REQ 1:

What classes will exist in your extended system:

- Gust of Wind
- Graveyard
- Puddle of Water

All of these new environment classes inherit from the abstract parent class 'Ground'.

- Heavy Skeletal Swordsman
- Lone Wolf
- Giant Crab

All of these new enemy classes will inherit from the abstract parent class 'enemy'.

These classes relate to and interact with the existing system by inheriting from an abstract parent class. Each of the new environments inherits from the abstract parent class 'ground' - as they are ground that the player can walk on. Each new environment also has an association with the enemy type unique to that environment.

The player will be able to walk over the new environments. The environments, each turn, will have a unique chance of spawning the enemy unique to each environment - which will be able to interact with the player in combat.

My design follows OOP principles by avoiding repetition (DRY) by each new concrete class inheriting from the abstract 'ground' class, as opposed to creating a brand new class for each new environment. Similarly, this means that more unique environments can be created in the future by inheriting from the abstract 'ground' class. Each new child class also follows the single responsibility principle by handling their own unique properties; there are no overlapping responsibilities between classes. Each class denotes its own unique chance of spawning its enemy, and which enemy to spawn - thus following SRP.

The diagram also represents an object-oriented system for the enemies system within a rogue-like adventure game. This includes 3 concrete classes implementing 1 abstract class.

The concrete enemies classes inherit the abstract class because many common attributes and methods are shared, hence the abstract class is made to avoid repetitions (DRY).

The enemies abstract class also has many dependencies and associations with other classes such as Behaviours, Application, Actor and AttackAction. By introducing this abstract class, we are able to reduce many dependencies that each concrete class may have as well as ensuring that if changes are to be made to those enemies classes such as balancing or reworking how the enemies are implemented, it will be easy to refactor and extend the system without modifying the rest of the code. This also allows new enemies to be added without adding multiple dependencies with those classes (Open-Close Principle).

Req 2:
The diagram represents an object-oriented system for the runes system and trading within a rogue-like game. This includes multiple concrete classes working together.

The concrete class RunesManager is introduced as a way to manage all runes functionalities and interactions within the application. This includes updating the number of runes a Player may have after killing an enemy. The Runes interface is also implemented to help out with tracking the number of runes of each Actor. Its sole purpose is to update and return the number of runes a player may have (Single Responsibility Principle).

The Runes interface is also implemented to avoid having to change implementation of the Player and Enemy classes if RunesManager were to receive a change or update to the class which may include the methods on calculating how runes are calculated (Dependency Inversion Principle).

To implement the Trader, we had to consider which items were buyable and sellable - the GrossMesser can only be sold to the Trader but cannot be bought from him. Therefore, to distinguish between which items can be bought and sold we created two interfaces for buyable and sellable items - as opposed to a singular tradable interface (interface segregation principle). However, having two interface classes for buyable and tradeable can be considered an overapplication of SRP, as the classes may get too small. In our implementation, in terms of classes, they will be from the players POV, as the player can sell the Grossmesser but cannot buy it.

Req 3:
The classes introduced in requirement 3 describe the special actions within the game including consuming a Flask of Crimson Tears and resting at the Site of Lost Grace. It also implements how resetting the game will function.

The FlaskOfCrimsonTears and SiteOfLostGrace class is introduced as a means to have those items within the game, whereas the ConsumeAction and RestAction classes are introduced to be able to perform certain actions, as they inherit from the Action abstract class. These classes are only introduced to perform special actions that may not appear elsewhere in the game, and can provide further uses if newer classes that also use those actions are added to the game. This follows the Single Responsibility Principle.

Resetting the game changes the states of the Player, Enemy and Runes classes. By using the resettable interface given, we are able to add these classes to the ResetManager class using the Liskov Substitution principle, as they are all different base classes but inherit from the same interface. This is also an application of the Dependency inversion principle since if we decide to change how the mechanics of resetting works in the game, it will not affect any of the classes that implement the interface.

Req 4:
The new classes that will be introduced in the extended system for requirement 4 will be Samurai, Bandit and Wretch classes for the combat archetypes that the player can select. Classes for the unique weapons of each combat archetype - the Uchigatana, the Great Knife and the Club - will also have their own class. Finally two new classes for the unique skills that the Uchigatana and the Great Knife allow the player to perform will be created - Unsheathe and Quickstep respectively.

