FIT2099 Object-oriented Design and Implementation
Assignment 2 - UML Diagrams, Interaction Diagrams & Design rationale

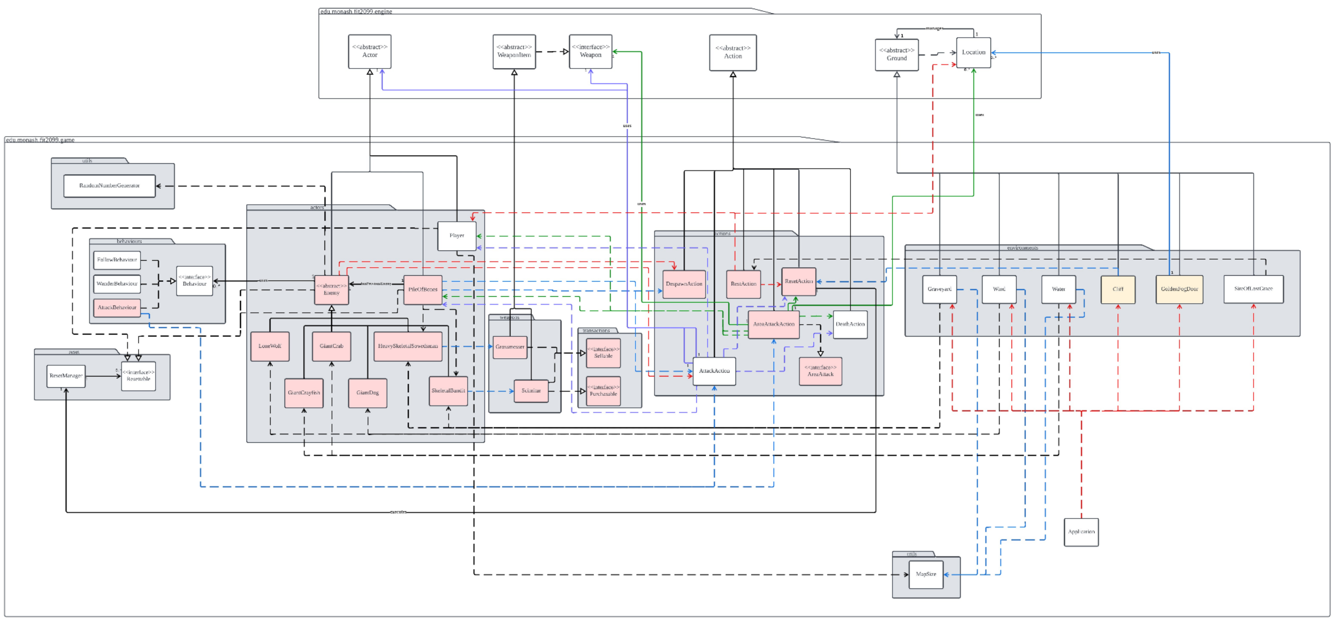
Group members:
1. Muhammad Mustafa Khan (33041202)
2. Nurin Damia Ahmad Azhar (32903510)
3. Iliyana Samsudin (32738781)

Link to Contribution log:
https://docs.google.com/spreadsheets/d/1e7polQC9IYCEiwOh9_jiUQ6UL1qxGAKn3CofYyGJVig/edit#gid=0

Note: Yellow classes are the newly created classes for assignment 3.

Requirement 1 - Travelling between Maps

UML Diagram



Design Rationale

The UML diagram above represents the object-oriented system of the newly added environments, Cliff and GoldenFogDoor.

The inclusion of the Cliff and GoldenFogDoor environments as concrete classes, which inherit from the Ground abstract class, effectively captures their shared attributes and methods that are inherent to a type of ground. Utilising Inheritance in this manner helps avoid code duplication and aligns with the DRY (Don't Repeat Yourself) principle.

In the case of the Cliff class, it overrides the canActorEnter method to restrict access only to actors with the capability of Status.PLAYER. Additionally, the tick method is overridden to inflict damage on the player using the hurt method, resulting in the player's death and game reset. By customising the behaviour of the Cliff class and leveraging inheritance, we ensure that specific ground types have their unique functionality while avoiding code redundancy.

