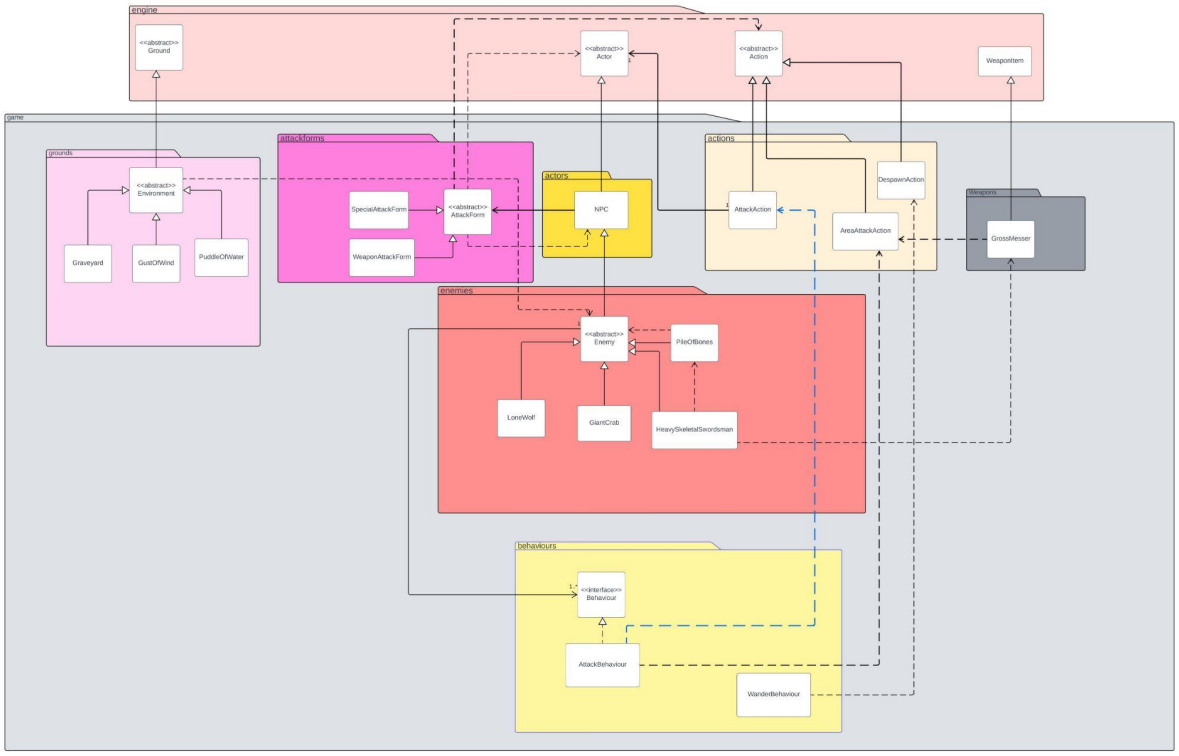
FIT2099 Oriented Programming Assignment 2 Elden Ring Object Design Rationale

By Matthew Ong, Tsun Law and Kenny Huynh

REQ 1



Design Rationale:

The Diagram represents an object oriented system for a part of a text based video game called Elden Ring. It represents the new enemies added, and the grounds that spawn them as well as a weapon that the Heavy Skeletal Swordsman carries. The Goal of this design is to be able to display how to extend the base functionality and classes of the game in order to implement different aspects of the game like enemies, weapons, grounds, in such a way that it adheres to the SOLID principles.

The 3 new types of ground are, Graveyard, Gust of Wind and Puddle of water, and they all extend from the environment abstract class, which in turn inherits the ground class. Since they are grounds which have a chance to spawn enemies, as well as they share common attributes and methods, it was logical for them to be subclasses of the ground class to avoid repetitions(DRY). I also added an environment abstract class to be the abstract class of the various grounds, with an abstract method spawn() that grounds can use to return the enemy instance they spawn. I did this so the subclasses all share the same tick method which spawns the enemy, and also to override the canActorEnter to false. I also mainly did this for the factory design method which I will cover in REQ 5. The way also generally allows for easy extensibility and follows the Open-Close principle , as we dont need to modify the

environment class if we want to add more special environments to spawn new enemies, we can just have it extend environment and define the spawn() abstract method. This follows DRY and Open-Close.

The different types of grounds have a dependency relationship with the respective classes they spawn, Heavy Skeletal Swordsman, Lone Wolf and Giant Crab in the respective order as we would need to return a new instance of the mentioned classes each time the enemies spawn.
There are 3 types of enemies: Heavy Skeletal Swordsman, Lone Wolf, Giant Crab that extends from a new class Enemy which extends from another new abstract class I introduced, the NPC class, which in turn extends from the actor class. The NPC abstract class represents common methods and functionalities that all non playable characters in the game have, for example a faction which they are part of, and using behaviours to get their turn actions.

The enemy class that extends NPC is meant to extend the NPC class for enemy functionalities and represents all enemies in the game, since there are things between them are different like:
- Dropping runes when dead
- Resettable
- Player can attack all enemies
- Follow actors with FOLLOWABLE capability

Having a subclass enemy class fulfils the liskov substitution principle as if in classes like attack behaviour uses NPC,we can substitute Enemy in and the functionality will remain as designed. It also allows for easy extensibility and follows the open close principle.

The NPC parent class has an association relationship with the behaviour interface, as each enemy has a behaviours hashmap as an attribute to store behaviours which are how the npcs' chooses what actions to execute. This dependency injection makes more sense as instead of having an association relationship with every possible behaviour added, which will cause the enemies to be dependent on too many classes, we just have it associated with any behaviour class implementing the behaviour interface.

Regarding choosing what type of attack action npcs return, I created a setAttackForm and getAttackForm method that retrieves and sets various attack forms for npcs. Npc class has an association with AttackForm abstraction class and the previous mentioned methods set and retrieve attack form classes. The attack form abstraction class allows subclasses to extend and implement the createAttackForm method according to what the attack form is supposed to be. For example special attack form .createattackForm method allows actors that are able to use a special attack with their intrinsic weapon or just a normal attack with their intrinsic weapon and returns whichever one accordingly, for example GiantCrab. All we have to do is define .getSpecialAttack in giant crab and set attack form to special attack form and the special attack form will call the getSpecialAttack method of the giant crab.

Building on this I then created an AttackBehaviour for npcs that calls the getAttackForm method when another actor is nearby. The attackform classes are mainly used for retrieving the attack type that npc's make so in Enemy class for allowable actions we can't use it to

return to player as we need to return every possible action the player's weapons can do so we just loop through weapon inventory and get all the skills. Back for AttackForm, although I am creating ana extra association and dependency between NPC and AttackForm, I believe implementing it this way allows easy extensibility and follows single responsibility principle as if I add more enemies to the game all I have to do is define the special attack if they have one, or regarding weapons, just pass in the weapon to weaponAttackForm and it will call the weapon's get skill if there is and that is pretty much it, it also significantly reduces repeated code and extends easily so I strongly believe these benefits mention outweigh the cons of adding dependencies.

In order to differentiate the enemy entities as well as other entities in the game without using instanceof, as it is bad practice, we decided to create a Faction enumeration capability that has values of CANINE, SHELLFISH and SKELETON. These values will be set using the setFaction method in of NPC.. This allows a more abstract way of checking during the attack action if the target is of the same faction as the attacking actor. Implementing it this way allows us to extend the code in an easier , more abstract way as multiple actors can be classified under the same faction by just simply setting the relevant faction in subclasses(Open-Close principle). This is a better alternative to instanceof and creating a different abstraction or parent class for each faction as that might lead to many unnecessary and extra inheritance which can be difficult to maintain in the long run.

For the despawning mechanic, we decided to add a check in the allowable actions method of enemies, checking if other actor has followable status capability and if it isn't already being follow by that instance, if so we then just add follow behaviour into the behaviour hash map. This makes more sense than our previous implementation of using despawnable interface as it was redundant, in this new method we can reduce dependencies with other classes.