The roles of the new classes are to provide the base unique stats, weapons and abilities to the player depending on the selected combat archetype. By having each combat archetype separated, we will support SRP and furthermore, by having them as subclasses of player, we will uphold LSP as the combat archetype classes can be used in place of Player.

The Samurai, Bandit and Wretch classes will inherit the Player class, and they will have dependencies with the Uchigatana, GreatKnife and Club respectively. The three weapons will inherit from the abstract class WeaponItem. The Uchigatana will inherit the UnsheatheAction and GreatKnife will inherit the QuickStepAction, both actions will inherit from the abstract Action class.

Req5:

The classes introduced in requirement 5 are the new enemies that exist in the East side of the map. The new classes are GiantCrayfish, SkeletalBandit and GiantDog, along with the new weapon of the SkeletalBandit, the Scimitar.

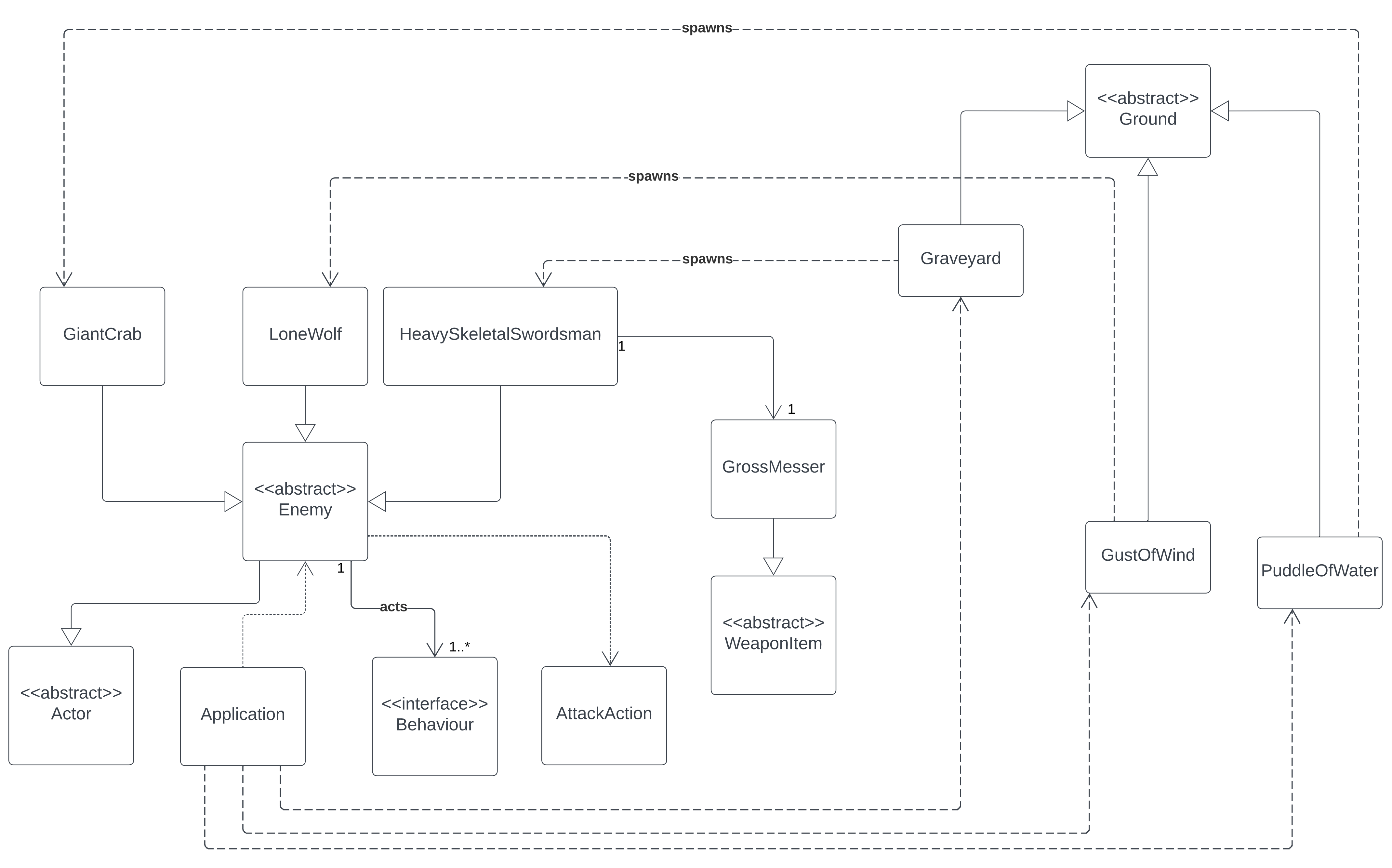
The new enemies will inherit the abstract Enemy class that we created in Req 1 and the Scimitar will inherit from the abstract class WeaponItem. Instead of having new abstract classes for East and West to differentiate between where the enemies spawn, we decided to have them just have dependencies with their spawn locations similar to Req 1, and will differentiate where they can spawn using a boolean variable in each Environment (isEast or isWest) to uphold the DRY principle.