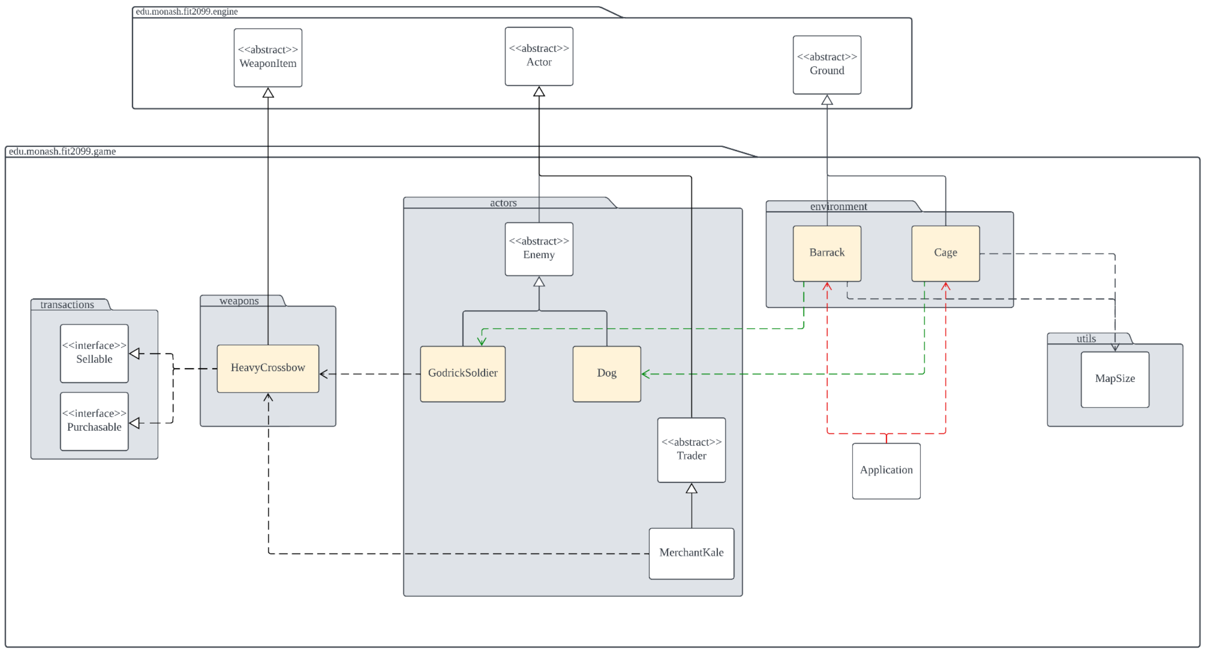
For the GoldenFogDoor class, it utilises a hashmap to store the player's location after they pass through the door. Similar to Cliff class, the canActorEnter method is overridden to limit access exclusively to actors with the capability of Status.PLAYER. Additionally, the allowableActions method is customised to grant the player the ability to perform actions, such as MoveActorAction. By employing a hashmap to track the player's location and overriding relevant methods, the Golden Fog Door class allows us to cater to its unique needs.

For requirement 1, there is no disadvantage as the design allows for easy extension of new environments by extending the Ground abstract class which also follows the DRY principle.

To extend the design in the future, new environments can be added by extending the Ground abstract class. This design approach enables each subclass to possess its own unique attributes and methods while inheriting the common functionality defined in the abstract class.

Requirement 2 - Inhabitant of the Stormveil Castle

UML Diagram



Design Rationale

The UML diagram above represents the object-oriented system of the newly added environment and enemies in the game.

The addition of the Barrack and Cage environments as concrete classes,extending from the abstract class Ground, effectively captures their shared attributes and methods that are inherent to a type of ground. By using inheritance, we prevent code duplication and adhere to the DRY (Don't Repeat Yourself) principle.