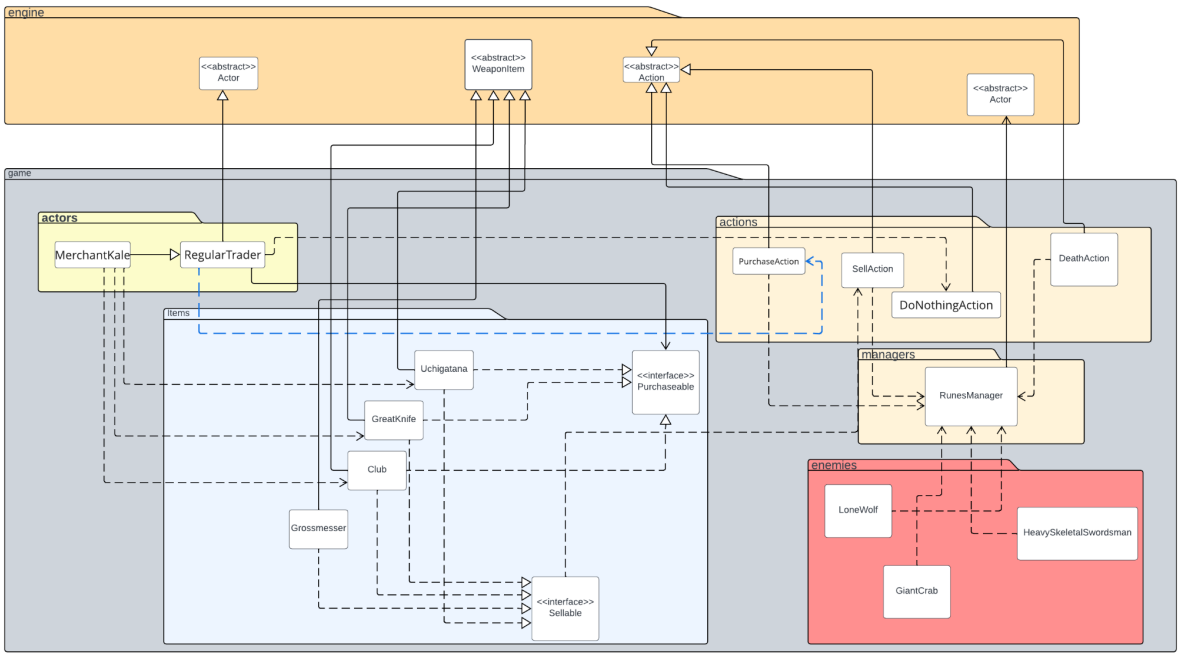
The Heavy Skeletal Swordsman is special as they have the ability to turn into pile of bones when they reach 0 Hp and if they are not destroyed within 3 turns, they revive back with full health. We implemented the mechanic of checking if a skeleton is unconscious in the playturn method, and then create a pile of bones class with the current skeleton type passed in. This is one of the advantages of creating an abstract enemy class as it accepts both SkeletalBandit and HeavySkeletonSwordsman and prevents the need to check their instances.

Inside pile of bones we essentially just have a counter attribute to keep track of game turns , so we can revive the skeleton instance inside the playTurn method of pile of bones. Another advantage of above-mentioned abstraction. We also pass in the relevant weapon to the pile of bones weapon inventory as different skeletons hold different weapons. When pile of bones die, they then can just drop all weapons in the weapon inventory and remove the class from the map. This was a much better way than our previous break action method etc, much less code and much easier to maintain and still follows SOLID principles like Single Responsibility.

Heavy Skeletal Swordsman also has a dependency with GrossMesser, as we add it to the inventory of Heavy Skeletal Swordsman in the constructor. GrossMesser is a weapon class that is extended from weaponItem class. We extend from the weaponItem class to utilise

the drop and pick up actions as well as the getSkill() method which will return a custom AreaAttack action inside getAttackStrategy method. The AreaAttackAction attacks everyone in the surrounding area by creating an attack action for each surrounding actor and executes them all. I made it a custom action as the GiantCrab also has a special ability to attack everyone in the surrounding area with their slam, so by creating a common action, we minimise the need for repeated code(DRY).

REQ 2



Design Rationale:
The Diagram represents an object-oriented system for a game called “Elden Ring”. The goal of this design is to show the extended abstract and extended class in order to implement the runes and trade system between the player and MerchantKale.

There are 4 types of actions that will be used in requirement 2 that extend from the “Action” abstract class which is the “PurchaseAction”, “SellAction”, “DoNothingAction” and “DeathAction”. As actions are how interactions happen in the game, and the new classes need to make use of the methods Action class has, they inherit from the action abstract class, as well as following the overall design of the system.

There is one new type of manager in requirement 2: “RunesManager” that is a class which manages all the runes. It has a dependency with “DeathAction " as when an enemy dies and DeathAction method is called, we will need to transfer runes dropped to player immediately, so this method will be called inside DeathAction. We created a RunesManager so it can both be used for when an enemy dies and the runes go to the player, or when purchasing or selling items to MerchantKale, which allows easy extensibility. This sort of design allows for code extension but not code modification which follows the Open Close Principle(O).

Additionally, RunesManager has dependency with”LoneWolf " , “GiantCrab " and “HeavySkeletalSwordsman " class in order to initialise the deathAmount of runes so that it can be dropped when enemy dies. We will only add the death amount once “HeavySkeletalSwordsman” becomes a pile of bones Since they will only be killed in order to add the death runes amount when they become a pile of bones. Additionally, RunesManger has an association with Actor class as RunesManager stores all the runes of each actor in a hashmap using actors as keys.

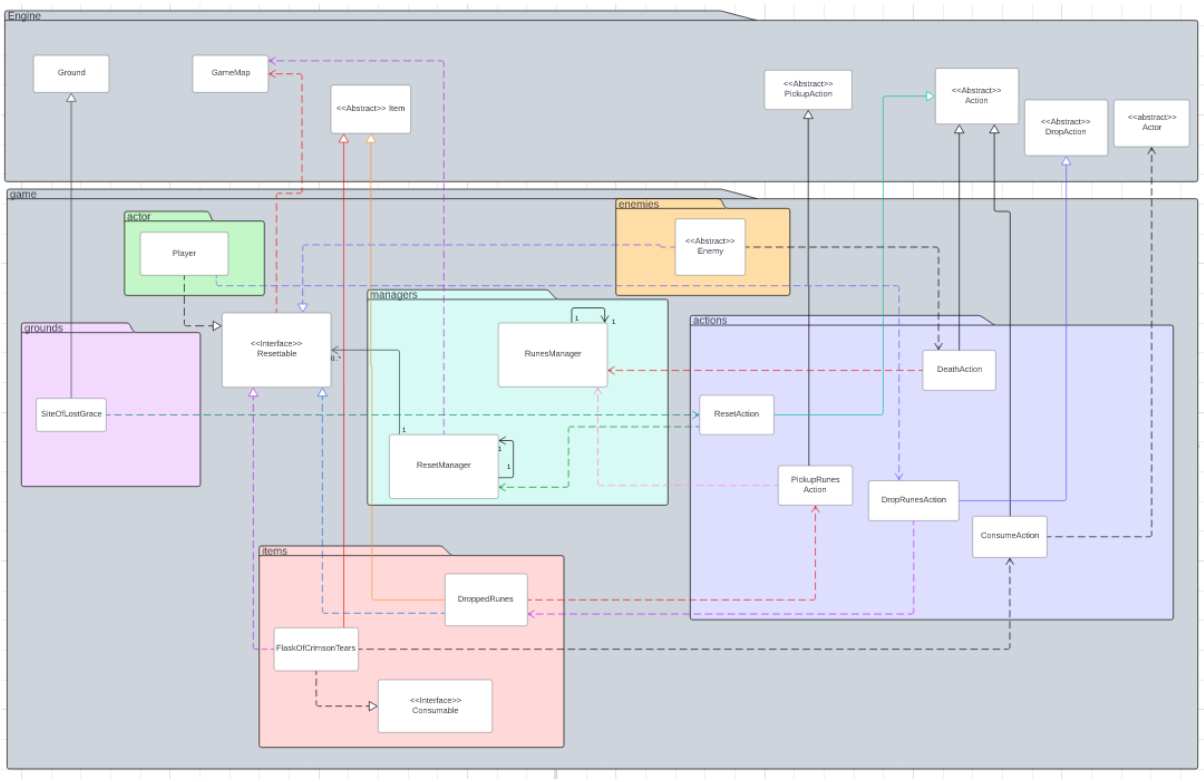
I acknowledge the use of [1] ChatGPT (https://chat.openai.com/) to [2] generate materials for background research and self-study in the drafting of this assessment. I entered the following prompts on 25 April 2023: Can you explain how the hashmap works in java? The output from the generative artificial intelligence was adapted and modified for the final response.

