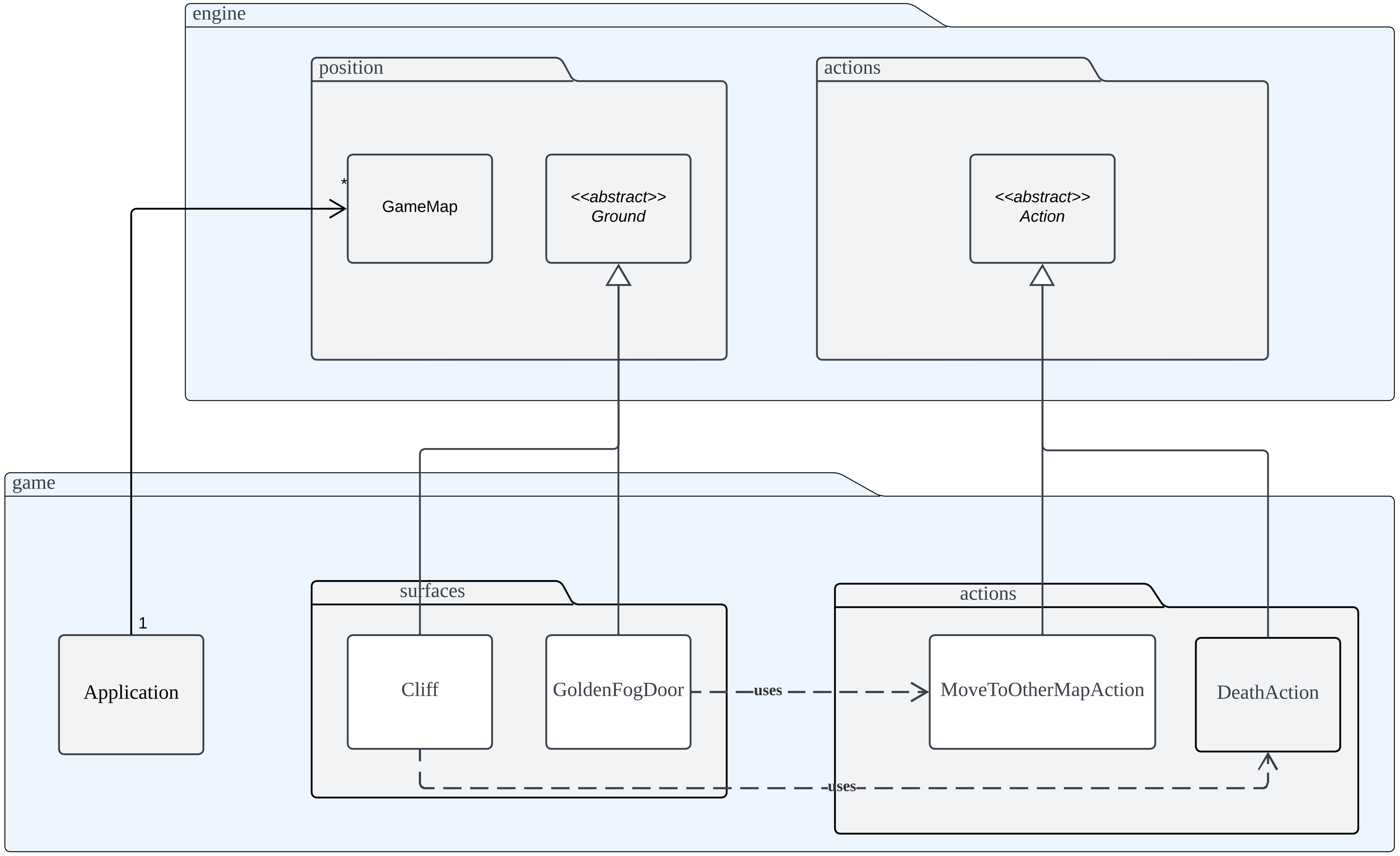
Req1

The color difference in the diagram has several reasons. The packages and classes with grey color are originally in the UML diagram provided and we added in Assignment 1. As for the packages and classes with white color are the addons in Assignment 3. For the following discussion, only the addons packages and classes are going to be mentioned.

