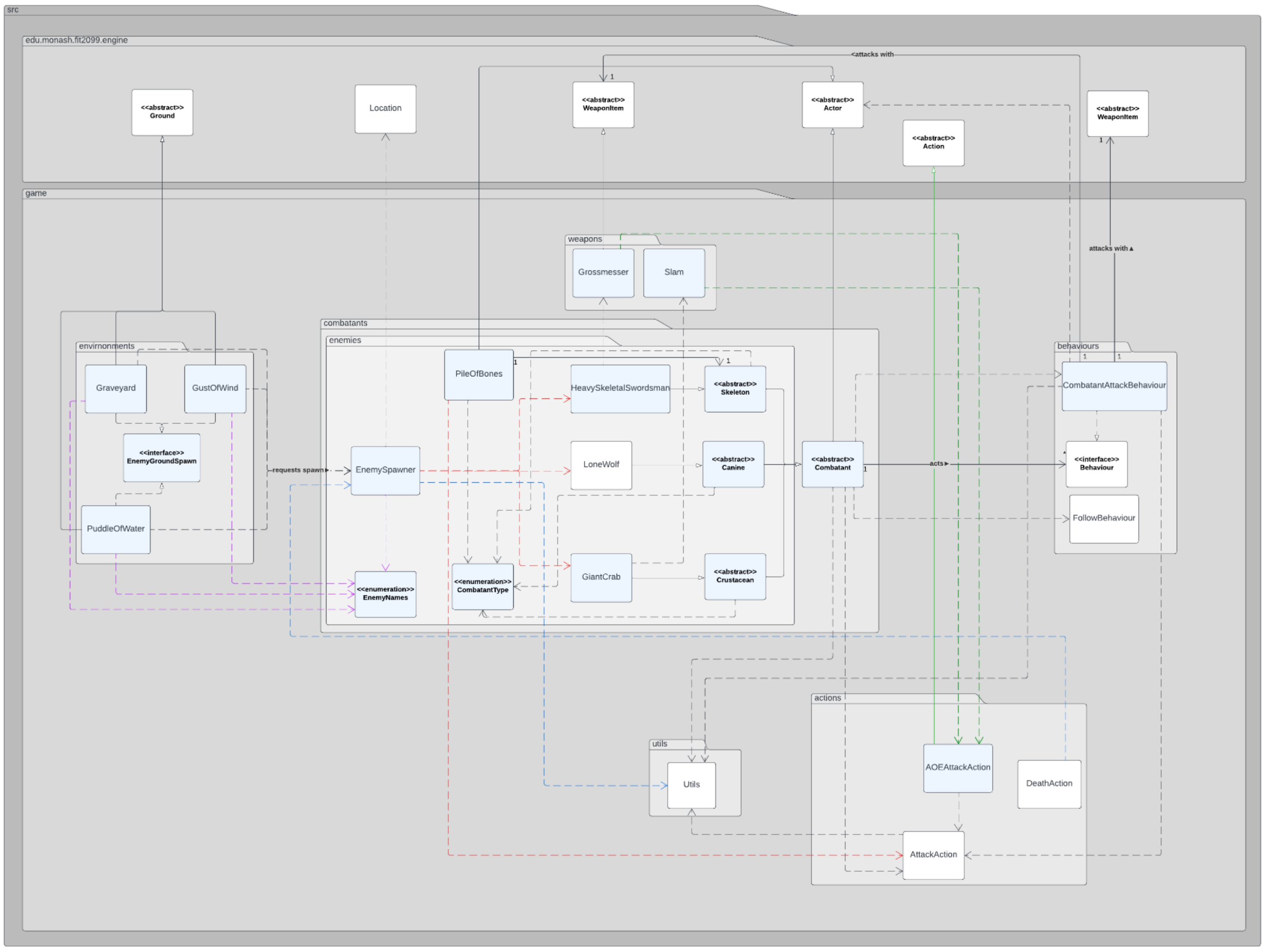
FIT2099 Assignment 3 Master Document

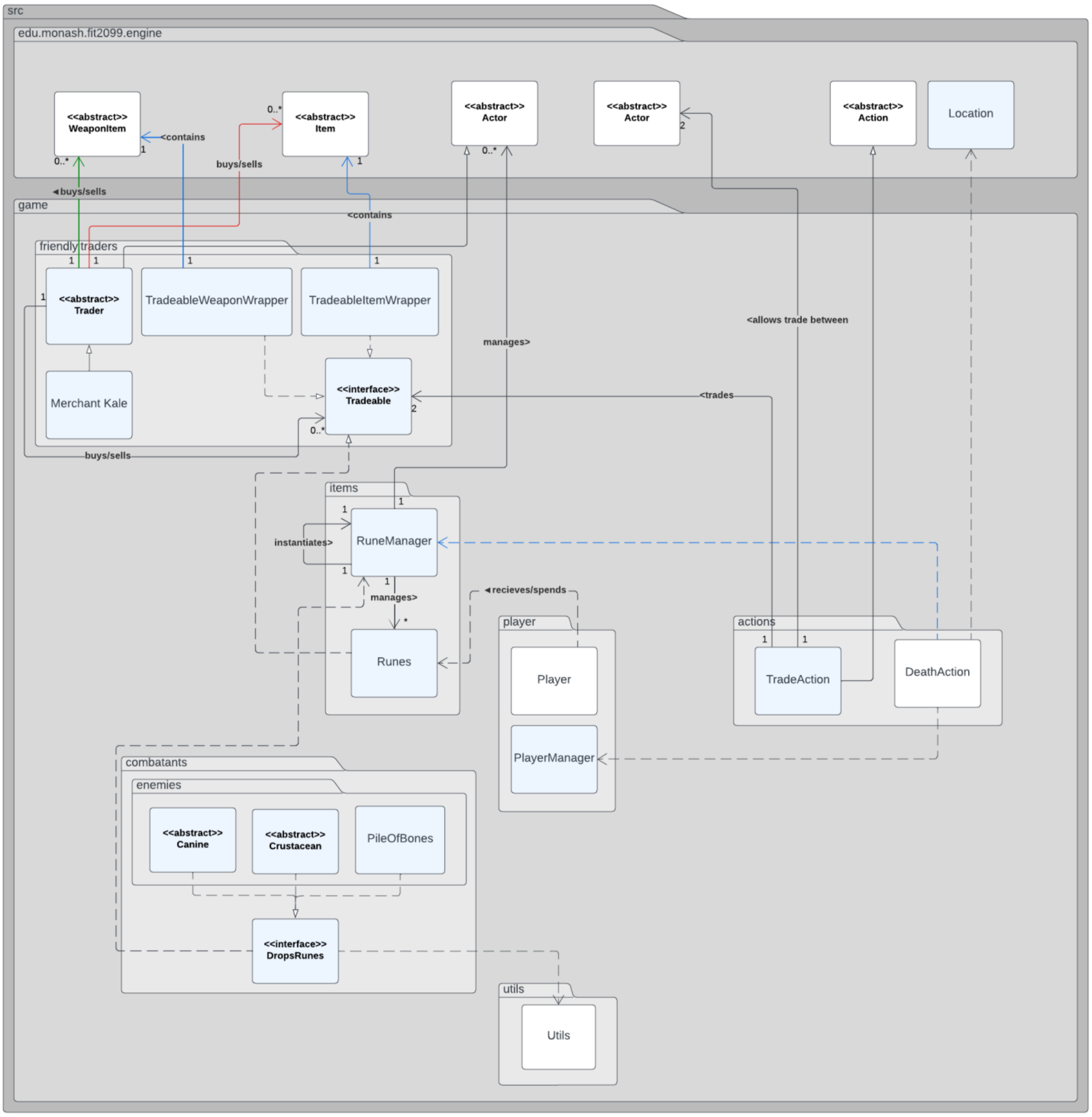