There are 4 types of items: “Uchigatana", “GreatKnife" , “Club” and “GrossMesser” that extend from the “WeaponItem" abstract class, as the player can deal damage with these weapons and certain weapons have skills, it makes sense to extend the weapons abstract class. Additionally, they all implement “Sellable " interface and “Purchasable " interface except for “GrossMesser " as it is not available for purchase. As our design is to write the purchase and sale price in each weapon class and we implement these two interfaces for the Override method in each weapon class and return it. One of our previous ideas was to create one interface for sale and purchase price but it is a better design to separate two interfaces(ISP). For example, the "GrossMesser" class does not implement the "Purchasable" interface because it is not available for sale. The design adheres to this principle as clients are not forced to depend on interfaces they do not use (ISP).

For the “Sellable” interface, it has a dependency with the “SellAction”. Our previous design had us violating DRY and open-close by using the tick method of weapons to check if merchant is nearby and add sell actions, which is why we changed it so that the merchant returns sell actions as allowable actions instead but in order to do that we had to use downcasting to make the weapon passed in a sellable instance. Even though using downcasting is not a good approach for programming, using downcasting in the regular trader class seems to be a better alternative approach in order to sell non-weapon items. Indeed this approach should be put in a high level class in order to avoid duplicating the code in different places. However, there are weaponItem, weapon, item and different abstract classes already, having another abstract class that only serves for the “SellAction” class seems not ideal and it may trigger a warming of over-abstraction. We need to consider weapons that cannot be sold and not require the “SellAction” if we put it in the WeaponItem class. Although it may violate some perspectives, for example, it may violate the Single Responsibility principle by not creating an abstract class to serve it. However, including it in the design outweighs the disadvantage of unnecessary abstraction. This also beats our previous design of sell actions and using sellable interface and making the code adhere to Open-Close and SRP and ISP. So in this case I believe downcasting is strongly justified and outweighs the cons.

We introduce a new parent class “RegularTrader” as the parent class to “MerchantKale” . Our original idea is to only create one single merchantKale class but what if we plan to add more merchants to the game? Therefore it is a better alternative for easily extensible if we plan to add more merchants to the game. Since "MerchantKale" is a subtype of "RegularTrader" and can be used interchangeably, this design adheres to this principle as all the subtypes are substitutable for their base types (LSP).Also “RegularTrader” extends from the “Actor” abstract class as it is the character inside the game, therefore they won’t repeat the same methods the actor class already has (DRY). The `RegularTrader class has a dependency with purchase action. It is because in the allowable actions method, if we check an actor with the CAN_INTERACT_TRADER capability is around RegularTrader, it will return “PurchaseAction” for every item in the trader's inventory. This design adheres to the single responsibility principle since “PurchaseAction” and “SellAction” are only in charge of their own respective actions. Furthermore, in purchase and sell actions if executed, they will call the RunesManager, to perform the transaction of runes between the actor and the merchant which displays the advantage I mentioned earlier of not having to rewrite more code and repeat yourself(DRY). Additionally, RunesManger handles the removal and addition of items in respective inventories. Originally we wanted to separate both of these functionality Our first idea was to separate this into 2 actions as we wanted to adhere to Single Responsibility Principle. However, it seems redundant to create TransferItemsAction and TransferRunesAction in our last design hence we made the transition of creating a RunsManager class for managing all the runes when it comes to purchase and sale which seems to be a better alternative. Another original idea we had was to also create some method in the player class that handles the rune transfer and weapon transfer but our current design is a much better alternative as it allows easy extensibility as what if we wanted to add a new actor that can also transact with MerchantKale? We can then just add the CAN_INTERACT_TRADER capability and that’s all we have to do which is really efficient. Additionally, our original idea is to create one “Trader” class but what if we want to add another trader that can perform something different to the current trader. Therefore we created a “Regular Trader” class instead. Additionally, RegularTrader has an association with Purchasable interface as Merchant Kale has an array list to store inventory of purchasable items. Originally we wanted to use weapon inventory but there is a better alternative to make a trader inventory instead of using the weapon inventory as we need to get the purchase price and make use of the purchasable interface purchased by method. There is also one dependency between “RegularTrader” and “DoNothingAction” as there is an Override playturn method inside the merchant class and RegularTrader is not supposed to do anything therefore it will return the DoNothingAction.

REQ 3



Above is the UML diagram for requirement 3.Requirement 3 of our game will mainly cover resetting the game, outline a few aspects of player death and a new concept “The Site of Lost Grace” which allows the player to reset the game by their own will and “The Flask of Crimson Tears” which is an item they player will have with them throughout the game.

First, the Flask of Crimson Tears will inherit from the abstract item class as to not repeat some of the methods the item class already has(DRY).This flask will not be able to be picked up or dropped and will start in the player’s inventory of items hence, it will not be portable, this is an advantage of inheriting the item class since they already have that attribute. This flask is initialised inside the Player constructor ,which has the advantage of not requiring the player to pick up the flask which is impossible due to it not being portable.

Logically it will also implement the resettable interface such that in the case that the player decides to reset the game, the uses of the Flask will reset back to 2. The Flask's sole responsibility is to pass the player the ConsumeAction into their menu.(S). Furthermore, as one of the items that are interactable in the player's inventory, we require the new ConsumeAction which extends the Action class so that it can be passed into the Flask's allowableActions list as an Action to be displayed on the menu (LSP). We also created a Consumable interface that can be implemented into items that has some sort of consume effect, we just have to individually define the consume() method in consumable interfaces to implement the required effect. TheConsumeAction will then have the sole responsibility when executed to execute the consume() method in the classes that implement consume(). So for a flask of crimson tears the consume method will restore the actor's health by 250 and decrease the flask usage in that actor's inventory by 1(Single responsibility principle). We made a consumable interface so it allows for easy extensibility as we only have to define the consume() method in classes that implement Consumable(Interface segregation principle, Open-Close principle).

The Site of Lost Grace is a unique ground that allows the player to reset the game upon resting near it and logically would inherit from the Ground interface to avoid repeating code for Ground methods(DRY). The site's sole responsibility is to pass the RestAction Action to be displayed in the player's menu(S). The ResetAction when executed will call upon the ResetManager's run method which loops through all the resettables inside and executes their respective reset() methods.
An advantage of this is that we generalise the site's action to a ResetAction and not RestAction previously so that ResetAction can be the generic Action to reset the resettables in the case that there's a new class that also wants to reset the game in a similar fashion.
It is important to note that the ResetManager is a singleton class , this was a logical choice as we will only need one ResetManager to keep track of the resettables and allow resting and player death to reference the same ResetManager instance. The ResetManager contains a static method getInstance that has the advantage of allowing it to be called anywhere but most importantly in the cases that the player rests (RestAction) or when they die they will call the same ResetManager.

Under the assumption that all enemies are resettable, instead of implementing the resettable interface individually to each enemy we implement it in the Enemy interface to prevent repeated code which is advantageous in reducing repetition (DRY) and also ensures that future enemies implemented will also be resettable(O). The reset() method in these enemies will simply remove them from the map. We have all enemies register themselves into the reset manager as opposed to having the Environment Class do that with the advantage to reduce dependencies. The alternative of course was that since all enemies are generated by the Environment class. The Environment class can also call the ResetManager to register the Enemies it spawns into the manager however this was not implemented because we are trying to reduce dependencies. Another alternative

method would be to make the grounds that spawn those enemies resettable instead. Using this method, the ground will contain a list of actors it has created and when reset() is called it will loop through that list and remove them from the map. However this would violate the sole responsibility principle as well as ensuring that these specific grounds are added into the resetManager which would involve downcasting.

