Design Rationale Assignment 3

MA_Lab05_Group6

Lai Khairong 33271232
Lim Yu Ean 32619561

Fixing of codes from A2

1) In the MenuManager class, we create a display object which is a class used to print messages to the user as an attribute in our MenuManager class. So that we reuse the code in the engine package without having repeated code (DRY).

2) In A2, we made a ground that can spawn enemies to implement an interface named SpawningGround which has a method to spawn enemies at a location. In A3, we change this implementation, by creating an abstract SpawningGround class, so that every ground that can spawn enemies should extend this base class. The abstract SpawningGround has an abstract spawnEnemy method, so each child class should implement this method. Therefore, we can easily add more spawning grounds into the application by extending the abstract SpawningGround (OCP), without needing to repeat the same code once again in the new class (DRY). Furthermore, in the abstract SpawningGround class, we added an attribute which is of type EnemyFactory, this attribute helps us to determine whether the ground is on the east or west side of a game map, so that we can spawn the correct type of enemy on the respective ground. EnemyFactory is an abstract class which has abstract methods to spawn enemy, we then we create two children classes which are EastMapEnemyFactory and WestMapEnemyFactory, these two classes then implements the abstracts method to spawn respective enemies on different grounds. Each of the spawning grounds has the attribute of EnemyFactory, this attribute is initialised based on the location of the ground on the map when the onTick method is called. Since EnemyFactory is an abstract class, we can't initialise the variable as EnemyFactory, instead we initialise EnemyFactory to either EastMapEnemyFactory or WestMapEnemyFactory based on the location of the map. As we wouldn't know the location of the map before the application starts running, it is logical to declare the attribute as of type EnemyFactory, and we can substitute the variable in by any of the subclass of EnemyFactory. This means that we obey the Liskov Substitution Principle and achieve polymorphism.

3) In A2, we have used downcasting to implement the logic in the allowableActions method in the enemy and player class which violates the SOLID principle. In A3, we fix this to remove all of the use of casting in the enemy and player class. Our logic of this implementation is that, instead of using casting to add the behaviour into the enemy's behaviour, we simply just add the behaviour into the enemy's behaviour when the other enemy wants to get the allowableActions in this enemy class. For example, we have a lone wolf and giant crab which are standing next to each other. So, when we process the turn of the lonewolf, we get the actions that can be done to the giant crab in the allowableActions in the giant crab class. In the allowableActions of the giant crab class, we will return the possible actions that lone wolf can do to the

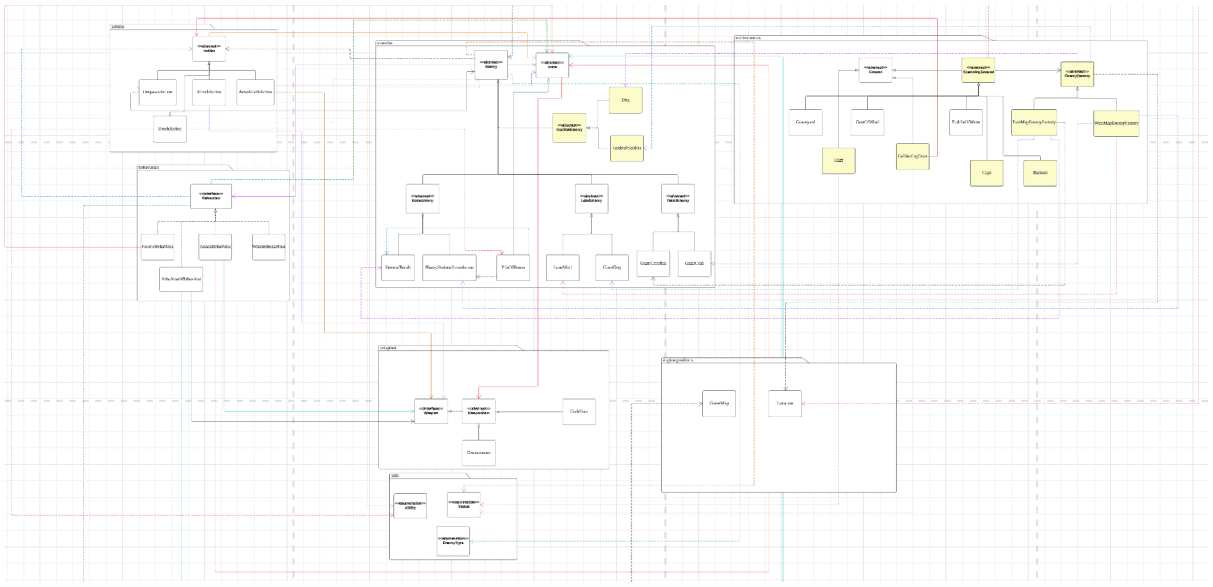
giant crab and at the same time we add in the possible behaviours of this giant crab that can do to the lonewolf in its own list of behaviours. In this case, the giant crab can attack the lonewolf when we are performing the turn of the giant crab. In short, we have changed the implementation logic in the allowableActions by eliminating the use of casting, which obeys the SOLID principles, and at the same time we can reduce the high possibility of run-time error which is caused by the use of casting.