Moreover, Barrack and Cage possess unique characteristics, each of the environments spawns distinct enemies and different probability to spawn the enemies. Through subclassing, we can accommodate the specific traits of each environment while maintaining a streamlined codebase.

However, one drawback emerges from the current implementation. Both of these environments are required to implement a static method from the MapSize class. This results in repetitive implementation of the same static method which causes us to follow the WET principle (Wasting Everyone's Time).

For the enemies, Enemy class is an abstract class and is extended from the Actor abstract class which provides all the functionalities required by an Actor to create a NPC (non-playable character). Then, GodrickSoldier and Dog are concrete classes that are extended from the Enemy class. Although there is a multiple tier of inheritance, each of the enemies share common attributes such as the AttackBehaviour, WanderBehaviour and FollowBehaviour which are implemented in the Enemy abstract class. The despawning of enemies is also done in the Enemy class by returning a new DespawnAction. By extending from the Enemy class, it allows us to reduce code repetition which complies with the DRY principle (Don't Repeat Yourself). Any new additional attributes and behaviours can be implemented in its respective subclasses to cater each class uniqueness.

The AttackBehaviour class that implements the behaviour interface is used for the enemies. This class is used to figure out an attack action that will be invoked by the enemy. The downside of this is that whenever a new type of area attack action is added for another enemy, the execute() method will be extended and modified, violating the Open-Closed principle.

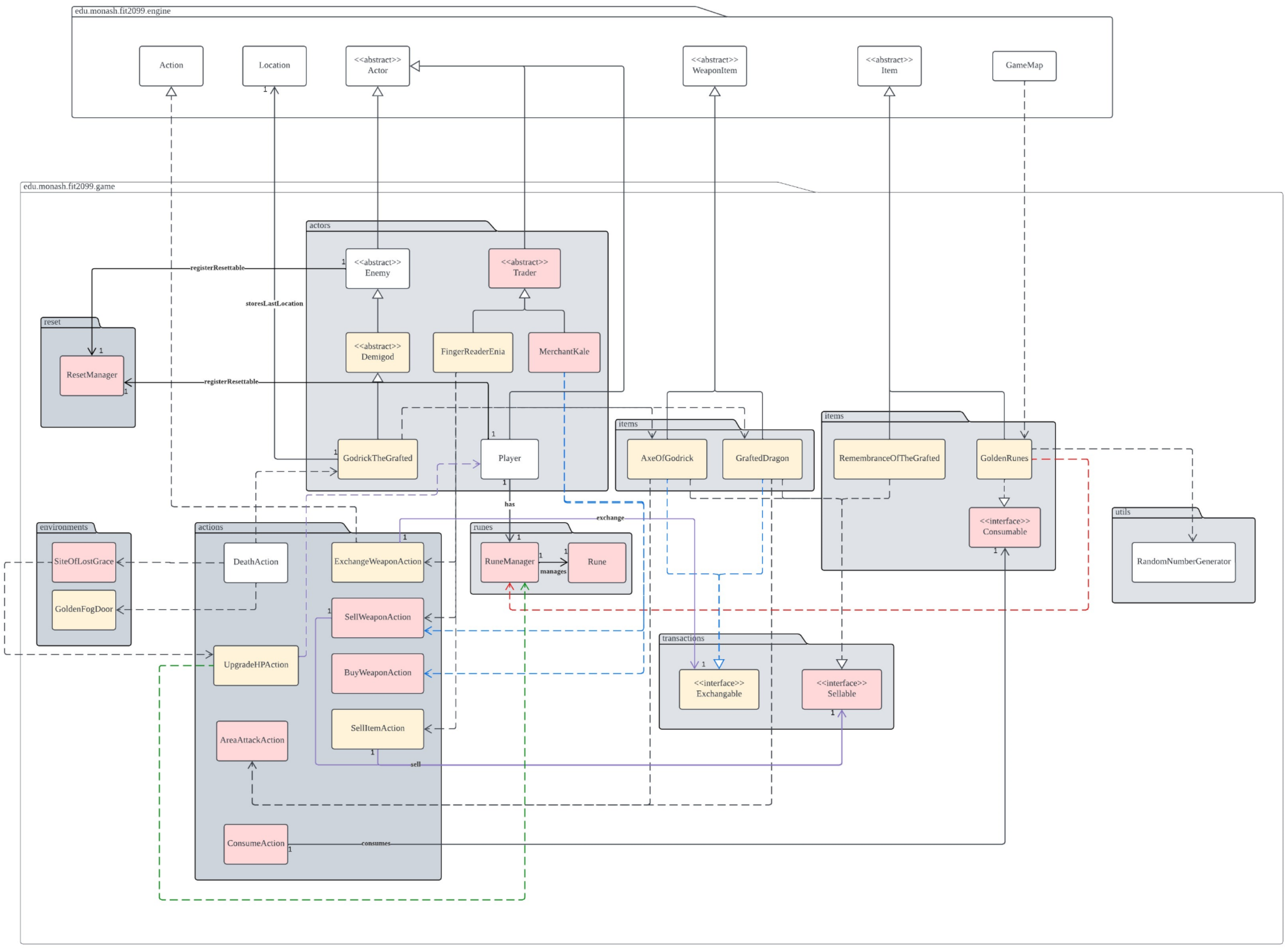
Furthermore, a new status is created for Dog and GodrickSoldier class which is used to be added as a capability in the constructor. This allows the dog and the godrick soldier to attack other enemies such as lone wolf or giant dog but not each other.