As per requirements the Player is reset back to their last visited Site of Lost Grace on death with full hp, it made sense for the Player to implement resettable and have the method restore player's hp to full.

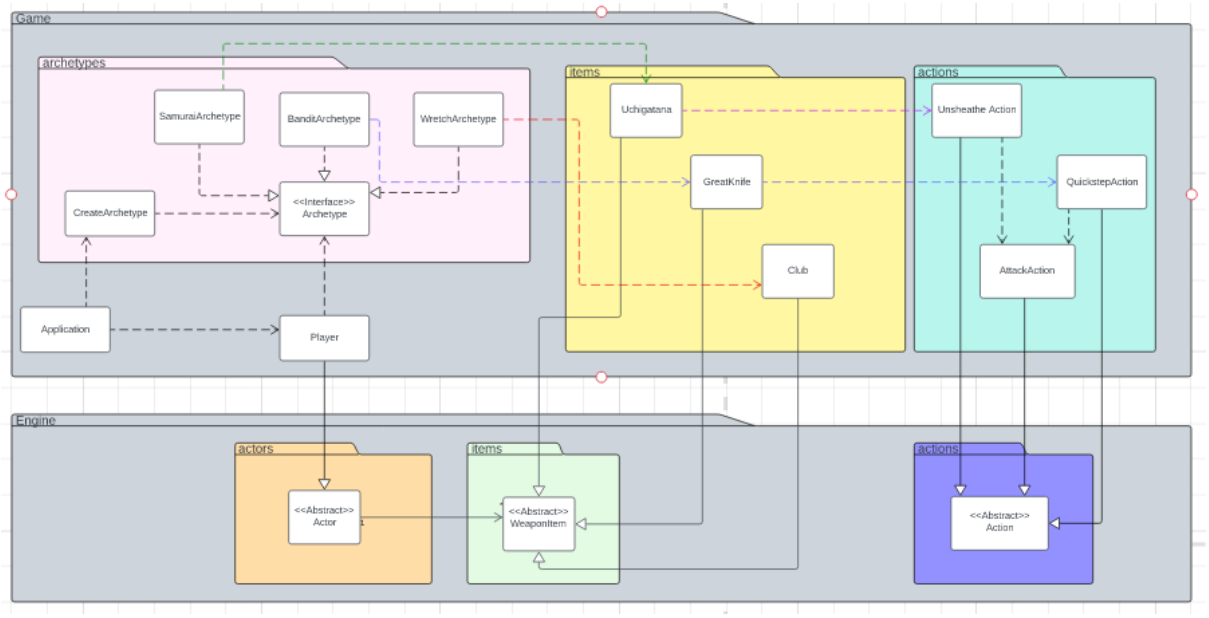
The moving the player to spawn is handled by the player themselves as we check whether or not they are conscious in the playturn method, we also return the move action here to prevent playturn from continuing to offer the player choices despite them being considered dead. This was originally done by the death action however, this comes with the tradeoff of introducing extra dependencies which we are trying to avoid. We also don't violate sole responsibility this way because the playturn method's job is to manage all the activities that happen per turn.

Lastly, when the player dies, their runes will be dropped on the floor and can be later collected if they make it back.These dropped runes are implemented in the DroppedRunes class that inherits the Item abstract class and is non portable. It will pass on the PickUpRunesAction that inherits the PickUpAction class. This action when executed will update the player's rune count using the RuneManager and has the advantage of removing itself from the map from PickUpAction. This again comes into play in the Player's playturn method, which handles the events of the player's death where this DropRunesAction is called there. An alternative is having the deathAction handle this however since dropping runes is only specific to the player it wouldn't make much sense to be constantly checking whenever a non-player dies, an advantage to this approach is that RUNES_DROPPABLE will apply to any actor that is capable of dropping runes and hence not limited to the player but makes no sense in the context of our requirements.

To handle the event where the actor dies before reaching their dropped runes, the Reset Manager contains a list with objects that ONLY reset on death. This again ensures that the Reset Manager has sole responsibility of resetting the resetables. The previous approach was to have another manager handle this but to reduce dependencies we moved this functionality to the Reset Manager since it's role is to reset.

An alternative approach would be to implement the droppedRunes as a ground, similar to pile of bones upon player death but this design seemed not to comply with giving the droppedRune sole responsibility of simply giving the player back their rune count. This would involve the droppedRunes replacing the ground it was dropped on and adding it back when collected, which can potentially be prone to incorrectly replacing the ground back.

REQ 4



Above is the UML diagram for requirement 4.Requirement 4 of our game will outline the player's choice of classes/combat archetypes that determines how much health and what type of weapon the player starts with.
These archetypes will be classes that implement an Archetype Interface that has two methods to retrieve the starting hitpoints and weapon of Samurai, Bandit and Wretch. This interface also currently only has two required methods that are needed following the interface segregation principle (I).This abstracts these current classes to return their one respective weapon, should in the future there be an archetype that is similar to the Samurai but starts off with more hit points and weapons, then it can be extended from Samurai(O). An advantage of this approach is that it's simple, and the player can still be treated as a player by the engine(LSP).
Since it is the player that will be selecting and assuming these classes, we'll have to modify its constructor to take in an Archetype, and inside that constructor call the methods to get the player's starting hitpoints and weapon.
An alternative approach would be to have the Archetypes extend from the player class itself, the different classes Samurai, Bandit and Wretch will inherit from the player class and set their initial health and weapon respectively however it's the player class that as per specifications is being controlled and not the Samurai, Bandit and Wretch.

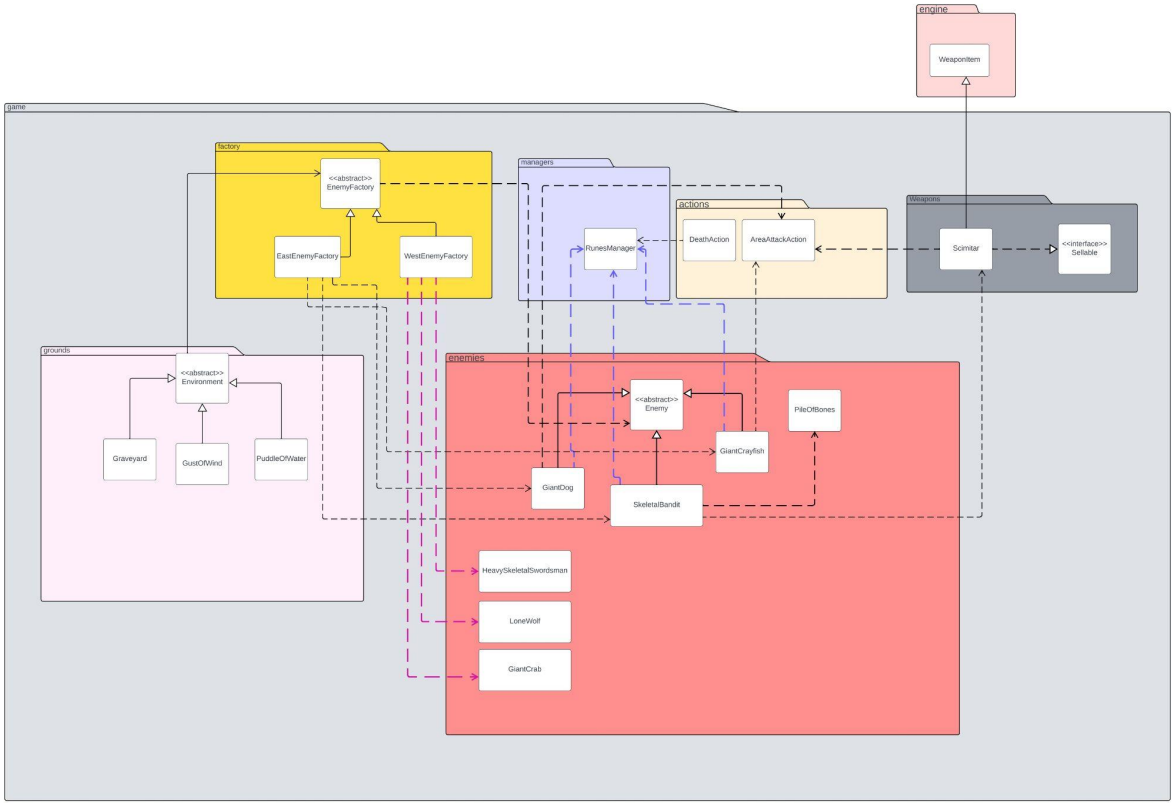
These weapons will also inherit from the weaponItem class as they function similarly but with differing hit rate and damage values.
The Uchigatana and Greatknife have their own unique skills Unsheathe and Quickstep, which will be implemented as new Action classes with the sole responsibility of handling the execution of those skills (S).