We have 2 packages named surfaces and actions. In the surfaces package, there's 2 class named Cliff and GoldenFogDoor and in the actions package, there's a class named MoveToOtherMapActions. Both Cliff and GoldenFogDoor class extends the abstract Ground class as for the MoveToOtherMapActions class extends from the abstract Action class. We did this to make sure that we don't have repeated codes and hence fulfill the DRY (Don't repeat yourself) principle. In the GoldenFogDoor class, there's a method called setGoldFogDoor. This is to make sure that we fulfill the SRP (Single Responsibility Principle) as we will be setting doors for each map in the Application class.

Now we are going to explain the non-abstraction relationship. Cliff class has a dependency relationship with DeathAction class, this is because when a player is on the cliff, player will die, and hence Cliff class triggers the DeathAction to cause the player to die. As for GoldenFogDoor, it has a dependency relationship with MoveToOtherMapAction. Example, when the player is on the GoldenFogDoor he/she can choose to go whether to go to the other map or not, if yes then the MoveToOtherMapAction will be triggered.


engine
position
actions
GameMap
<<abstract>> Ground
<<abstract>> Action
*
game
surfaces
actions
Application
1
Cliff
GoldenFogDoor
MoveToOtherMapAction
DeathAction
uses
uses

Req2

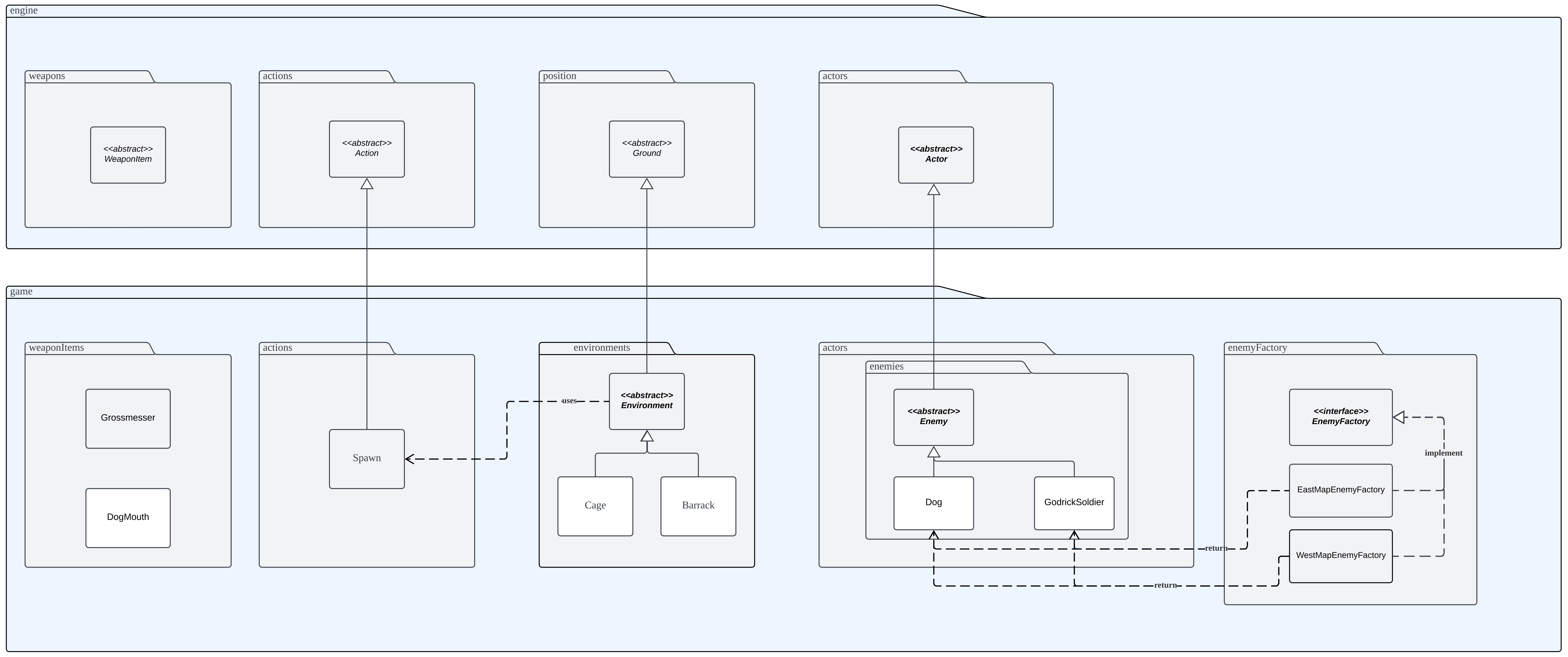
The color difference in the diagram has several reasons. The packages and classes with grey color are originally in the UML diagram provided and we added in Assignment 1. As for the packages and classes with white color are the addons in Assignment 3. For the following discussion, only the addons packages and classes are going to be mentioned.