Lastly, the HeavyCrossbow is a weapon that is used by the Godrick Soldier. It is extended from the WeaponItem class which extends from the Item abstract class and implements the Weapon interface. This allows new Weapon types to be created without having to modify the existing code. By extending the WeaponItem class, Heavy Crossbow shares common attributes and methods that are implemented in the WeaponItem abstract class and the Item abstract class which can be reused. This allows us to follow the DRY (Don't Repeat Yourself) principle. The Heavy Crossbow class also implements the Sellable and Purchasable interface which allows this weapon to be bought from or sold to the Merchant Kale. This shows that we have followed the ISP principle (Interface Segregation Principle). However, we are required to modify the merchant kale class everytime a new weapon can be bought from the class. This causes our implementation to violate the Open-Closed principle. Note that the heavy crossbow is implemented as a normal weapon (ranged attack is not implemented).

To extend the design in the future, a new environment can be created by extending the Ground class and implementing the static method from the MapSize class. Secondly, new enemies can be added by extending the Enemy class and modifying the attack behaviour class if the enemy has a unique attack action such as area attack action. Lastly, a new weapon can be added by extending the WeaponItem class and implementing Purchasable or Sellable interface if they can be bought or sold.

Requirement 3 - Godrick The Grafted

UML Diagram



Design Rationale

The UML diagram above represents the object-oriented system of Godrick the Grafted, which is a new boss in the game.

A new abstract Demigod class is created, extending the Enemy abstract class. This is because a Demigod also behaves like an Enemy, sharing common functionalities for Enemy NPCs. By extending the abstract Enemy class, the Demigod class inherits common attributes and methods, promoting code reuse and modularity. GodrickTheGrafted is the first boss introduced to this game, extending the Demigod abstract class. Polymorphism is demonstrated as Godrick the grafted overrides specific methods to implement unique boss behaviours for its two phases, utilising the flexibility provided by the common interface. This design adheres to the OCP principle. The Demigod abstract class will serve as a foundation for creating additional boss types in the future, each with their own unique characteristics and

mechanics. Furthermore, the design facilitates code organisation by encapsulating boss-specific behaviour within the Demigod and GodrickTheGrafted classes.
However, there is a potential drawback of this implementation, where we assume that the GodrickTheGrafted class will only have two boss phases, which limit the addition of other phases in the future. Another drawback is that for the Demigod class, there is a multi-level inheritance, from Actor -> Enemy -> Demigod, which may increase code complexity, so it requires careful consideration for maintainability.