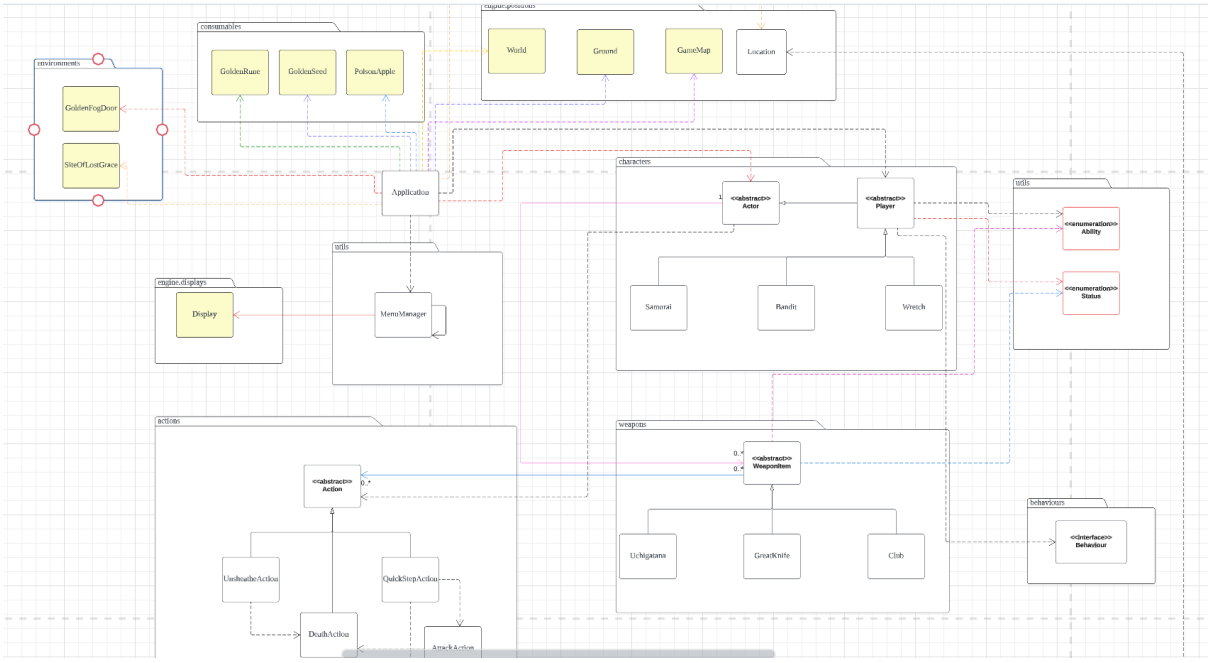
4) The logic for our runes were a bit complicated and messy for A2. There was a IProcessRune interface which only the Player implements to get and set the amount of runes the player had, a DropRetrieveRuneManager which is used to update the amount of runes the player receives when an enemy is killed. The flaw of this design is that whenever we have to update the amount of runes the player has, we have to access and modify the runes through the player. For example, when implementing the logic for purchasing and selling weapons we had to get the player instance and access its runes through it to make changes to the runes. For A3, I decided to remove both and only add a RuneManager class which handles the player's rune. The RuneManager is a singleton class which has an attribute of the player's rune. This design ensures that the player's rune can be managed anywhere easily. For example, when an enemy is killed we can easily use the RuneManager class to add the amount of runes the enemy drops and when we purchase or sell a weapon, the logic for the runes can be easily implemented using RuneManager. This design can be easily extendable (OCP) because new components of the game that require making changes to the player's rune can be easily executed by utilising the RuneManager class. Besides, it also helps to reduce code redundancy (DRY) as we don't have to get the player's instance every time we have to make changes to its runes.