Applied Session 3, Group 8

UML Changes from Assignment 2

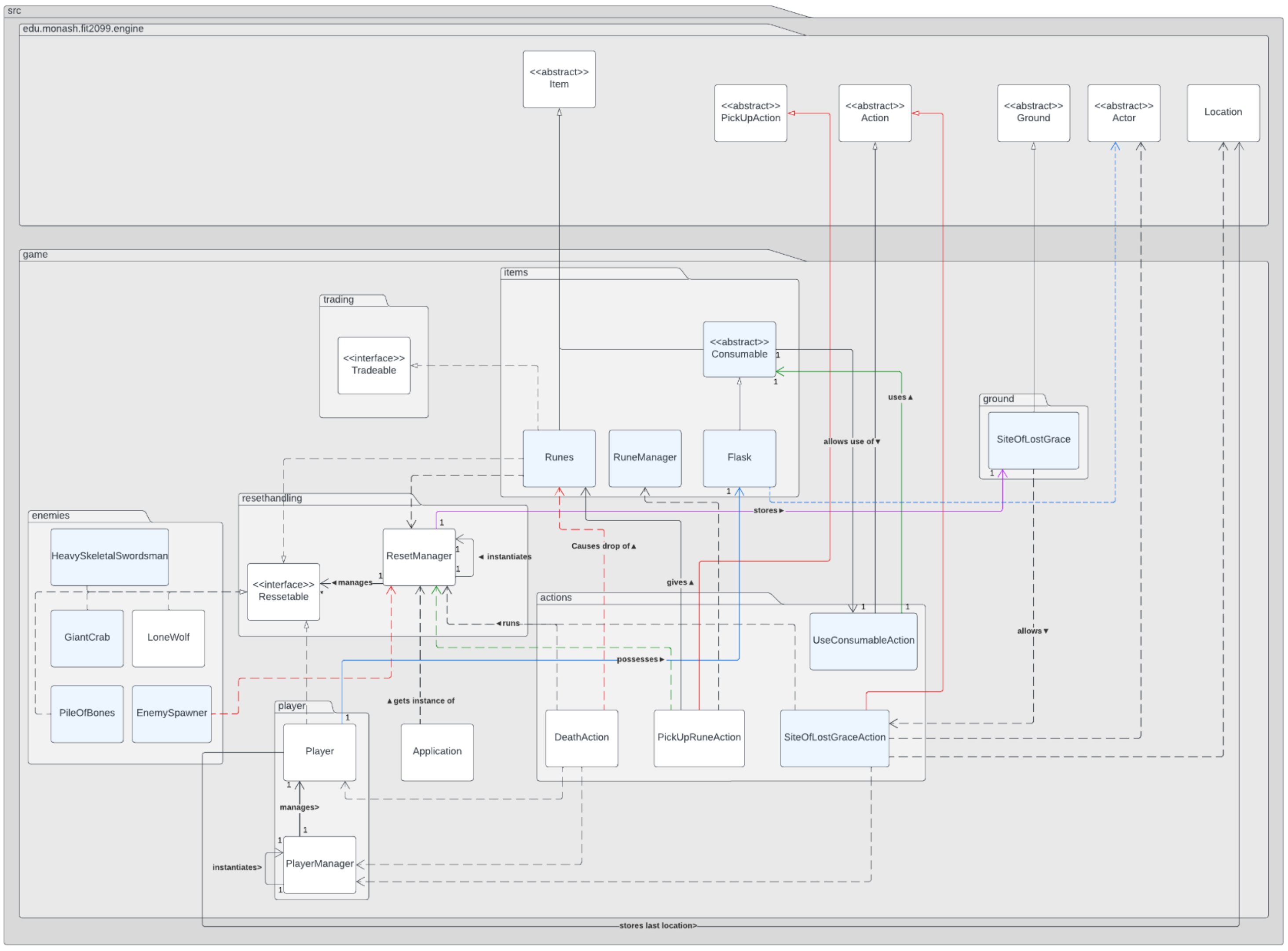
<u>Req 1 UML Diagram:</u>