Furthermore, the Golden Rune class extends from the Item abstract class and implements the Consumable interface. When a Golden Rune is consumed, a random amount of runes within a specific range will be given to the Player. The Golden Rune item shows the Dependency Inversion Principle (DIP). In the case of the Golden Rune, it depends on the abstract concept of the Consumable interface rather than specific implementation, promoting loose coupling between the Golden Rune class and the concrete classes implementing the Consumable interface. It allows for flexibility and interchangeable usage of different consumable items within the system.

The Trader class from the previous implementation was modified. In the new implementation, Trader class is an abstract class which extends from the Actor abstract class. For now, it has two subclasses, Merchant Kale which was implemented previously and a new trader, Finger Reader Enia with distinct functionalities and behaviours. The concrete trader classes (e.g., Merchant Kale and Finger Reader Enia) can be used in place of the abstract Trader class. This ensures that any code written to interact with traders through the Trader interface can seamlessly work with different trader types without needing to modify the code, adhering to LSP.

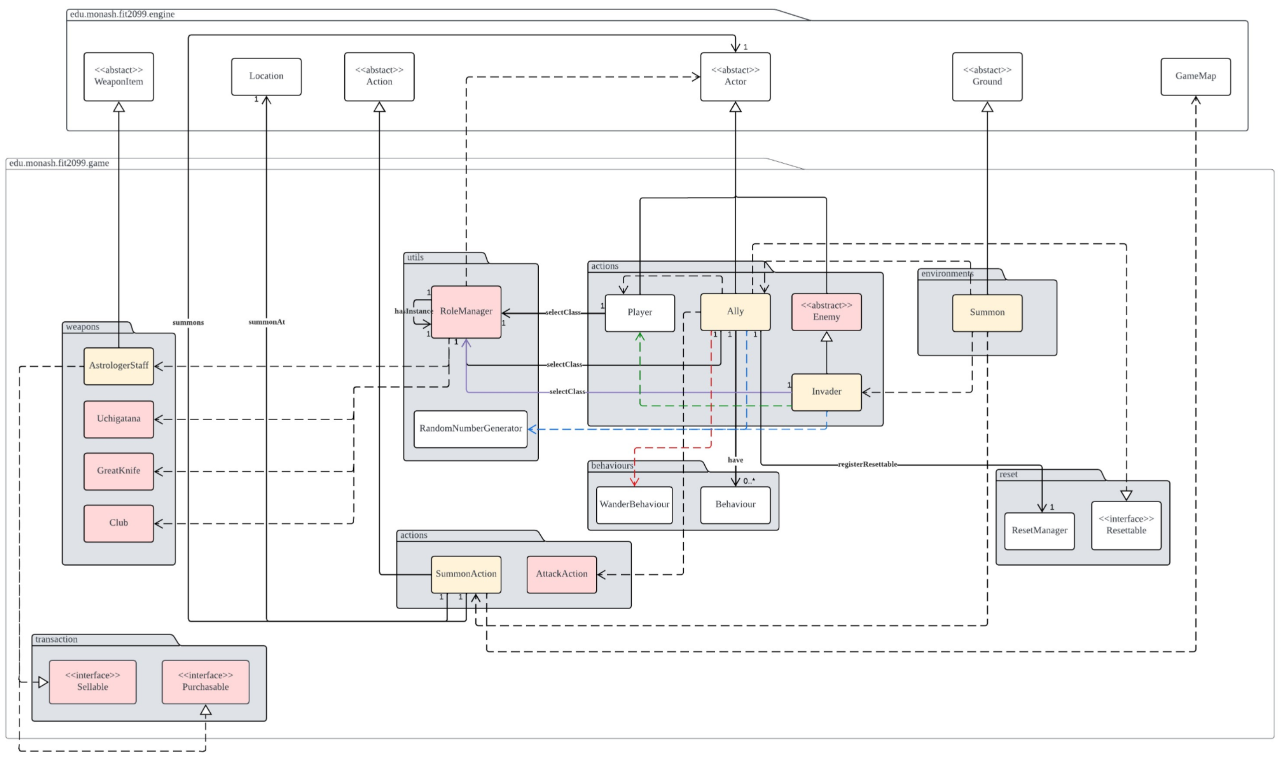
Finger Reader Enia is responsible for exchanging Remembrance of The Grafted for one of Godrick's weapons, and it can also be sold for 20000 runes. This is done by introducing two new Action classes, SellItemAction class and ExchangeItemAction class. These new Action classes extend the existing functionality without requiring modifications to the existing code. This design aligns with the OCP principle as it allows the incorporation of new item-related actions in the future, enhancing modularity and code reusability by encapsulating the behaviour of buying and exchanging items within dedicated classes.

