixed number of GoldenSeed players have in their inventory. Other than that, methods like `randomXDropGoldenSeed` and `randomYDropGoldenSeed` and its usage could be removed from GoldenSeed and Application class to eliminate the possibility of GoldenSeed being spawned randomly across the Game Map while the GoldenSeed would not have the status of `PLAYER_DROP_ITEM` as since player would permanently have it in their inventory like Flask of Crimson Tears then once GoldenSeed is used up then it will be directly remove from the player's inventory. Due to GoldenSeed having it permanently in their inventory then GoldenSeed would not need to result in action like `DropItemAction` with the `execute` method and `menuDescription` method being included for players to choose from when dealing with GoldenSeed.