5) After receiving feedback for A2 during the interviews, we were told to remove the purchase and sell managers and implement the logic inside their respective action classes. After removing the manager classes, I made the SellAction and PurchaseAction to be abstract classes. This is because if we reused the original implementation where they are concrete classes, we would need to use downcasting to remove/add the relevant items/weapons from/to our inventory. To bypass this issue, I took inspiration from the pick up and drop logic in the engine class, where I created a SellWeaponAction, SellItemAction and PurchaseWeaponAction which are concrete classes that extend from their respective abstract classes. By doing so, we can implement the removing/adding of items/weapons from/into our inventory without using any downcasting. This design achieves the Open-closed principle(OCP) as it can be easily extendable, for instance when purchasable items are added into the game we can easily implement it by creating a PurchaseItemAction class which extends the abstract PurchaseAction class.

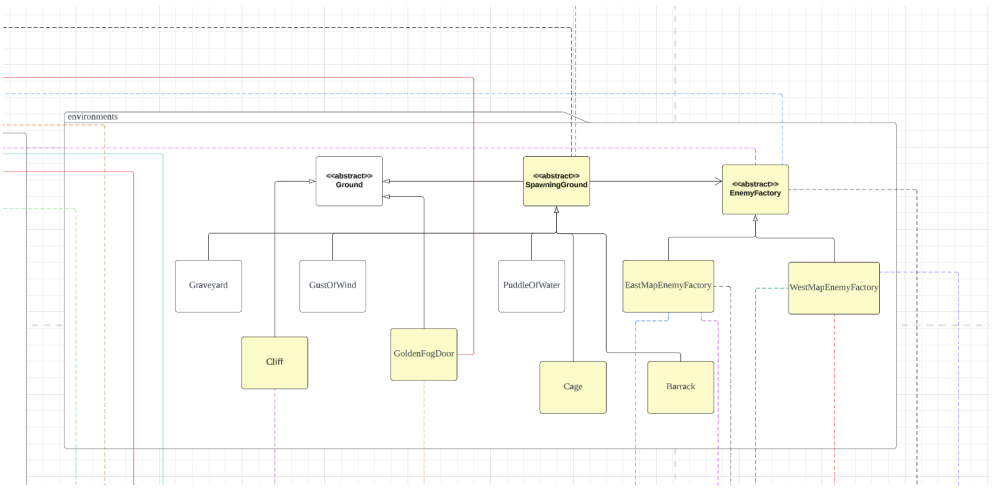
6) After knowing that there was another consumable item in A3, I decided to make changes to the logic inside the abstract ConsumableItem class. Previously, it was more tailored towards the FlaskOfCrimsonTears where the logic for number of uses left and even the health restore amount was implemented inside. I then added an abstract method - consume in the ConsumableItem class where all the consumable items' consume logic will be implemented. Then, for the ConsumeAction class which was also previously implemented in a way where it was tailored towards the FlaskOfCrimsonTears, I have changed it to have an ConsumableItem attribute and in the execute method the ConsumableItem's consume method would be called. This fulfils the Liskov Substitution Principle(LSP) where the ConsumableItem is replaceable by its subclasses. Besides, this design allows our code to achieve the Open-closed principle(OCP) where in the future if extra consumable items are added to the game it can be done easily just by extending the abstract ConsumableItem class and implementing the logic by overriding the consume method.