As an example of OCP, each environment class from Req 1 has been extended to spawn different enemies depending on whereabouts the environment is on the world map.

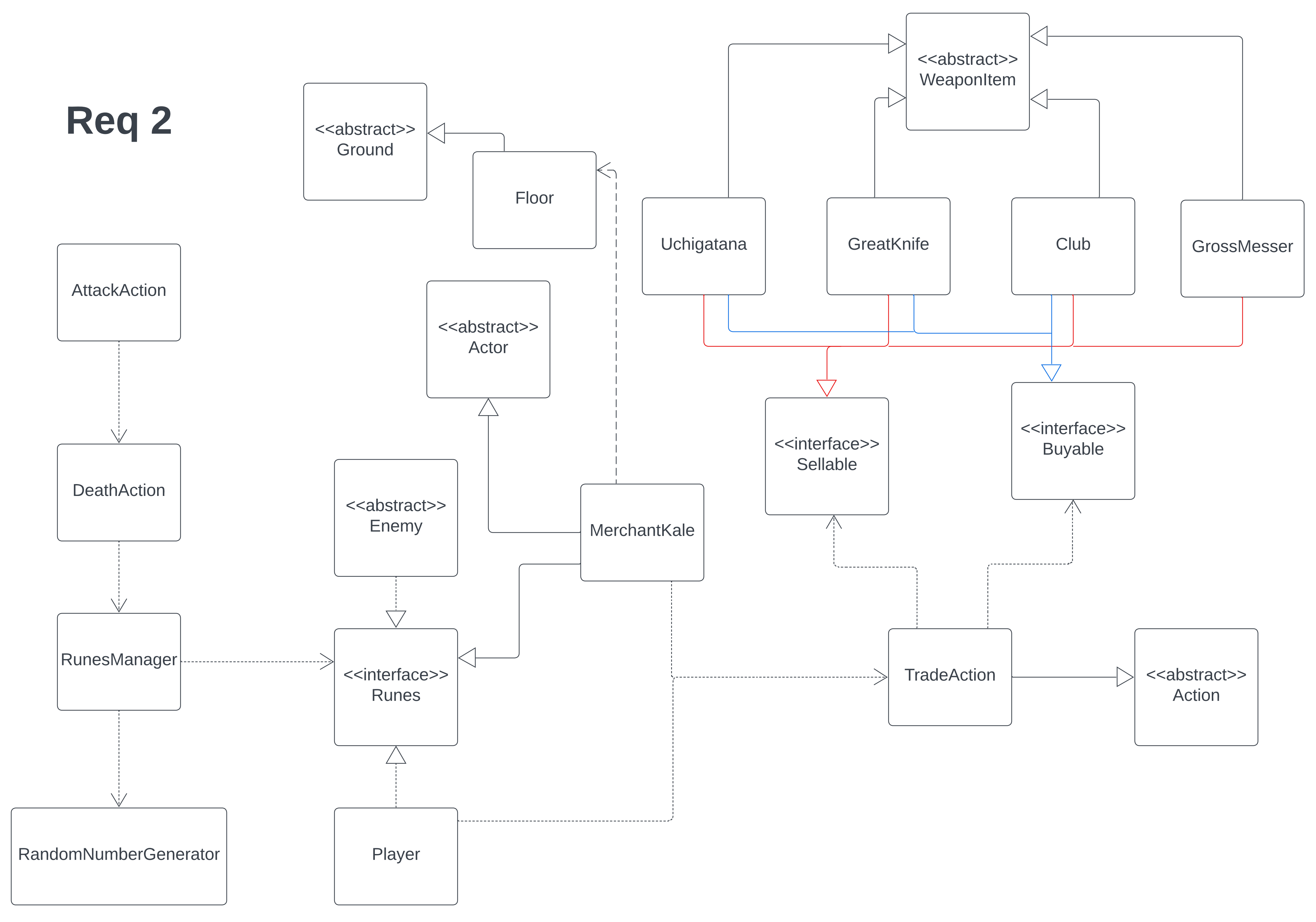
Contribution Log Link

https://docs.google.com/spreadsheets/d/1ESvnFQNk3XJ2QLDXeMLZOv2KC0u1jdanL7zptNXUyIY/edit?usp=sharing