When it comes to implementing the UnsheatheAction, we've created a constructor in AttackAction that takes in values that can multiply damage and accuracy, we can adjust the damage ratio and "modify" the attack. This is used by the UnsheatheAction where the input is 2 times the current damage of the Uchigatana and a 60% hit chance. This attackAction can also be used by future weapons items where skills can have a negative effect on the damage you deal and be variable, hence we've created this constructor so that it can be generally used by any weapon(O). Of course the alternative was to straight implement the UnsheathAction extending from Action, but of course, Unsheath will always do 2 times the damage at a 60% hit rate this comes with the disadvantage in the case that there's another weapon that may want to do 1.7 times it's damage and would mean we'll need to make a whole new repeat class similar to UnseatheAction and changing the damage to 1.7 times.

In regards to the QuickStepAction, we know that it will require access to the user, target, weapon and direction. We can extend this class from Action and introduce attributes Target, weapon and direction that can be passed when the QuickStepAction is created. When it's executed, like all Action execute methods, it takes in the actor that executes the Action and the GameMap instance. So it was logical that the execute method of QuickStepAction create its own AttackAction to damage the enemy, then check for the first open area and move the actor using the map.moveActor method. This design gives QuickStepAction its own responsibility to carry out the attack and move the actor(S).

The handling of these skills require the WEAPON_SKILL enum but also check for TARGET_WEAPON_SKILL to differentiate between weapon skills for combat or general use. This is because the WeaponItem abstract class contains two getSkill methods, one for attacking and general use. This check is handled in the Enemy's AllowableActions that will give the player the option to use the skills if the conditions are met. Since the weapon item also inherits from item when the weapons are created these enums are passed into their allowableActions list to later give the player the option to choose to use these skills.

REQ 5



Design Rationale:
The Diagram represents the object oriented system for Requirement 5, since requirement 5 is very similar to requirement 1 with just a few additions I will only mention the new relationships and classes added.

There are 3 new enemy classes that have been introduced into the game that spawn from the existing environment subclasses Graveyard, GustOfWind, PuddleOfWater. On the east side of the map, GiantDog which spawns from GustOfWind, SkeletalBandit which spawns from Graveyard and GiantCrayfish which spawns from PuddleOfWater.
These new enemies are of the same type as HeavySkeletalSwordsman, LoneWolf, GiantCrab respectively and as such also have their corresponding faction enum so they don't attack npc of the same type as mentioned in REQ 1, which showcases the easy extensibility without needing to add dependencies with extra classes. They also all extend from the same Enemy parent class which shows the easy extensibility for creating enemy subclasses(Open-Close).

The SkeletalBandit have the exact same relationships and functionality as the HeavySkeletalSwordsman, the only difference being they have a different WeaponItem that they carry called the Scimitar and have different stats. So skeletal bandit will have a dependency with Scimitar and Scimitar will have a dependency with AreaAttackAction since

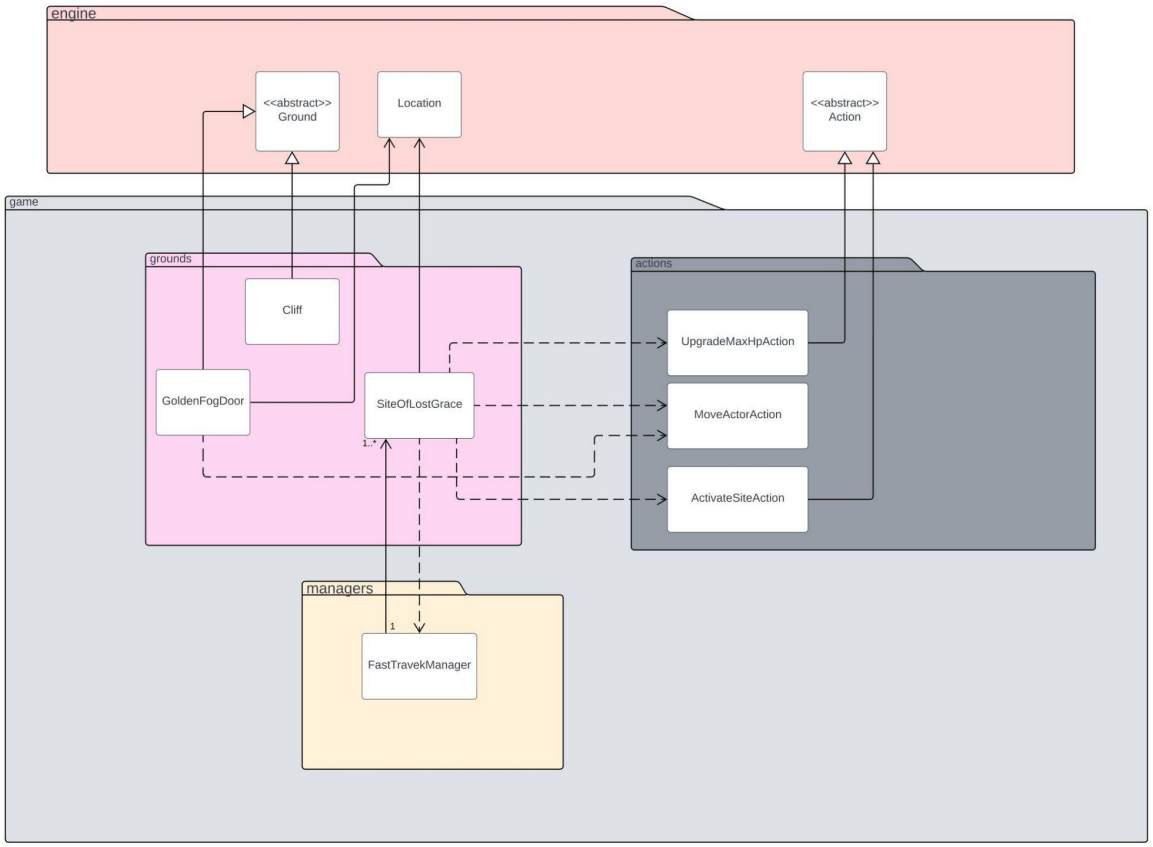
it can also perform a spinning attack, which is returned in getAttackStrategy. The pile of bones mechanic is the exact same as for HeavySkeletalSwordsman.
The SkeletalBandit is also part of the SKELETON Faction and can also turn into a pile of bones.

Since giant dog also has a slam area attack we can just override the getAttackStrategy method to add a new area attack action into it, hence the dependency between them, same goes for GiantCrayfish. This is one of the situations which displays the advantages of the way we designed it, it makes our code easily extensible so if any new npcs can perform the same area attack, we just have to add the capability making extending the functionality extremely seamless instead of modifying each individual class(Open-Close principle).

The biggest change would be implementing the splitting of maps and making the different grounds spawn different enemies based on the side. We decided to implement this with the factory method design pattern, which is a way to create objects without specifying their concrete type. We created an abstract EnemyFactory class that has two subclasses, WestEnemyFactory and EastEnemyFactory. The abstract methods are createSkeleton, createCanine, createShellfish. We abstracted these classes so that when creating the environments, you can pass either and it will be accepted in the parameters. In the environment subclasses all we have to implement is 1 method, spawn(), and it just returns the enemy subclass that corresponds to the relevant region, upholding the single responsibility principle as environments should be in charge of deciding what objects to create. So in graveyard, gust of wind and puddle of water it is just calling the enemy factory .createSkeleton, .createCanine, .createShellfish methods and it will return the relevant enemies that spawn in the associated regions. In the tick method of environment we are just calling the spawn() method and checking if there is an enemy returned and it can be spawned in the surrounding area. Designing this way allows greater flexibility in creating objects, and the abstraction allows the code to be more modular and easier to understand. As when passing in we can simply define another type of factory and utilise that to create objects. Which follow the open for extension and closed for modification principle(Open-Close principle).