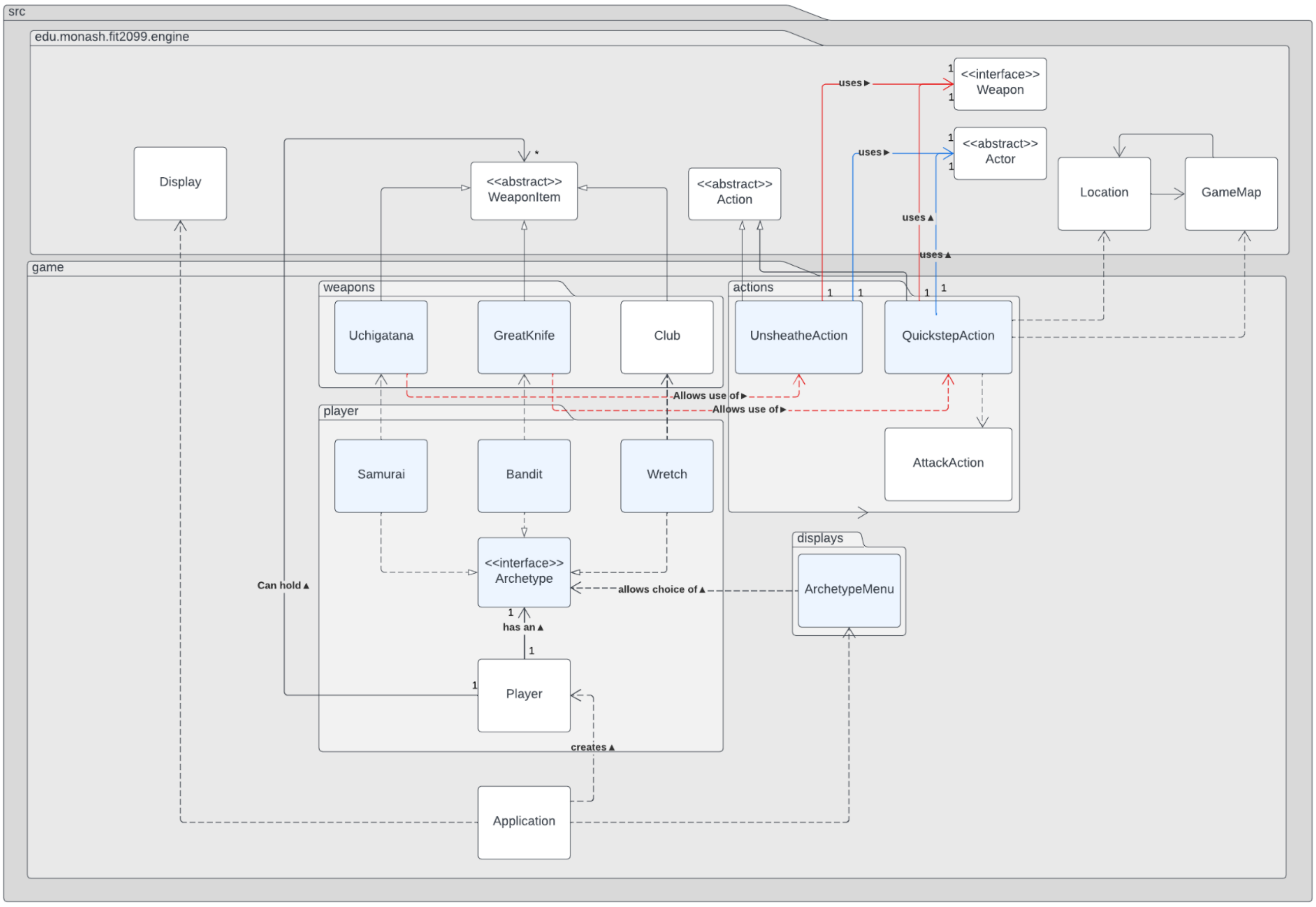
Req 2 UML Diagram:



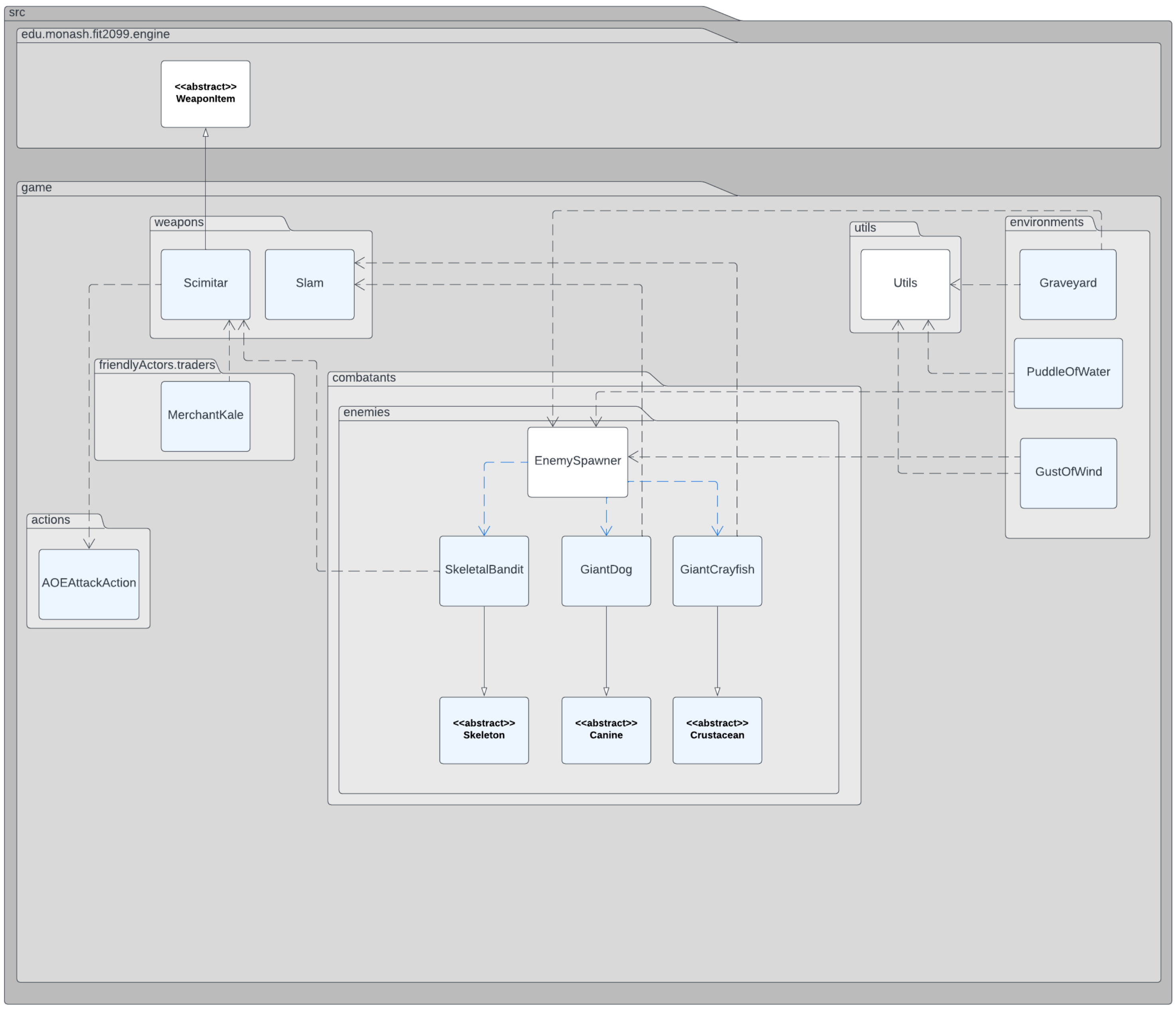
Req 3 UML Diagram:



Req 4 UML Diagram:

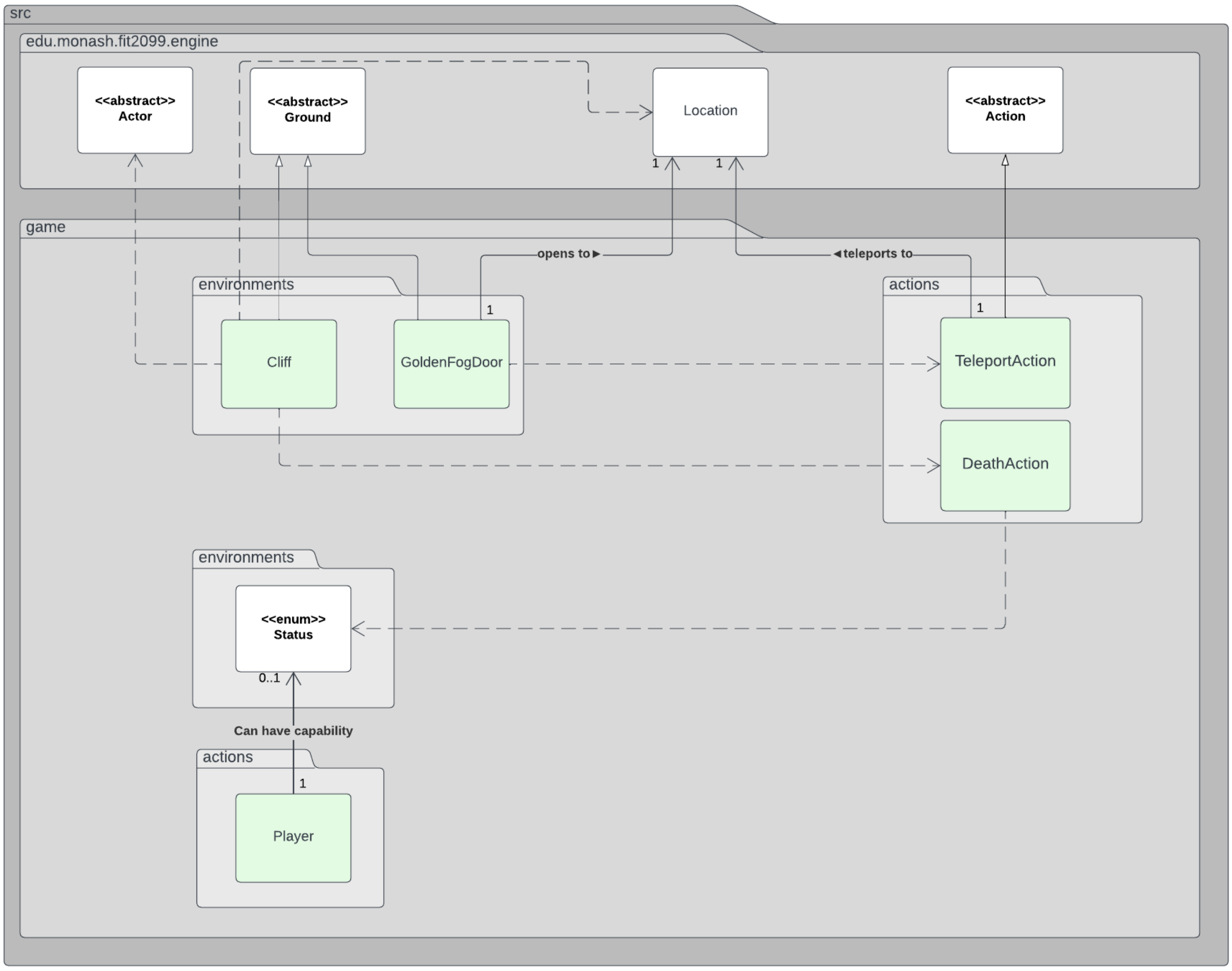


Req 5 UML Diagram:

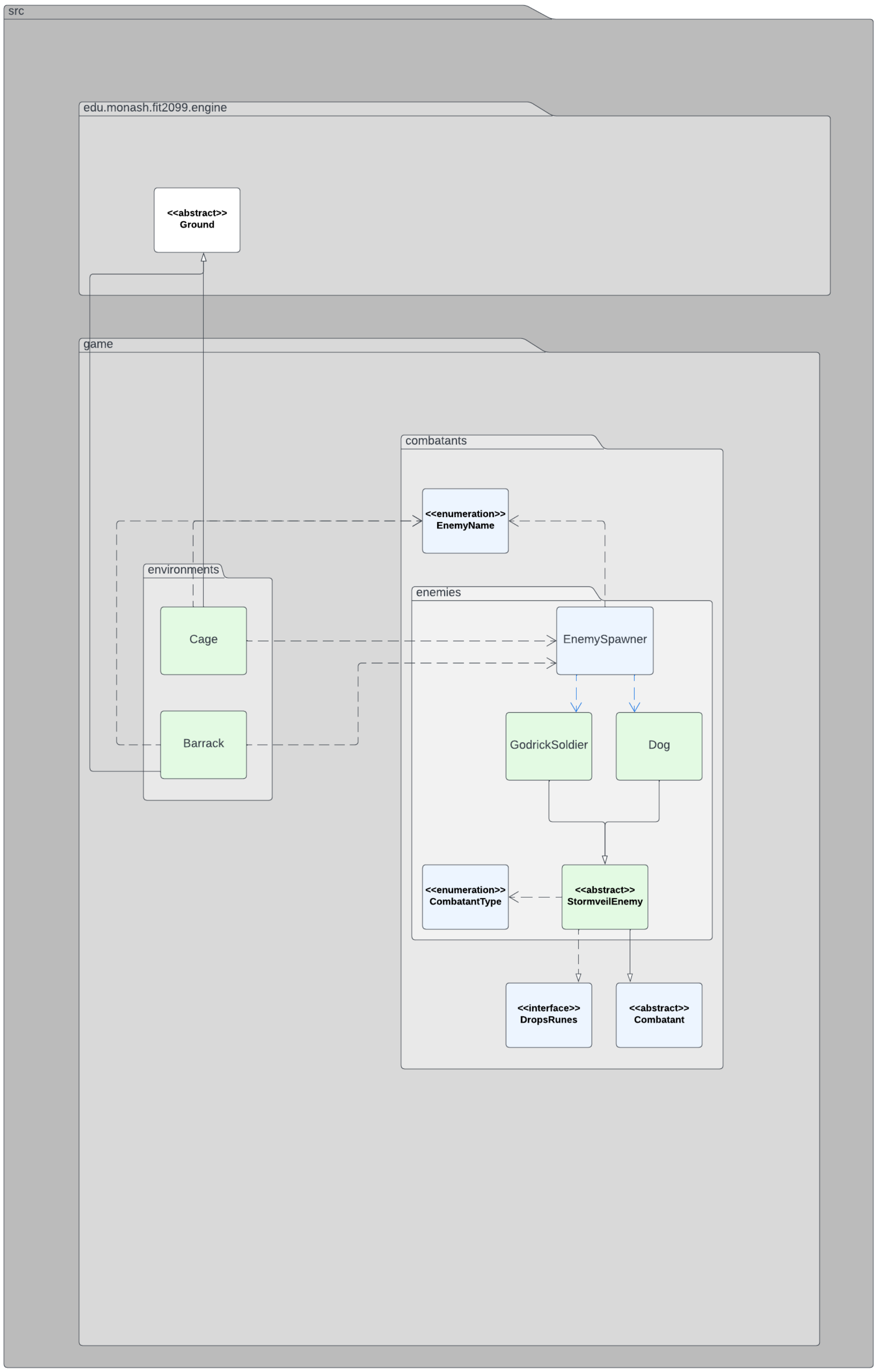


Assignment 3 UML Diagrams

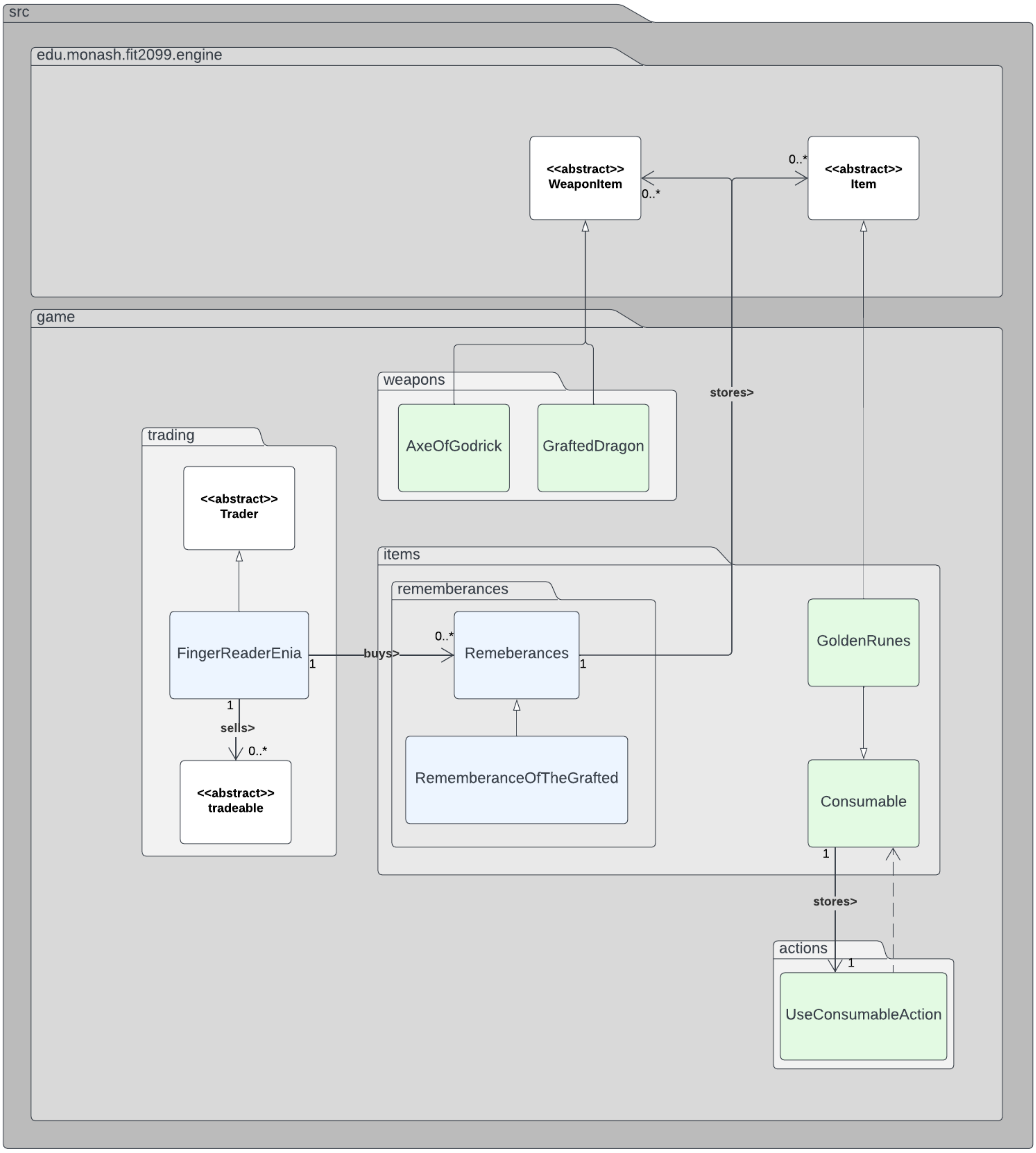
Req 1 UML Diagram:



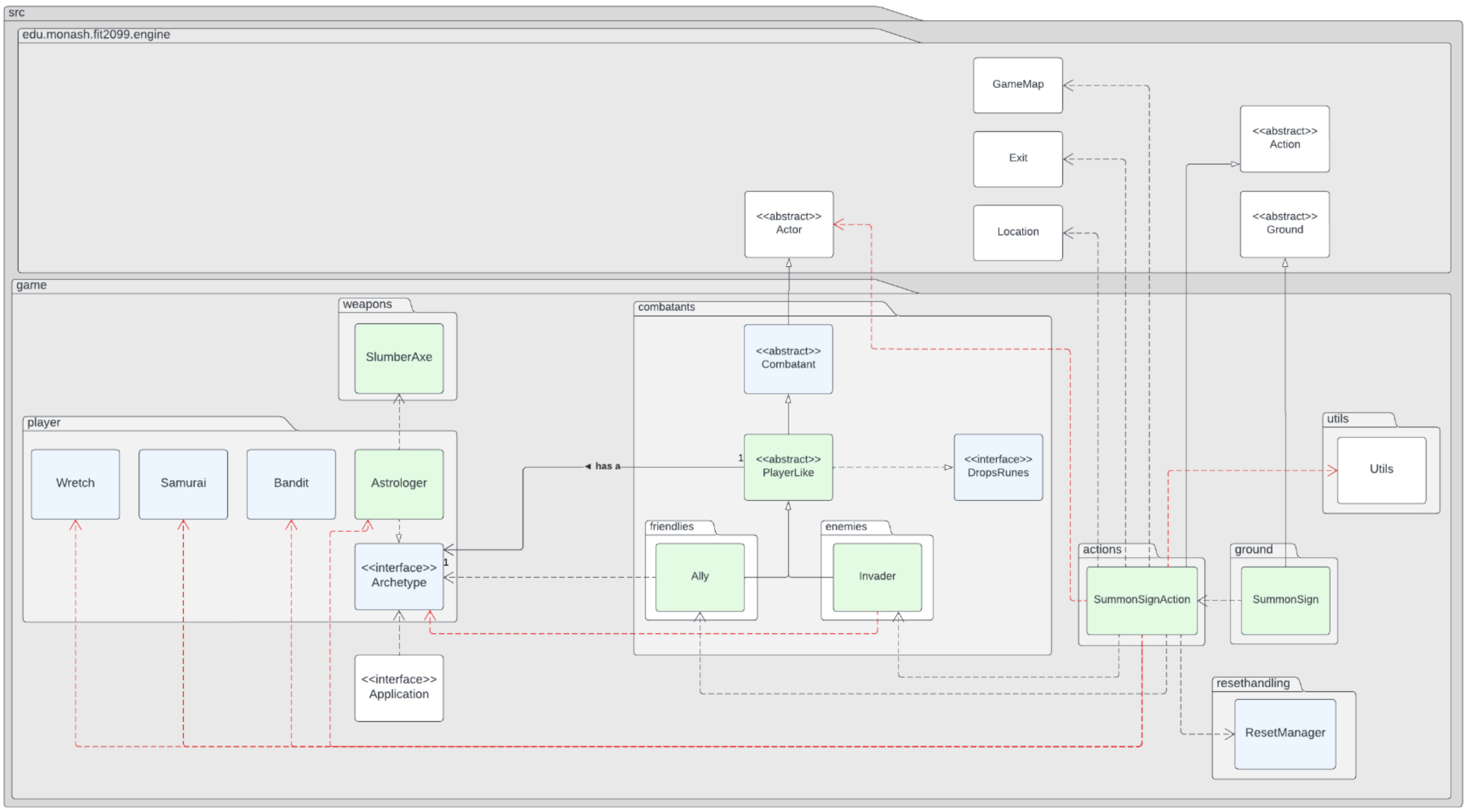
Req 2 UML Diagram:



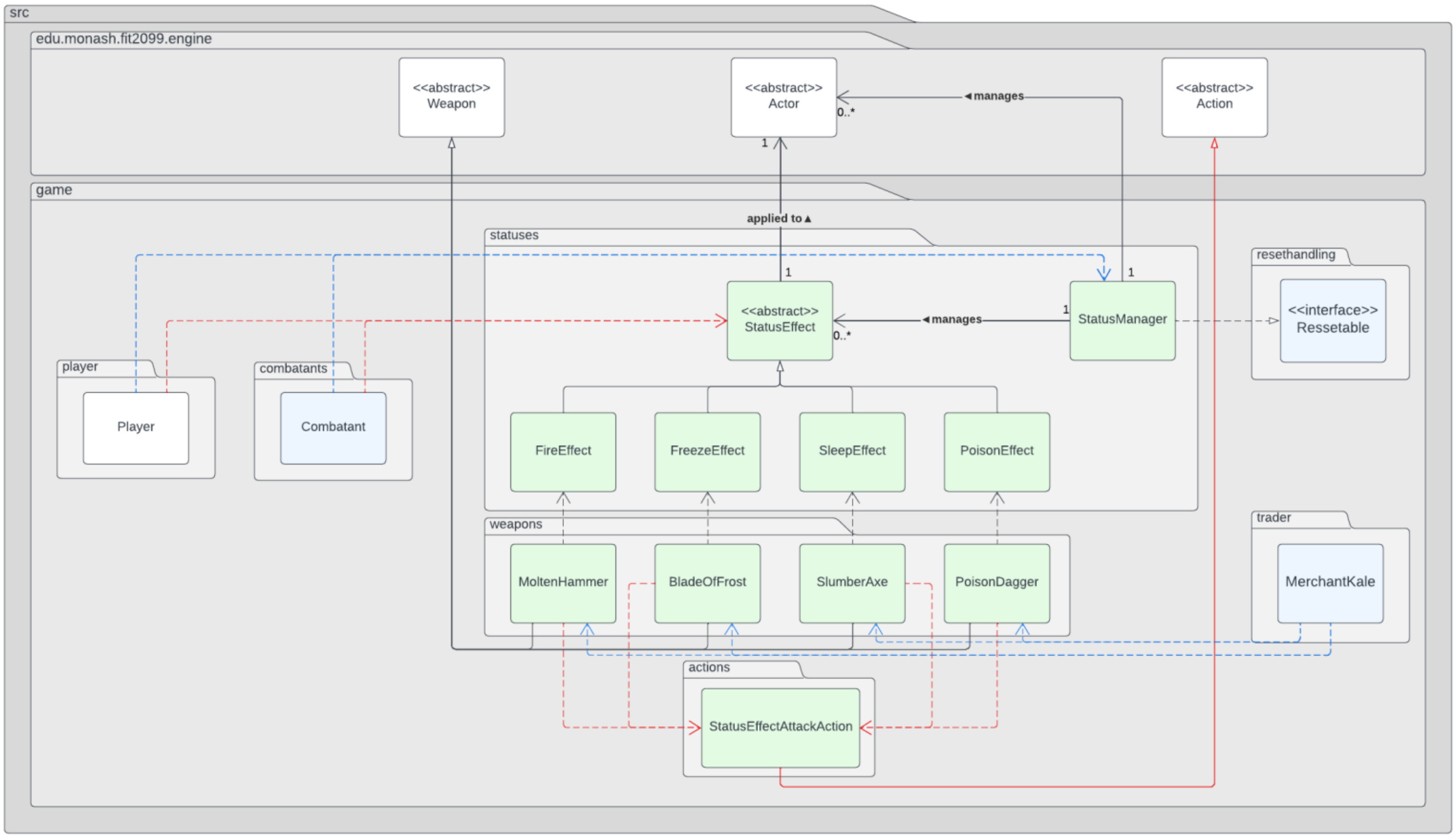
Req 3 UML Diagram:



Req 4 UML Diagram:



Req 5 UML Diagram:



Design Rationale and Class Descriptions

Changes from Assignment 2:

Req 1: Enemies

- Abstract classes were implemented for each type of enemy; this follows DRY however may lead to long class extension. This allows for easy extendabilty by allowing the addition of new enemies for specific types.
- Follow behaviour is added within the combatant abstract class, if any actor with the FOLLOWED_BY_ENEMY status comes within 2 blocks from an enemy, this allows for easy extendability as if a new actor is added that can also be followed by the enemy, they can just get the FOLLOWED_BY_ENEMY status.
- PileOfBones holds the instance of the skeleton that was killed and respawns the same instance. This removes the need to create a new skeleton every time.
- The Utils class now finds the parity of the location by acquiring the GameMap that the location is on. This solution allows for any number of GameMaps to be added without any changes required.

Req 1: Weapons

- AOEAttackAction calls an AttackAction for each actor to be attacked, this greatly reduces repeated code, upholding DRY and upholding SRP as AOEAttackAction only has the responsibility of finding all targets for the attack, however this increases the dependencies between the two classes.

Req 2: Trader and Weapons

- A massive overhaul of this requirement occurred.
- BuyAction and SellAction have been replaced by the TradeAction, as the distinction between the 2 actions was not enough to warrant separation and led to code repetition.
- A Tradeables interface was created that allows for adherence to the Single Responsibility Principle, as classes implementing the interface are simply adding the capability to the singular concept of tradability.
- Runes implements this interface as they have a unique method of being traded (by using RuneManager), and the Item and Weapon Tradeable Wrappers are created such that all logic for removing and placing into the actor's inventories are encapsulated within each class.
- This allows for a large level of extension, as any item or weapon can be traded using this technique (follows Open Closed Principle)
- Doing this also allows for the prices of items or weapons to be anything that is tradeable, also allowing for a massive amount of extendibility (weapon can be traded for weapon)

Req 3: Flask of Crimson Tears

- Removed HealConsumable interface and added Consumable abstract class, which is inherited by the HealItem abstract class. This allows for all consumables (not just heal consumables) to share an abstract class and removes need for repeated code.

    - This allows for new consumables to also inherit the new abstract class, abiding by the open-closed principle.
- Removed the HealConsumable interface and created the Consumable abstract class that all consumables should inherit.
- Created a useConsumableAction so that any consumable may be used.
- This allows immense extendibility without need for modification, as any consumable can utilise this Action (Open Closed Principle followed)
    - It can achieve this because of the Single Responsibility Principle, as each consumable class is fully responsible for all aspects related to it (removing itself, keeping track of uses, descriptions.

Req 3: Runes

- The pickup rune action no longer inherits from the Pickup Action, and so the Liskov Substitution Principle is no longer violated. Instead, the Pickup Rune Action inherits from Action.

Req 4: Weapons

- Altered QuickStepAction to utilise AttackAction to deal damage, abiding by DRY principles.
- Changed way in which QuickStepAction searching for valid locations for actor to be moved, by checking possible exits. This again abides by DRY principles and improves readability of code.

Req 5: Enemies

- Abstract classes added for Req 1 were also used here, following DRY principles.