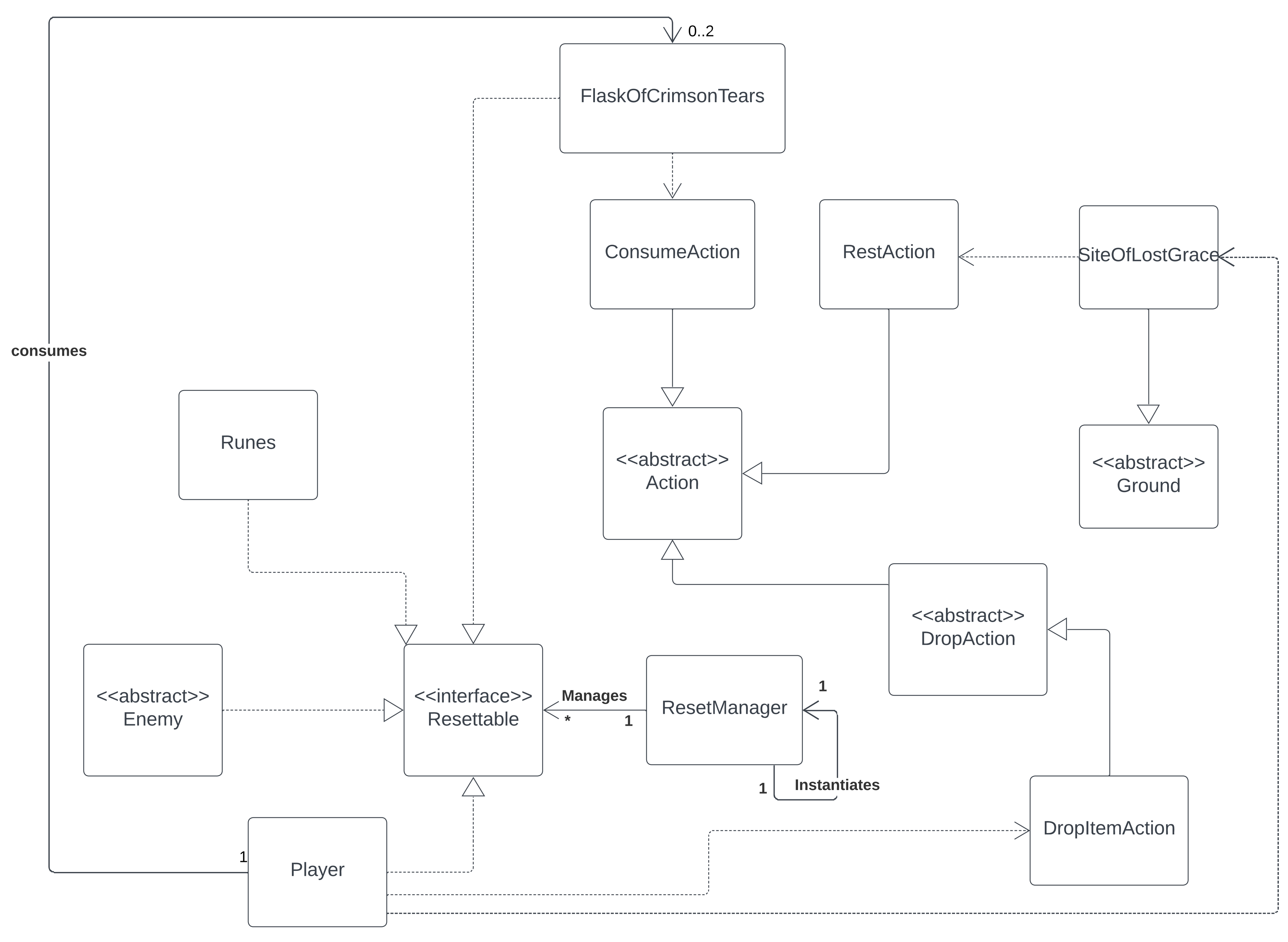
Req 1



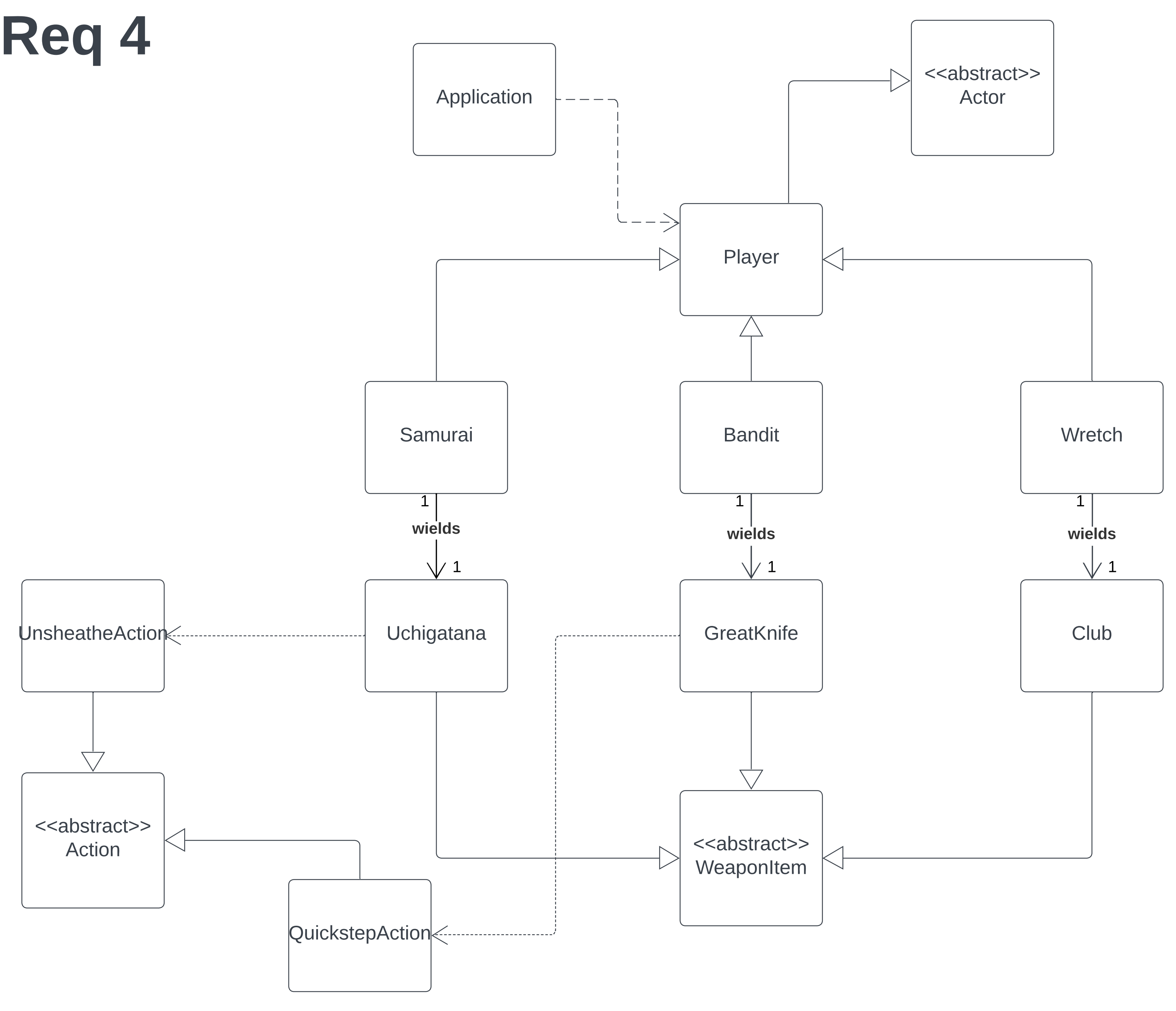
Req 2



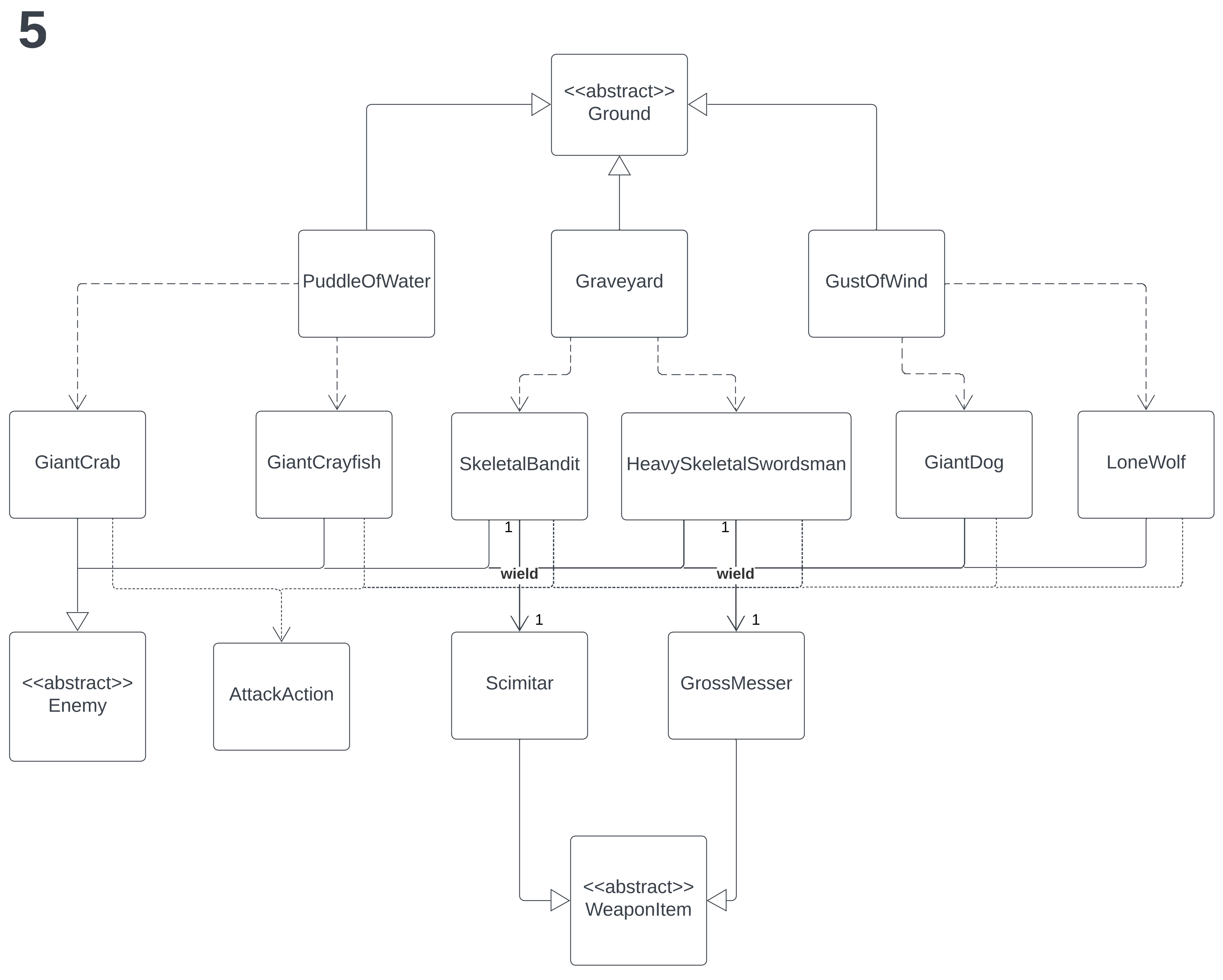
Req 3



Req 4



Req 5



<<abstract>> Ground
PuddleOfWater
Graveyard
GustOfWind
GiantCrab
GiantCrayfish
SkeletalBandit
HeavySkeletalSwordsman
GiantDog
LoneWolf
1
wield
1
1
wield
1
<<abstract>> Enemy
AttackAction
Scimitar
GrossMesser
<<abstract>> WeaponItem