Requirement 1

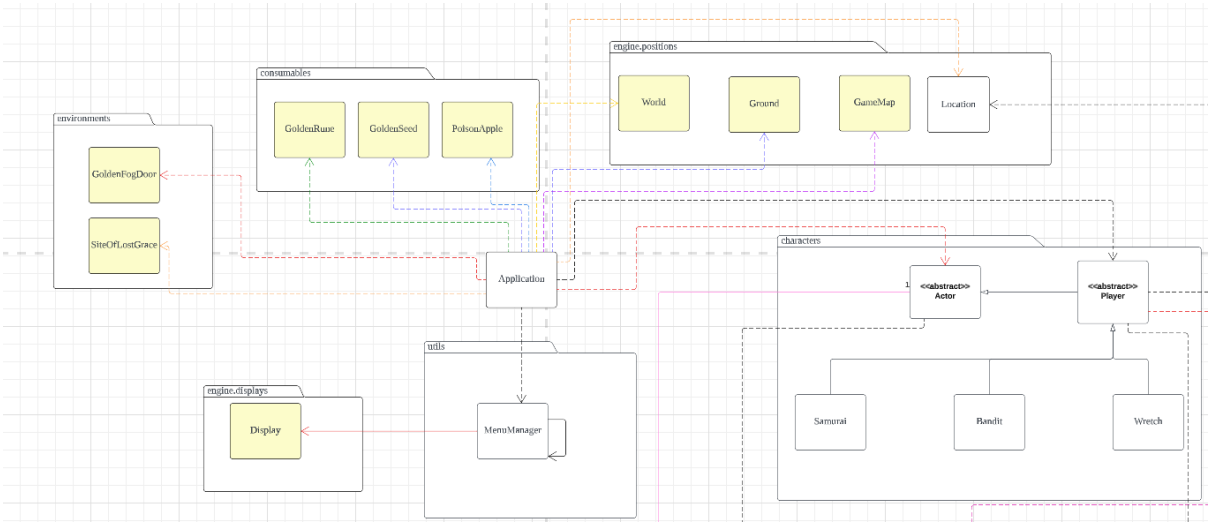




The diagrams represent an object-oriented system for Requirement 1 and 2 which consist of different environments, enemies, weapons, actions and behaviour. The design goal of Requirement 1 and 2 is to **reduce code repetition**, achieve the **SOLID principle** and as much as possible.