The pros of this design is the use of inheritance and interfaces promotes code reuse and modularity, allowing for the creation of specialised items like the Golden Rune. The abstraction of the Trader class provides a foundation for multiple traders with unique behaviours, such as Finger Reader Enia. The cons are the introduction of multiple traders may increase code complexity, as managing the behaviours and interactions of different traders can be more challenging. The tight coupling between the concrete trader classes and their actions may require modifications in multiple places if changes are made to the abstract Trader class.

To extend this design in the future, new bosses can be created just by extending the Demigod class. Each boss can introduce their own behaviours and probably drop items in their respective classes, where the drop items can be created by extending the Item abstract class. New traders can also be created by extending the abstract Trader class, with their own unique behaviours.

Requirement 4 - A Guest from Another Realm

UML Diagram



Design Rationale

The UML diagram above represents the object-oriented system of Guests from Another Realm, which are Allies and Invaders.

Allies and Invaders are the new classes which extend from Actor and Enemy abstract classes respectively. These two actors will have their roles randomly generated using the RoleManager and RandomNumberGenerator from the utils package. Allies are non-hostile creatures that inherit all the common attributes and methods that are required for actors with their own unique behaviours that allows them to assist players in the game. On the other hand, Invader classes inherit all the common attributes and methods of the Enemy class with their own stated behaviours as well. Therefore, both of these two NPC classes showcase the principles of inheritance and polymorphism.

The design of these two classes adheres to the SRP, because both of the classes have a clear and focused responsibility, showcasing an advantage to increasing understandability and also maintainability of the code. The disadvantage of this implementation is that this design may increase the complexity of the system with the introduction of additional classes that extend to the existing classes with multiple-inheritance (e.g. Enemy abstract class). Careful management of interactions and dependencies is required between the Invader and Enemy classes.

To summon these two guests, players will have to stand on the Summon sign and interact with it. Hence, we have created a Summon class which extends the Ground abstract class and a SummonAction class which extends the Action abstract class. This implementation adheres to the SRP because the Summon class has only one responsibility, which is to handle the summoning of Allies and Invaders. The SummonAction class on the other hand adheres to OCP because it allows the extension for a new summon action by creating a subclass of the Action class, SummonAction, without having to modify the existing code.

The pros of this implementation is the code reusability. Using abstract classes and extensions allow the reusing of existing code such as the Ground and Action abstract classes. The two newly implemented classes extend the two classes with their own unique characteristics without modifying any of the existing classes. On the downside, this increases complexity due to the introduction of new classes, which requires a lot of consideration so that the added classes do not decrease the understandability of the codebase.

The addition of a new player class, Astrologer, is done in the RoleManager class by adding a new option to the menuItem class (*refer to line 48 of RoleManager class*). Previously, the choices of the player were done in the method created in the Player class. But for the new implementation, that method is moved to the RoleManager class and renamed to selectedClass method, where it takes an actor and an integer which is the player selection in the parameters. The reason for this modification is that now Allies and Invaders will also have these player classes chosen at the start of their creation. The new modified Role Manager class now follows the OCP principle, so new classes can be added to the menuItem class by extending the method without altering the existing codes.