FIT2099 Oriented Programming Assignment 3 Design Rationale

REQ 1



Cliff:

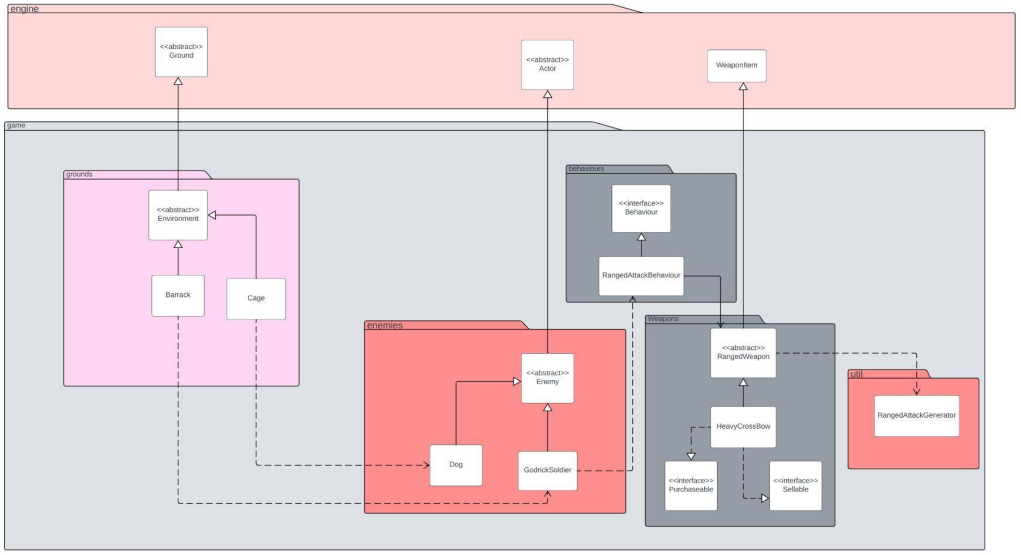
Cliff is a new ground that kills the player when walking on it. All we did was have it extend regular ground and edit the tick method so it constantly checks if the actor standing on it can be affected by cliff and we hurt it by the max value of integer, as player's death action is handled inside play turn itself and DeathAction class is mostly used for npcs. We used .hurt() so that the player reset will be triggered in playturn, as using death action will mean we have to implement player death logic in death action and that would cause it to become too convoluted.

Golden Fog Door: The golden fog door is a new requirement that allows the player to travel from different maps or locations. Since this is a tile that the player can interact with, it's only logical to make it extend ground. Within the constructor we can pass in the Map, x and y coordinates to be passed into its MoveActorAction as a destination. This allows Golden Fog Door to have the sole responsibility to pass on this MoveActorAction as well as allowing the doors to be interacted with as a ground (LSP) and be passed through using the canActorEnter method. This also allows MoveActorAction to remain responsible for actually moving the player. An alternative was to replicate something similar to the rocket in the Mars demo but this rocket was actually an item that the player could pick up and it contextually made no sense in this case for a player to pick up the golden fog doors. According to the

demo it was also important that we displayed the map to where the player would be going to using the door, our approach was to simply pass in that as a string into the constructor. An alternative was to extend further from golden fog doors into a specific map door for example Limgrave Door. However this would mean that we would have to repeatedly make new door classes for every new map which would violate DRY.

Fast Travel:
In order to differentiate different sites, we added a name attribute to each site of lost grace so we can print out the right name for menu description, we did not create subclasses to represent each different site as that would be unnecessary inheritance. In the site allowable actions we check if the current site has the CAN_REST capability meaning player has activated it, if it has not we return activate site action, which when executed just adds said capability to the site and adds it to the fast travel manager static list of discovered sites, which can only be accessed once player has reset on the site at least once, does a check using boolean attribute rested. If it already has the capability then we add ResetAction which returns the reset action to the player, and registers this spawn site as the latest rest site in reset manager each time player rests. Although we created a separate manager to store the fast travellable sites, I think its necessary as our only other alternative would be to have the static list inside the site itself which violates SRP. We also just reused the move actor action for each site that can be fast traveled to, instead of creating another travel action as it would be unnecessary to create a Travel Action and violating DRY.
This design allows easy extensibility as all we have to do if we introduce a new site is to just create the site instance and input the name and that is pretty much it, which follows open-close principle.

REQ 2



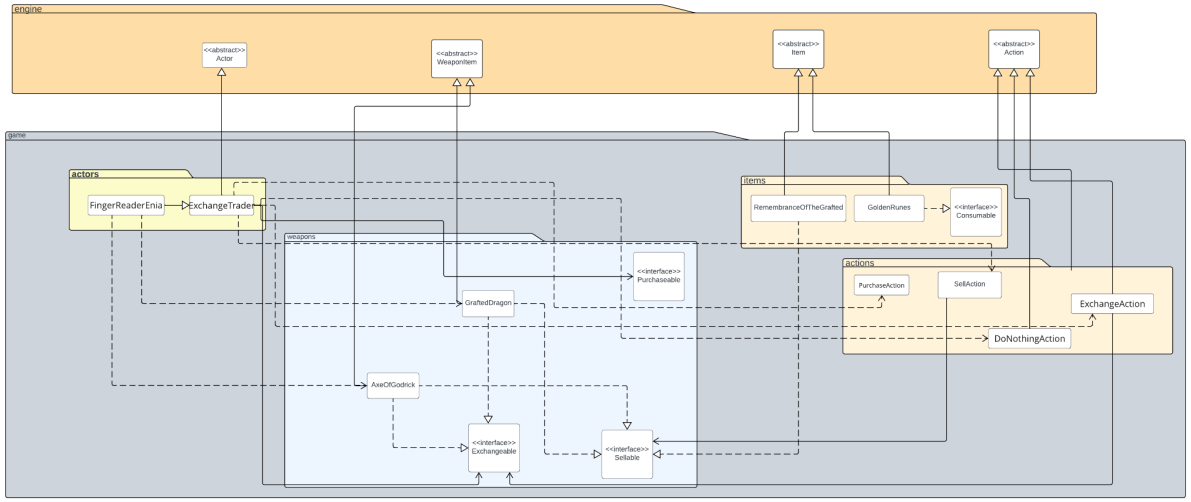
Grounds and enemies:
The 2 new grounds that are introduced in the stormveil map are Cage and Barrack, we have these classes extend environment as it has the functionality of spawning enemies(DRY). Inside the new grounds classes itself, in the implemented spawn() method we just calculate the spawn chance and return the enemy classes that it spawns so Barrack spawns godrick soldiers and cage spawns dogs. This shows the easy extensibility of our design and follows open-close principle and single responsibility principle. This is also why there is a dependency between the grounds and their enemy spawns. However I get that technically Dog is a canine species as well like based on real life and general logic, but the way I set factions in this game is based on the game logic so hence I set both Godrick Soldier and Dog to CASTLE_ENEMY faction, so that dog will still atttack lone wolf and giant dog.