Inside environments package, we have added two classes named Cage and Barrack which can spawn Dogs and Godrick Soldiers. Both classes extend the abstract Environment class as they share some common attributes and methods, it is logical to abstract these identities to avoid repetitions (DRY - Don't Repeat Yourself principle)

In addition, we have Dog class and Godrick Soldier class inside enemies package. Same as other enemies, they have follow behaviour, wander behaviour, they will drop runes once they are attacked by player etc. To avoid repetition of code, we make both classes extend the abstract Enemy class. (DRY - Don't Repeat Yourself principle)

In future, if there's new environments or enemies, we can create a new class and extend from the abstract class instead of rewriting common attributes and features which are really bad design.

However, there is a con for us when extending a new enemy. Every turn the environment spawn enemies, it always uses EnemyFactory to return an enemy's object. Inside WestSideEnemyFactory and EastSideEnemyFactory, we need to hardcode to return certain enemy after checking the environments, which would violate OCP (open-closed principle).


engine
weapons
<<abstract>> WeaponItem
actions
<<abstract>> Action
position
<<abstract>> Ground
actors
<<abstract>> Actor
game
weaponItems
Grossmesser
DogMouth
actions
Spawn
uses
environments
<<abstract>> Environment
Cage
Barrack
actors
enemies
<<abstract>> Enemy
Dog
GodrickSoldier
enemyFactory
<<interface>> EnemyFactory
implement
EastMapEnemyFactory
WestMapEnemyFactory
return
return

Req3

The color difference in the diagram has several reasons. The packages and classes with grey color are originally in the UML diagram provided and we added in Assignment 1. As for the packages and classes with white color are the addons in Assignment 3. For the following discussion, only the addons packages and classes are going to be explained.

Abstraction makes us more convenient to use the functions or attributes when both or more classes have same behaviours and properties. To adhere Don't Repeat Yourself (DRY), we make the following classes extend appropriate abstract classes:

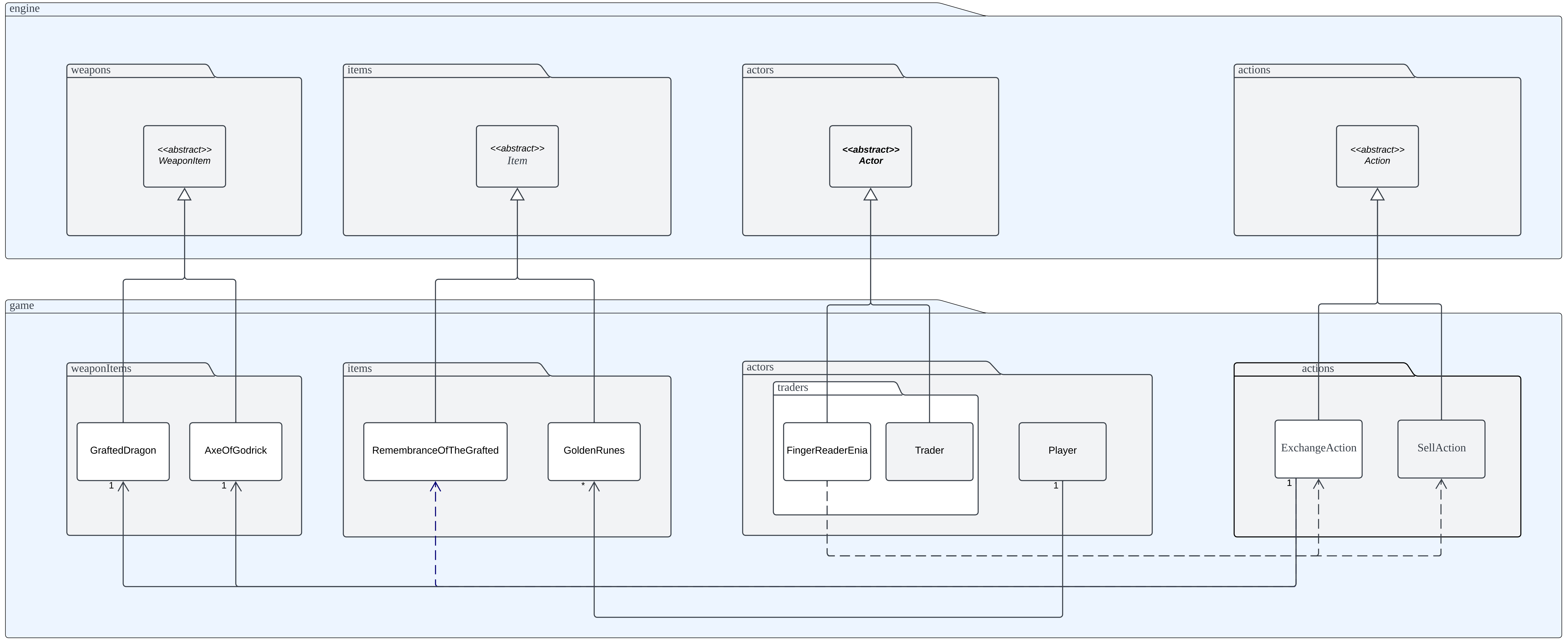
First, we have GraftedDragon class and AxeOfGodrick class which extend abstract WeaponItem class. Moving on, we added new items that are RemembranceOfTheGround class and GoldenRunes and make them extend the abstract Item class. A new trader named FingerReaderEnia add in our game extends abstract Actor class. Last but not least, there is an ExchangeAction extend abstract Action class.

We created a new package named traders package inside actors package as now we have 2 traders to put in. It would be more readable and understandable.

Now we are going to explain the non-abstraction relationship. GoldenRunes class has an association (one-to-many relationship) with Player, i.e. each player can have multiple GoldenRunes. Furthermore, when player meets FingerReaderEnia, player only can sell their weapon items or exchange either AxeOfGodrick or GraftedDragon using RemembranceOfTheGround item from the trader. Hence, FingerReaderEnia will use ExchangeAction (Dependency).

There are associations (one-to-one relationships) between AxeOfGodrick, GraftedDragon and ExchangeAction, I.e. every ExchangeAction will only have either one AxeOfGodrick or one GraftedDragon for player.