Assignment 3 Design Rationale:

- *Req 1: Golden Fog Door*
- The GoldenFogDoor extends the ground abstract class as to follow DRY methodology and OCP.
    - This is because all environment classes share common attributes and methods.
- The GoldenFogDoor returns a new TeleportAction in its allowableActions method, allowing the player to travel to the destination location.
    - This follows SRP by having the transporting of the player being separate from the GoldenFogDoor class.
    - This allows for easy extendibility as if another ground/actor allows the player to teleport, it can use the TeleportAction
- TeleportAction extends the Action abstract class, following DRY methodology and OCP.
    - This is because the TeleportAction uses the same methods as in the Action abstract class.

- The Cliff class extends the Ground subclass such that DRY principles are followed. Moreover, to ensure no code repetition occurs, the Cliff class creates a DeathAction for the player if they are stood above them to so that conflicting methods of killing actors would not exist.
    - To achieve this, the DeathAction had to be modified, as well as the player. However, this change was merely to make the class more robust, as a new status WAS_KILLED was created to ensure that the player class is truly informed whether a death action has occurred. So, while modification did have to occur, this was ultimately allows it to be reused in a greater number of areas.

Req 2: Enemies

- The Dog and Godrick Soldier will extend the StormveilEnemy abstract class, which in turn extends Combatant abstract class, which in turn extends the Actor abstract class.
    - This allows for easy extendibility as new stormveil enemies only need to extend the StormveilEnemy abstract class to receive most of their functionality, for example, a new stormveil mage enemy would just extend the StormveilEnemy class to receive the allowable actions and play turn functionalities, thus upholding the open-closed principle.
    - Pros: This upholds the DRY principle, as the repeated code of the actors and enemies will be replaced with abstractions.
    - Cons: There is an increase by a level of inheritance.
- Enemies will attack the player and all enemies except those with the same type Enum (StormveilEnemy, Skeleton, etc.), thus the dog and Godrick soldier wont attack one another, but will attack all other enemies.
    - This allows for easy extendibility, as a new type of enemy just requires a new Enum to be added, for example, an undead type of enemy would just need an UNDEAD Enum, thus upholding the open-closed principle.
- The new Barrack and Cage environments inherit the Ground class to ensure DRY is adhered to. They also utilise the same method of spawning (using the EnemySpawner class) to ensure that the Single Responsibility Principle is followed.

Req 3: Golden Runes

- With the new Consumable refactoring, the Golden Rune simply inherits the abstract class so that DRY is adhered to, and all logic is related to using the golden rune is encapsulated within the class so that the Single Responsibility Principle is followed.

Req 4: Finger Reading Enia

- The remembrance abstract class was created that inherits from the Item abstract class such that DRY is adhered to. The Remembrance of the Grafted then inherits this class for similar reasoning. The remembrances also store their possible trades so that encapsulation and the Single Responsibility Principle is followed.
- Enia also inherits from the Trader so that it DRY is adhered to.

Req 4: New Role

- New Astrologer combat archetype extends the Archetype interface.
    - This abides by OCP and DRY principles.

Req 4: Summon Sign

- SummonSign class extends the Ground abstract class.
    - Allows for inheritance of ground class's methods, abiding by DRY and OCP principles.
- New action SummonSignAction inherits the Action abstract class and is added to the SummonSign class' allowable actions, if the actor has the status IS_PLAYER.
    - This abides by DRY and OCP principles.
    - Ensures the player is the only actor that can interact with summon sign.
- SummonSignAction stores all the archetypes the Allies/Invaders can possibly have in an ArrayList.
    - One con of this implementation is that it means that the SummonSignAction class is dependent on all the archetype classes.
    - Another downside of this implementation is that when any new archetypes are added, they must be added to this archetype list if they should be able to be used by the Allies/Invaders.
    - On the other hand, this could be seen as a positive as it means that not all archetypes have to be used by both the player and the Allies/Invaders, improving customizability in the future.
- The execute method of the SummonSignAction first checks all the location's exits to find a valid location for the ally/invader to be spawned.
    - Using the exits here instead of checking each location individually allows for simplified code and abides by DRY principles.
- It then randomly selects and archetype from the class' archetype list and spawns an ally or invader with a 50% chance each.

Req 4: Allies/Invaders

- Both Allies and Invaders inherit a PlayerLike abstract class
    - This abides by DRY principles by reducing the shared abilities of the two entities, with both having a combat archetype.
- The PlayerLike abstract class extends the Combatant abstract class and passes to the constructor true for the enemy attribute for the invader, and false for the ally.
    - This implementation allows for the combatant class to be used for both enemies and actors friendly to the player, abiding by DRY and OCP.
- The Invader has the capability invader, which makes sure that invaders don't attack other invaders (checked in the combatantAttackBehaviour method).
- The Ally has friendly capability.
    - This makes sure that not only do allies not attack other players, they also don't attack the player, as the player also has this friendly capability.

Req 5: Creative Requirement - Status Effects and Weapons

- StatusEffect abstract class is inherited by all status effects and defines the methods for the number of turns a status effect lasts. Also holds the attribute for the actor the status effect has been applied to.
    - This abides by DRY and OCP principles, by reducing the repeated code between status effects. Also allows for LSP to be used in the actor's play turn, only needing to expect for a StatusEffect object to be given instead of checking specific status effects.
- StatusManager is used to map each actor to a list of status effects that they currently have.
    - StatusManager utilises a singleton design pattern, which can lead to tightly coupled code.
- In a player/combatant's playTurn method, before allowing the actor to do an action, the list of their current statuses is iterated through, and the execute method for any statuses they have is run.
    - This allows for status effects which immobilise an actor to force it to do a DoNothingAction, instead of allowing the choice of an action.
- The new weapons all implement the WeaponItem abstract class.
- Each of the weapon's which apply a status effect override the getSkill method from the Weapon interface and return a StatusEffectAttackAction.
    - The specific status effect applied by the weapon is passed to the constructor of StatusEffectAttackAction.
    - Implementing things this way means that only one action class is needed for all weapons which inflict a status effect, instead of each having their own action.

    - This allows for simplified code and abides by DRY principles.