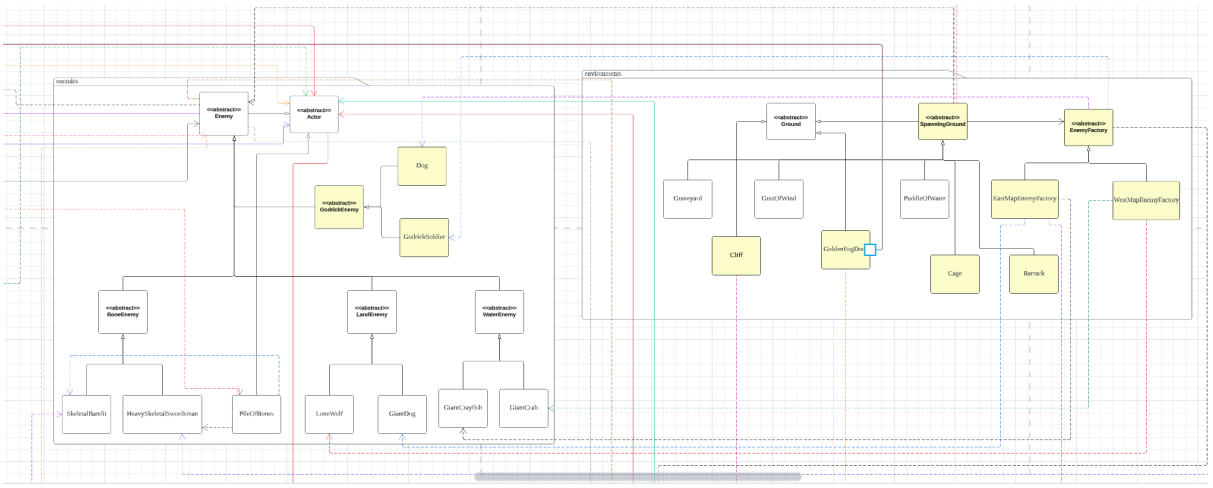
1) The two types of new ground namely Cliff and Golden Fog Door extends the abstract Ground class, since they share some common attributes and methods (OCP), it is logical to abstract these identities to avoid repetitions (DRY). Cliff and Golden Fog Door can reuse the existing method in its parent class, without the need of coding the same method (DRY) to implement their own logic. For the cliff which has the special ability, which kills the player when the player steps on the cliff, for this implementation, we create a new ability in the Status enum class to represent the special ability of the Cliff, so when the player stands on the Cliff, the player will check if the ground has a special capability, then the player will instantly get killed which means that a death action is created. For the Golden Fog Door which allows the player to travel from one map to another map, we implement in such a way that when a Golden Fog Door is created, it requires an action parameter in which this action will move the player from the current map to the other map. So, in the allowableActions in the Golden Fog Door class, we check that the 8 exits of the Golden Fog Door contain a player or not. If the surroundings of Golden Fog Door contain a player, we will add this action into the list of actions to be returned as only the player can use this special door to travel between the maps. The action parameter uses the MoveActorAction from the class in the engine package, so we just reuse the action (DRY) without creating a new action class to move the actor to another map. The disadvantage of this implementation is that the Golden Fog Door class has an association relationship with the action class which increases the dependency between classes.



2) For the implementation of several new game maps in A3, we just create a new game map object for each of the game maps and then use the addGameMap method in the World class to add in more game maps into the world. As the Limgrave map has two doors which can bring the player to the Roundtable Hold map or Stormveil Castle map, in the application class, we create each Golden Fog Door object and place the door on the particular location in the Limgrave map. This works the same for the other map which has the Golden Fog Door.

3) In the Application class, we create all the required objects in order to run the game such as the game map, player, trader, enemy and items. For each of the doors on the game map, we reuse the setGround method in the abstract Ground class (DRY) to set a particular location in the game map as the GoldenFogDoor as the GoldenFogDoor extends the abstract Ground class. By using this special door, the player can travel between maps.