The disadvantage of using the approach above is that if there's a new weapon item can be exchanged, we need to modify the ExchangeAction which would violate OCP (open-closed principle).


engine
weapons
<<abstract>>
WeaponItem
items
<<abstract>>
Item
actors
<<abstract>>
Actor
actions
<<abstract>>
Action
game
weaponItems
GraftedDragon
AxeOfGodrick
1
1
items
RemembranceOfTheGrafted
GoldenRunes
*
actors
traders
FingerReaderEnia
Trader
Player
1
actions
ExchangeAction
SellAction
1

Req4

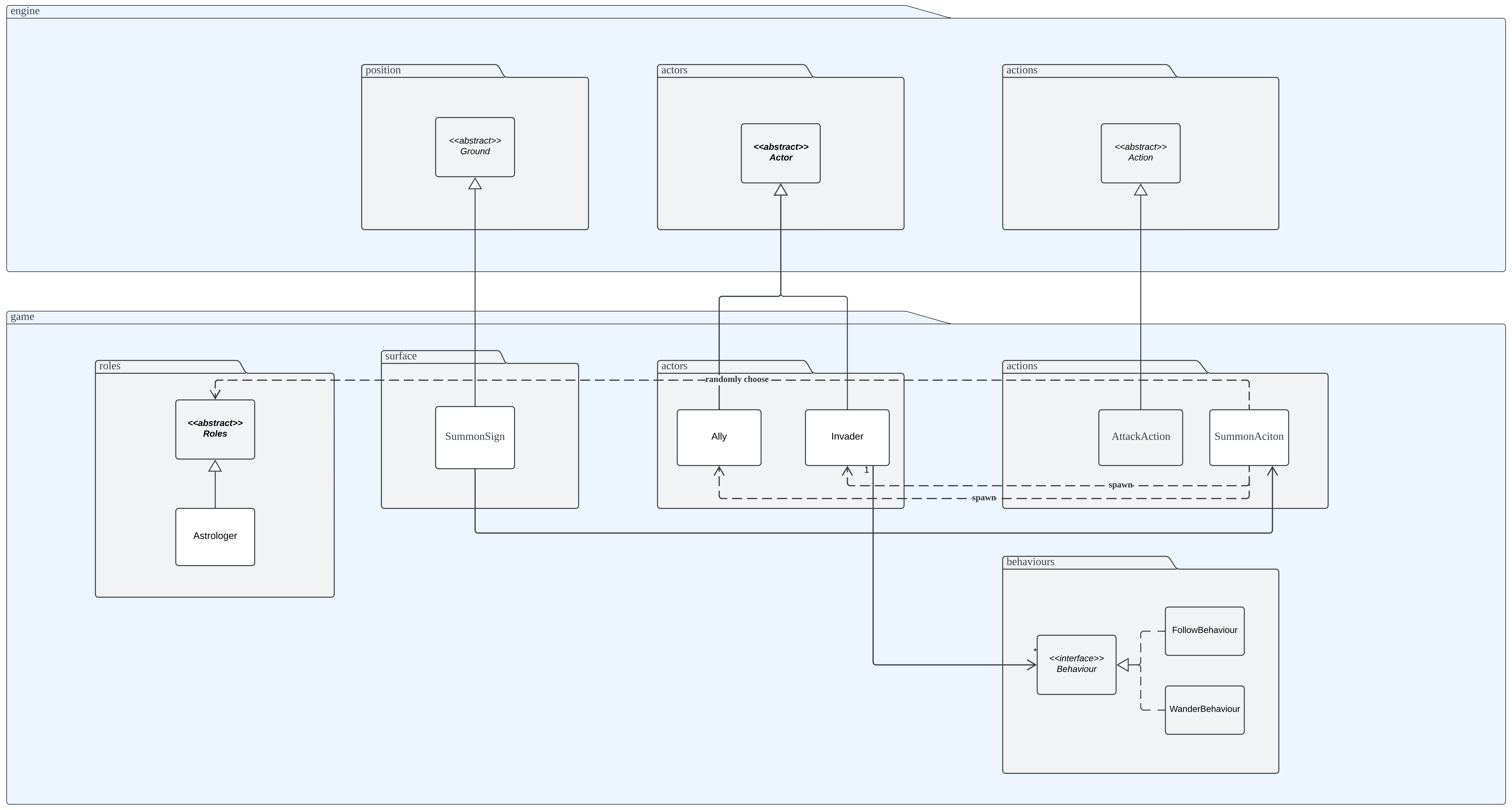
The color difference in the diagram has several reasons. The packages and classes with grey color are originally in the UML diagram provided and we added in previous assignments. As for the packages and classes with white color are the addons in Assignment 3. For the following discussion, only the addons packages and classes are going to be explained.