Heavy crossbow OPTIONAL:
Since the engine does not support ranged attacks, I created a raned weapon class that extends weaponItem. I created a new class called RangedAttackGenerator that has a static method called getRange which allows any other ranged weapons to just pass in the range of that weapon and the weapon itself and location of the holder, and runs an algorithm to calculate all squares in the surrounding range. There was nothing in the engine code that supported ranged attacks so we felt this static method and separate class was necessary in holding the logic of calculating the algorithm. The method then returns an action list of attack actions on the surrounding actors if they are able to attack them. In the ranged weapon class tick method we essentially remove all allowable actions from previous game turn and add new ones returned by the RangedAttackGenerator. In the ranged weapon class there is also a getRange() method to retrieve the range of the ranged weapon in order to use it in RangedAttackBehaviour which is a new behaviour I created to use for ranged attacks. I

created a new behaviour as since behaviours are how npc's make their turn I feel it is necessary as the other alternative would be to have to check the actions passed in in playturn if its the instance of attack action returned from tick method of the weapons itself which would be worse practice. So the tick method in this case is mainly used for player which hence it first checks if the actor holding it is a playable actor. One con is creating new dependancies and association but I think this provides a more extensible way of extending ranged weapon functionality.

Also I wanted to mention this although its not related to this requirement per se but I dont know where to put it, I added a playable faction capability to identify player as my previous way of identifying player was using a combination of HOSTILE to enemy status and Can summon status, so I feel this is a cleaner way and I am taking into account as what if we add a co-op mode in the future, we can just assign the same capability and game will work as intended.

REQ 3



Upgrading Hp:
There is not much to this requirement all there is to do is to create a new UpgradeMaxHp action that can be returned in allowable actions and inside we just call the static public method increaseHpUpgradePrice that increases the price to upgrade hitpoints and actually increase player maxhp in the action. I made the attribute and methods static as it should be shared between all instances of site of lost grace, keeping the logic of upgrading hp in site of lost grace class adheres to SRP.

Finger Reader Enia:
We introduce a new parent class “ExchangeTrader” as the parent class to “FingerReaderEnia”. We created a “ExchangeTrader” in order to represent traders in the game that mainly deals with exchanging items. The idea is to separate different trader types into different classes, making it easier to extend and to maintain and clearly define what each trader is supposed to do(Open-Close, SRP). The “Exchange Trader” has a new inventory list of Exchangeable interfaces which subclasses can just add the weapons they allow for exchanging to. In relation to exchange trader, hence the association. We created a new Exchange Currency enum in order to be used in exchange traders to set their desired currency, which easily allows us to identify if the actor nearby has that currency or not by adding the enum to the currency item itself and checking through the inventory. If actors who can interact and at least have one item of the desired currency are nearby, we return an exchange action for every item in the exchangeable inventory. This design allows easier extensibility as subclasses just need to add the weapons that can be exchanged to the exchangeable inventory and have those weapons implement the exchangeable interface. Inside exchange action we just call the exchange method of the exchangeable item. This design adheres to the single responsibility principle since “ExchangeAction” and makes sure exchange action is mainly dependant on exchangeable interface. If we want to add future exchange traders we just have to extend from exchange trader and define which items that can be exchanged for and define their desired currency and that's it(Open-Close).

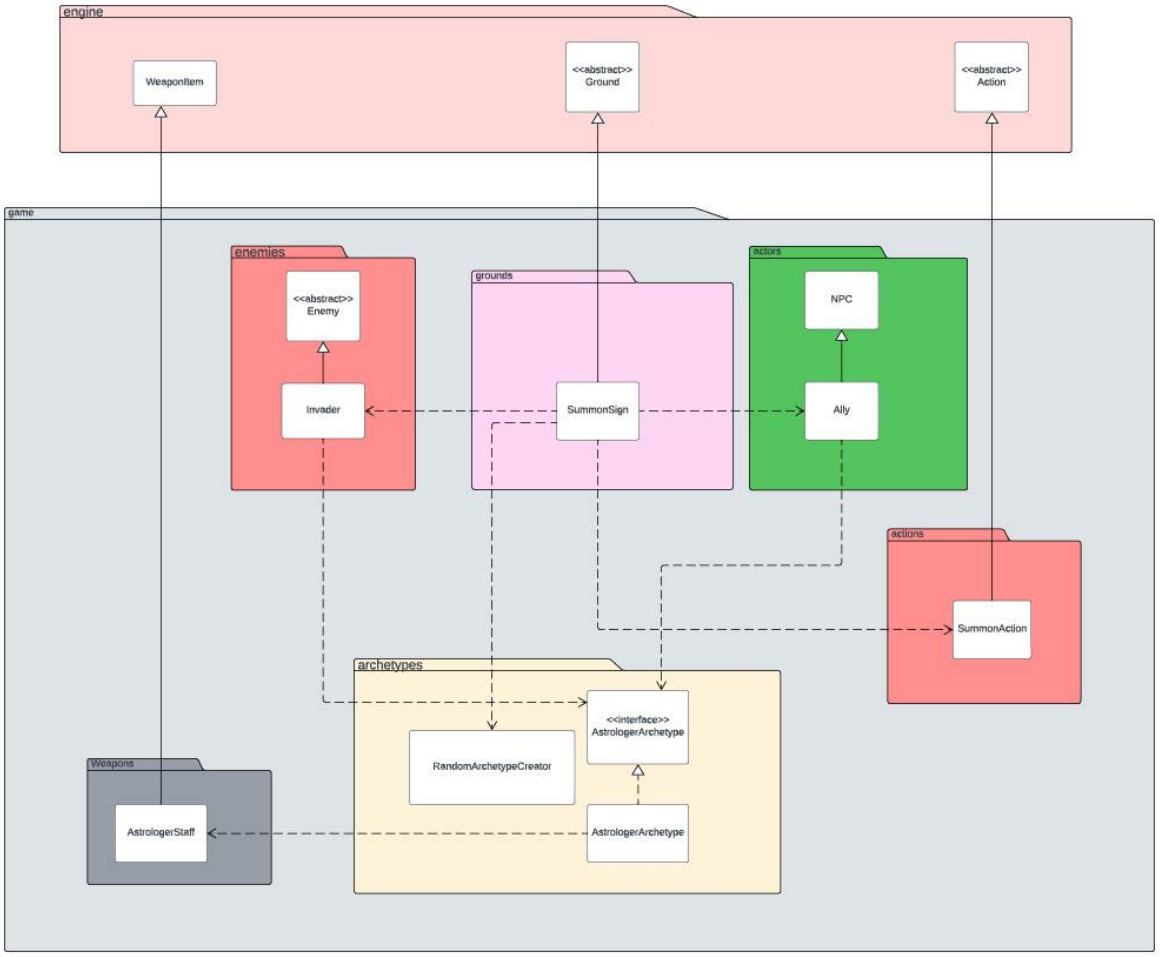
Weapons:
There are two new types of weapons items in this requirement: “AxeOfGodrick” and “GraftedDragon” are extended from weapon. Additionally, they have a dependency with

“Exchangeable” and “Sellable” interfaces. Inside the exchange method we just remove the desired currency item from the actor inventory and add the weapon being exchanged into the inventory, which shows we essentially dont ever have to modify exchange action just the exchange method that is defined in the Exchangeable item itself. It is better to separate the purchasable, sellable, exchangeable interfaces in order to follow the interface segregation principle, so classes won’t have to implement methods it does not use.

Remembrance of the Grafted and Golden Runes:
There are two new items in this requirement : Remembrance of the Grafted and Golden Runes is extended from item abstract class. Additionally, Remembrance of the Grafted implements the “sellable” interface since it can be sold for runes, which demonstrates the benefits of ISP as it won’t need to implement methods it does not use. The Remembrance of the Grafted is used for exchanging the weapons item therefore it adds the capability of ExchangeCurrency.REMEMBRANCE. This demonstrates the easy extensibility of our design as if any new item introduced can be used as an exchange currency we just create the enum and add the capability(Open-Close).

It was logical to implement the Golden runes as items since by design they can be picked up off the ground and dropped from the player. Having this extend from item allows us to make use of Item’s portability to return pickup, drop actions to the player and be stored in their inventory of items(Liskov Substitution Principle). The main aspect of the Golden Runes was the ability for it to be consumed by an actor to gain a random amount of runes. This of course was implemented through our Consumable interface and giving the Golden Runes, as a consumable the responsibility to pass the consume action to which the consume action will call the consume method to give the player runes. This is an advantage since it allows us to abstract the Golden Runes to a consumable item and make use of already existing classes in our design.