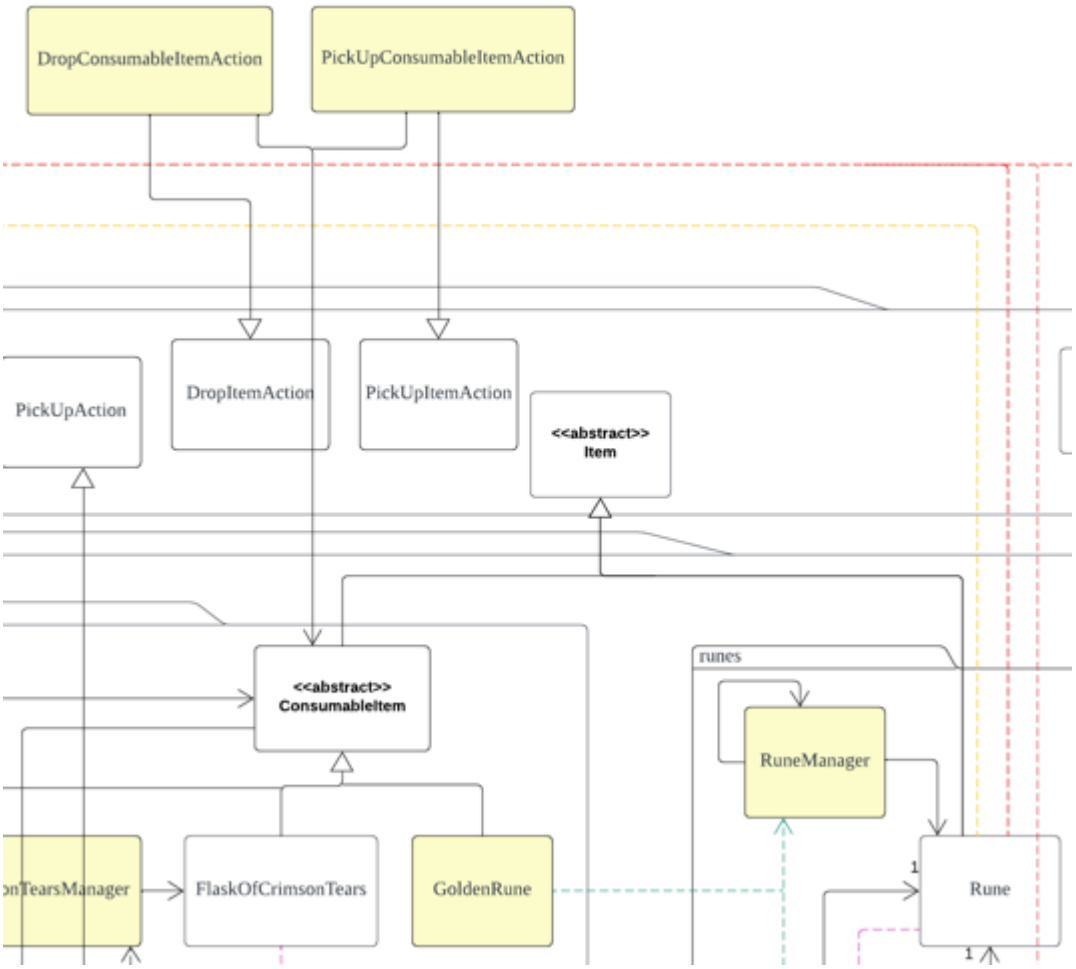
Requirement 2



1) The two types of new ground namely Cage and Barrack can spawn different enemies. Since these two grounds can spawn enemies, we will make these two classes extend the abstract SpawningGround class since they share some common attributes and methods (OCP), it is logical to abstract these identities to avoid repetitions (DRY). Therefore, any ground that can spawn enemies can be added into the system easily which fulfils the open for extension and closed for modification principle. The two new grounds appear only in the Stormveil Castle map and the enemy such as Dog and Godrick Soldier spawn from the Cage and Barrack cannot attack each other. To implement this logic, we create an abstract GodrickEnemy class and make the Dog and Godrick Soldier class extend this abstract class. Since dog and soldier cannot attack each other, we create a new enemy type in the EnemyType enum class to represent that they are of the same type (OCP).

2) Since the new weapon Heavy Crossbow requirement is optional, we reuse the Club which is the weapon of the Wretch in A2, as the weapon of the Godrick soldier, so that a club will be dropped instead of a Heavy Crossbow when the soldier is killed.