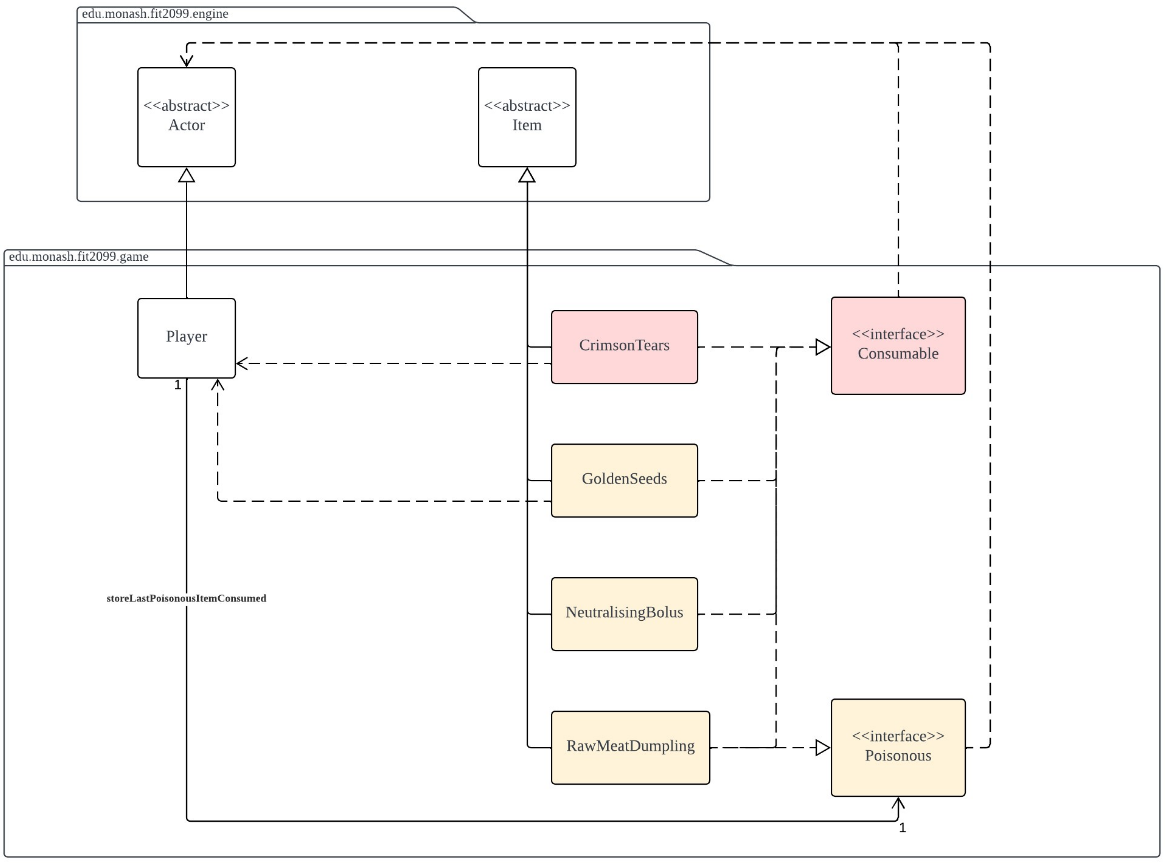
The Astrologer class will start with 396 HP and a new starting weapon Astrologer staff. Note that the astrologer staff only acts like a normal weapon (ranged attack not implemented). The Astrologer staff extends the WeaponItem abstract class with its

own characteristics. The Astrologer staff class encapsulates their own characteristics and behaviours specific to the astrologer staff, adhering to the SRP principle.
The pros to this design is that it promotes extensibility and flexibility, where the code for menuItem can be extended when new player classes are added into the game. The cons to this is that there is limited functionality to the Astrologer staff, since the RangedAttackAction was not implemented for this weapon, so it does not fully showcase the expected skills of the weapon.

How this design can be extended in the future is that if there are new types of Allies or Invaders in the future, they can extend to these two classes with their own unique behaviours and attributes, such as their own specialised skill, or new co-op strategies for new Allies. But this extension must be managed and extended carefully due to the deep-layer inheritance. For new types of summoning actions, they can just extend the current SummonAction class with their own functionalities. As for the skills for ranged weapons, the skills can be implemented by creating a new action class, RangedAttackAction class so that the weapons will be fully utilised to its real capabilities.