REQ 4



Astrologer archetype and weapon:
When adding the new archetype, all we have to do is create the astrologer archetype class and define the archetype interface methods getStartingHitPoints and getStartingWeapon and add it to the ArchetypeCreation class to enable user to choose it as a choice. Displays our easy extensibility when adding archetypes to the game(Open-Close).

The new weapon Astrologer Staff extends weapon item and in the tick method has the same logic as heavy crossbow, calling the RangedAttackGenerator and just using the range as input to generate the attacks necessary. Also implements the necessary purchasable and sellable interfaces and methods which follows Open-Close.

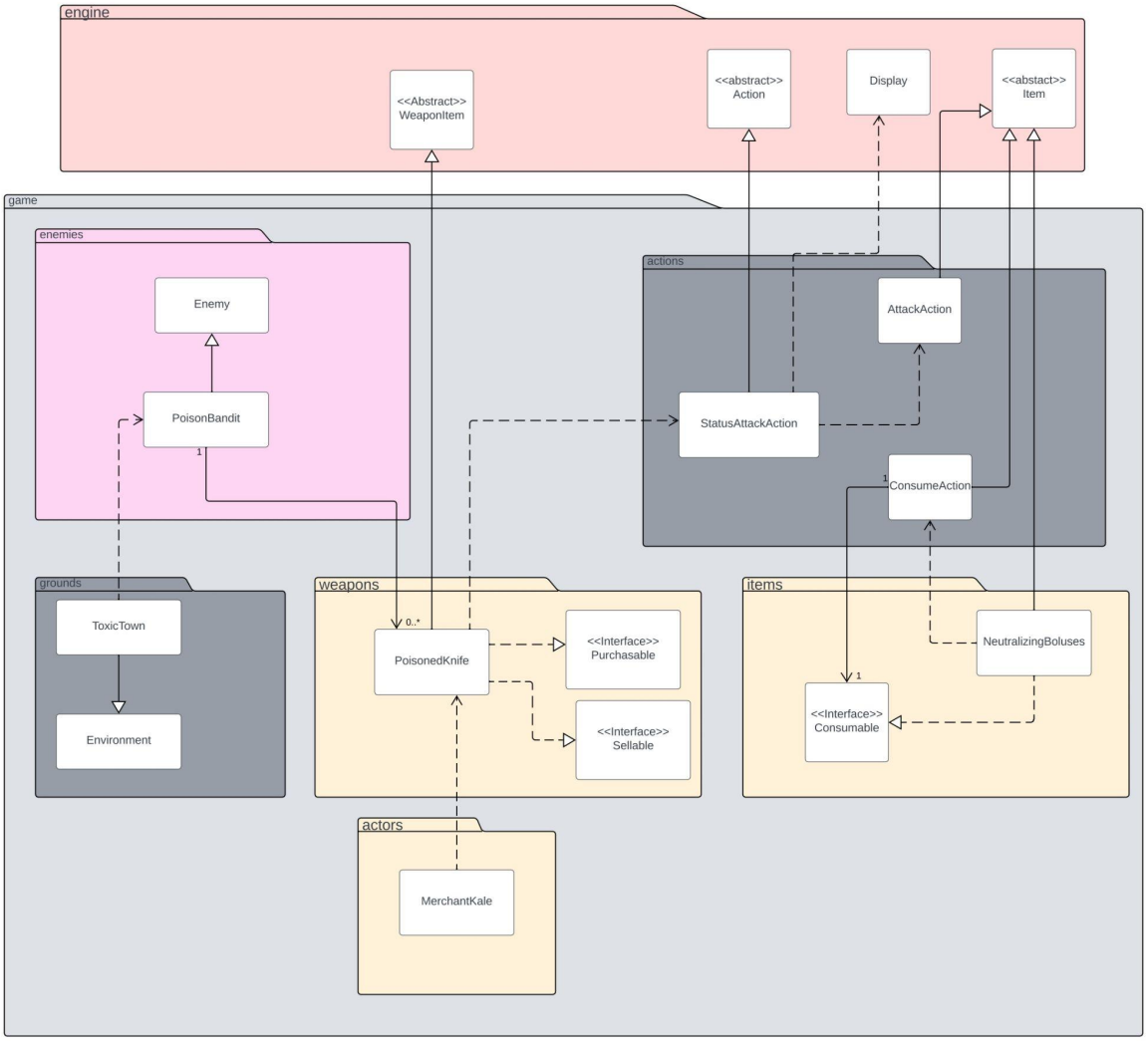
Summon Sign:
New ground that extends ground as this is a special kind of ground that only spawns and ally or invader based on an action performed by the player. We extend ground instead of environment as the spawning has to be done through interactions. We override the allowable actions to first check if the nearby actor has CAN_SUMMON capability and then calculate the 50% chance to create either an invader or ally npc, and pass that into the summon action to spawn the passed-in actor to the surrounding area. I kept the deciding which actor

to spawn logic inside the ground allowable actions itself in order to adhere to Single Responsibility principle allowing the ground to define what to spawn. This also allows summon action to be used more abstractly as what if there were other grounds that allows the player to summon different things, we can just use the same action to spawn the actors. One con with this is if the player is nearby, it will keep creating new instances each time even if you don't spawn, but I believe this is a better trade-off than having a sole summon action just for allies and invaders when executed. As what if we want to be able to have more grounds that summon or spawn things in using actions? We would have to create a new action for each one so I wanted to keep the summon action reusable which follows DRY.

Allies:
Allies are a new type of npc that helps the player kill enemies and will be given a random archetype when spawned in. So I created a new class called Random Archetype Creation that randomly creates an archetype and passes it into the ally constructor. I set the ally faction to allies so it won't attack traders and player as well. They extend from NPC instead of Enemy as it should not return attack actions in allowable actions to player. Also in the constructor we register this class into the resettableOnDeath list in reset manager that only gets called when the player dies. Again this shows easy extensibility of our design and not needing to create new dependencies or modify parent classes we extend from (Open-Close).

Invaders:
Invaders are a new type of enemy that can be spawned from summon sign. Since it has the same traits as regular enemies it makes sense to have them extend the Enemy class. We also again just pass in the archetype, set the INVADER faction and set death runes amount and thats pretty much it(Open-Close), showing the easy extensibility of our design not needing to create new dependencies.

REQ 5



Our creative requirement surrounded the new implementation of a status POISON that would damage whoever it inflicts every turn. We implemented a weapon that could inflict this status, an environment that spawns this enemy to use the weapon and a consumable to cure the poison. We demonstrated that our previous implementations of AttackAction, Consumable, Environment and Enemy classes can be extendable and follow open and closed principles. We created a new StatusAttackAction for our PoisonedKnife that returns an Attack action, this allows the StatusAttackAction to handle the inflicting of the status and retains AttackAction's role to damage the target(S). The alternative was to add another constructor into the attack action to handle the statuses but this would violate sole responsibility hence we split this up into another action. Our new poison removing consumable Neutralising Boluses is a one time use item that will remove the status from the actor consuming it. This was implemented as an item so that it can pass an action to be consumed to the player every turn, similar to the flask of crimson tears but also be portable and treated like an item(L). We can also make use of our sellable and purchasable interfaces to allow it to be sold or purchased from a merchant in the future (O).
In order to process this status effect, we implement a new capability status.POISON that is checked in the player and enemy's play turn, again adhering to sole responsibility of the

playturn method. This approach does come with the con of requiring us to constantly add new if statements in the playturn method but this is a fair tradeoff to utilise the set properties of capabilities and allow for extensibility. In the event of a playerdeath, we also have to remove this poison, this is implemented in the reset method. We still maintain sole responsibility this way since the reset method's role is to handle the events of a player's reset.

An alternative method of implementing this status would be as an item so that we can utilise the tick() method and avoid dependencies. However the main disadvantage of this method was that multiple instances of the status can be passed in and if we wanted to we have to constantly check through the inventory that the status item is already there which means it would have behaved like a Capability Set. This implementation also wouldn't work for future statuses like sleep or paralysis which would require the player to call a doNothingAction and execute it somewhere during their turn. With our method this can be handled by checking the status enum in the play turn and returning that appropriate action which still allows us to maintain sole responsibility for the player.