Requirement 3

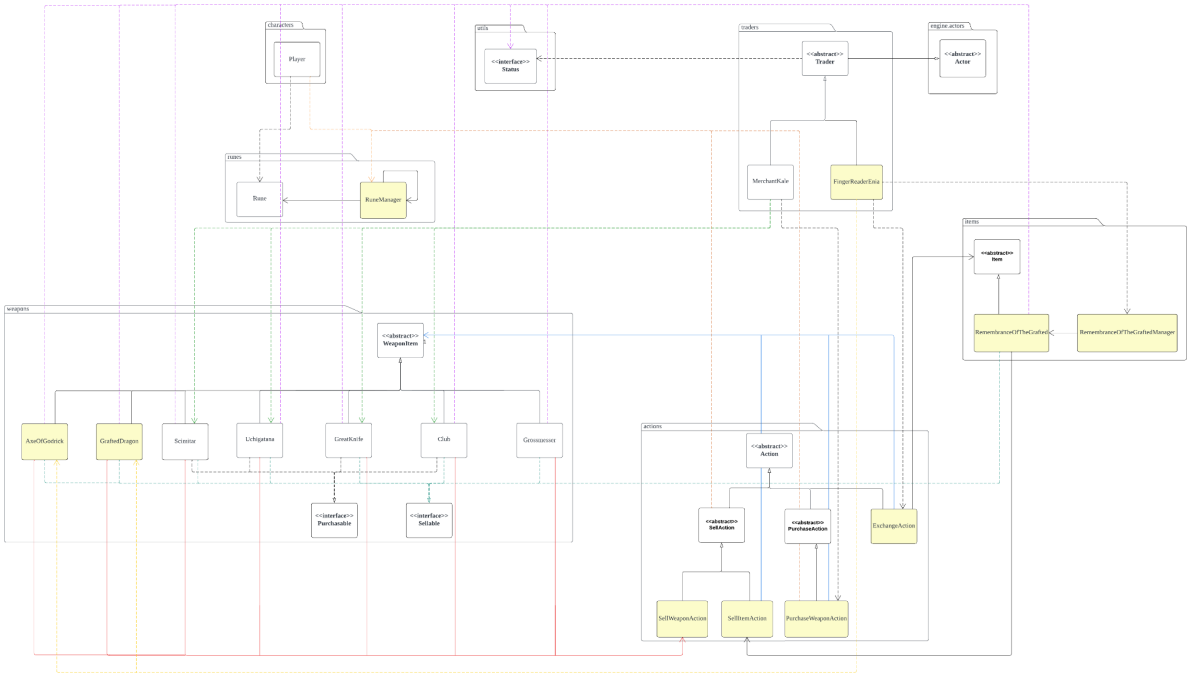


For requirement 3, we were required to implement another consumable item which is golden runes. I implemented it by creating a GoldenRune concrete class which extends the abstract ConsumbaleItem class. To generate the random amount of runes, I used the methods in the RandomNumberGenerator class and to add the runes to the player, I used the RuneManager class which is a singleton class that manages the player's rune.

To ensure the golden runes are only consumable when the player picks them up and not when it is on the ground I created a PickUpConsumableItemAction and DropConsumableItemAction class which extends their respective parent class. Inside the classes the adding and removing of ConsumeAction is implemented. I then override the getPickUpAction and getDropAction in the ConsumableItem class to return the newly created classes. This ensures the correct logic where the consumable items are only made consumable after the player picks them up and not consumable when they are on the ground.

This design fulfils the Don't repeat yourself principle(DRY) as if new consumable items are added to the game in the future, they won't have to override their getPickUpAction and getDropAction because it has already been done in the ConsumableItem class. Besides, it

also proves that our design fulfils the Open-closed principle(OCP) as a new consumable item can be easily added to the game by extending the abstract ConsumableItem class.



Because I did not implement REQ3A, I created the two weapons of Godrick the Grafted (AxeOfGodrick & GraftedDragon) by extending the WeaponItem class and the RemembranceOfTheGrafted by extending the Item class. For all of them to be sellable, they all implement the Sellable interface. The selling logic is then implemented inside their tick method where it checks whether there is a trader located in one of the eight exits of the player by checking if the Actor has the CAN_TRADE capability, if there is then only add the respective sellAction.

To implement the new trader I created a FingerReaderEnia class which extends the abstract Trader class which was implemented for the last assignment. This proves our code is easily extendable (OCP) as a new trader can be easily added to the game just by extending the Trader class.