Requirement 5 - Creative Requirement

UML Diagram



Design Rationale

***Note**: The newly added Consumable items, Golden Seeds, Raw Meat Dumpling and Neutralising Bolus are added to the Player's inventory at the start of the game for testing purposes.*

The design rationale for our creative requirement below is a copy of the Google form submitted for the creative requirement.
(refer to: https://edstem.org/au/courses/10331/discussion/1384755)

1. Title for requirement: New consumable items with varying effects

2. Description of requirement:

Implement new consumable items:

1. *Golden seeds*

   Golden seeds will extend Item (from Engine package) and implement the Consumable interface (existing interface in Game package). When golden seeds are consumed, the number of uses of the Crimson Tears will be increased. For example, the current uses left for Crimson Tears is 1, and when the player consumes a Golden seed, the number of uses for Crimson Tears will be increased by 1, so the current number of uses left will be 2.

2. Neutralising boluses

   Neutralising boluses will extend Item (from Engine package) and implement the Consumable interface (existing interface in Game package). When consumed, it instantly alleviates poison buildup and cures poison.

3. Raw meat dumpling

   Raw meat dumpling will extend Item (from Engine package) and implement the Consumable interface (existing interface in Game package). When consumed, it will restore the player's HP but at the same time will also cause Poison buildup (inflict poison status which causes player's HP to be drained for 3 rounds).

3. Why our creative requirement adheres to SOLID principles

1. The implementation of new consumable items will adhere to:

   Single Responsibility Principle - GoldenSeeds have only one responsibility, which is to increase the number of uses for crimson flasks, and will not have other unrelated functionalities in the class. The number of uses of Crimson

Flask is delegated to the Player class, which indicates compliance with SRP as well. (Applies to the other consumable items)

Open-closed Principle - The implementation of GoldenSeeds can be achieved just by extending the existing code without any modifications. (Applies to the other consumable items)

Liskov Substitution Principle - The GoldenSeeds class extends the Item class and implements the Consumable interface, indicating that it can be substituted for other consumable items without affecting the correctness of the program. (Applies to the other consumable items)

Interface Segregation Principle - The Consumable interface only contains the consumeItem method that is common to all consumable items, and the GoldenSeeds class does not force dependencies on additional methods. (Applies to the other consumable items)

Dependency Inversion Principle - The Player class will have a dependency on the Consumable interface, and not the consumable items such as CrimsonFlask or GoldenSeeds directly. It can be seen that in the playTurn method:

```
for (Item item: this.getItemInventory()){
   if (item.hasCapability(Status.CONSUMABLE)){
   actions.add(new ConsumeAction(item));
    }
}
```

This allows for loose coupling and facilitates easier changes, such as adding new consumable items without modifying the Player class. (Applies to the other consumable items)

4. Must use at least 2 classes from the engine package. Which classes will we be using and how will we use it?

- Item abstract class. Example, GoldenSeeds extends Item
- Action abstract class. Example, ConsumeAction extends Action

5. Re-use at least one existing feature from A2 for this requirement

- The ConsumeAction class and Consumable interface was implemented in A2 for the consumption of Crimson Flask, which will be reused for the implementation of this requirement.

6.Must use existing or create new abstractions. Explain what will be re-used or created and how will it be used?

- We will be re-using the Consumable interface. The consumable interface has one method, consumeItem which will contain the logic of the consumption of the item (e.g. Increase uses for crimson flask of a player)

7. Create new or use existing capabilities and explain how it will be used
We have a Consumable status, which indicates the item is consumable. In the Player's playTurn method, we check if the player has any consumable items by using "if (item.hasCapability(Status.CONSUMABLE))" before adding a new ConsumeAction to the actions list. We will also be creating a new status which indicates POISON being inflicted to players. So while the player have this POISON status, damage will be inflicted to the player.