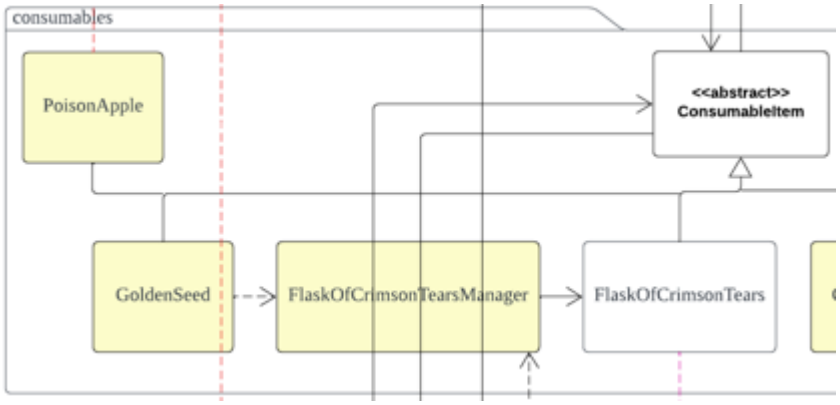
To implement the exchange logic, I created an ExchangeAction class which accepts an Item and a WeaponItem as an attribute. Then, in the execute method I just added the weapon to the actor's inventory and removed the item from the inventory. This fulfils the Liskov substitution principle(LSP) as the WeaponItem and Item can be replaced by its subclasses which allows any item or weapon to be exchanged in the future.

After watching the recording of the consultation/demonstration session, I saw that the player was only provided the options of exchanging Godrick the Grafted's weapons when they have the RemembranceOfTheGrafted in their inventory. To adhere to this logic, I made a RemembranceOfGraftedManager which has a singleton pattern for the RemembranceOfTheGrafted. I made this so that the checking of whether the player has the RemembranceOfTheGrafted in their inventory can be done inside the FingerReaderEnia

class. After checking and ensuring that the player has it, then only would the allowableActions method return the ExchangeAction.

To ensure that anything that is sellable can be sold to this trader, I just used the same logic as stated above for every sellable item/weapon where we check whether the actor located at any of the player's exit has the CAN_TRADE capability, and only add the SellAction when it's true. The capability is then added to the trader in the abstract Trader class which can reduce code repetition (DRY) as we do not need to add it in every subclass.

Requirement 5



For REQ5, I decided to add two new consumable items which are GoldenSeed and PoisonApple. After consuming GoldenSeed, the player will gain one extra use for Flask of Crimson Tears. After consuming PoisonApple, the player will be poisoned for 3 rounds and each round 50 hit points will be deducted.

Both the consumable items that I added extend from the abstract ConsumableItem class. The ConsumeAction then accepts ConsumableItem(s) to be consumed. This design fulfils the Liskov Substitution Principle (LSP) as the superclass ConsumableItem can be replaced by objects of its subclasses without breaking the application. This design can be easily extended as new consumable items can be easily added by following the same method. I also implemented new PickUp and Drop actions for the consumable items to override the original pick up and drop actions to add the ConsumeAction only when the item is picked up and remove it when the item is dropped. This design fulfils the Open-Closed Principle (OCP) as it can be easily extended when new consumable items are added to the game.

The classes from the engine package that I used are the Item, Action, PickUpItemAction and DropItemAction class. I created an abstract ConsumableItem class which extends the Item class for all consumable items to extend from. I used the Action class for my ConsumeAction which implements the logic for consuming consumable items. Lastly, I used the PickUpItemAction and DropItemAction to create new pick up and drop actions to override the original pick up and drop logic to add the ConsumeAction when the consumable item is picked up and remove it when it is dropped.

I used the Flask of Crimson Tears from Assignment 2. I used it for my new consumable item - GoldenSeed which will increase the number of uses of it after it is consumed. To implement this I used a similar method for the RemembranceOfTheGrafted where I created a FlaskOfCrimsonTearManager which has a singleton pattern for the FlaskOfCrimsonTear. This enables the adding of the uses of the player's FlaskOfCrimsonTear to be done in the GoldenSeed class.

The abstraction that I used is the abstract ConsumableItem class. I used it for all my consumable items including the new consumable items for requirement 5.

I created a new capability - POISONED and added it to the Status enum. This capability is used to implement the logic when the player is poisoned after consuming the PoisonApple. I implemented the logic inside the player's playTurn method which checks if the player has the POISONED capability and deducts fifty hit points if true. This will last for three turns and after three turns the capability will be removed from the player. If the player were to die in any of those turns, a DeathAction would be returned.