The principles we included were SRP, DRY, LSP.

Inside the roles package we have added a new class named Astrologer, in the surfaces package, we added a class named SummonSign. And in the actors package we added 2 classes named Ally and Invader. Astrologer class extends the Roles class. Astrologer is class similar to Bandit, Samurai and Wretch, the only difference between Astrologer and the rest is that has a different hp and weaopen. SummonSign class extends the abstract Ground class, Ally and Invader extends the Actor class. The reason we did this is to reduce the number of duplicate codes (DRY – Don't repeat yourself).

There's a new Action created named SummonAction, this class is responsible of generating either Ally or Invader. We created SummonAction so that each class related to summons would have a single responsibility (SRP – Single Responsibility Principle).

Now we are going to explain the non-abstraction relationship. SummonSign has an association relationship with SummonAction, eg. The player can choose whether to summon or not when he/she is on/ nearby the SummonSign. Ally and Invader both have a dependency relationship with the SummonAction, as when the player chooses to perform the summon action, the SummonAction class is responsible for spawning either Ally or Invader. SummonAction also has a dependency relationship with the Roles class, eg. When they are being spawned it will trigger the Roles class. Invader class has an association relationship with Behaviour class as Invader also has wander and follow behaviour.


engine
position
<<abstract>> Ground
actors
<<abstract>> Actor
actions
<<abstract>> Action
game
roles
<<abstract>> Roles
Astrologer
surface
SummonSign
actors
Ally
Invader
randomly choose
actions
AttackAction
SummonAciton
spawn
spawn
1
behaviours
*
<<interface>> Behaviour
FollowBehaviour
WanderBehaviour

Req5

The color difference in the diagram has several reasons. The packages and classes with grey color are originally in the UML diagram provided and we added in previous assignments. As for the packages and classes with white color are the addons in Assignment 3. For the following discussion, only the addons packages and classes are going to be explained.

The concepts we included were the DRY, SRP, OCP and Inheritance and Abstraction.

In the surface package, we had added a new class named PoisonLandFill. It extends abstract Ground class. Since this surface and the environments share some common attributes and methods, it is logical to abstract these identities to avoid repetitions (DRY - Don’t Repeat Yourself principle).

In Assignment 3, we had added a new map called "PoisonMaze". This map and "ToxicLandFill" each have different responsibilities. "PoisonMaze" is responsible for defining the maze map, while "ToxicLandFill" is responsible for defining the behavior of the toxic ground. Each class is responsible for only one clear responsibility. (SRP- Single Responsibility Principle).

Besides, to attract player to actively enter the map, we created environments where runes and enemies can be generated. This suggests that scalability was designed with scalability in mind, extending the map by adding new runes and enemies to generate environments rather than modifying existing code. (OCP- Open-Closed Principle)

In addition, we had an abstract Ground class that has the common properties of all the surfaces and environments and hence only the specific properties would be coded into their respective concrete classes. This makes our code for each enemy class to be less redundant (Inheritance and Abstraction). If we need to add a new class in the future, we can just extend from the abstract class. This will be very convenient.

engine
has
use
actions
<<abstract>>
Action
items
<<abstract>>
Item
has
position
GameMap
<<abstract>>
Ground
actors
<<abstract>>
Actor
extends
depends on
game
actions
AttackAction
Spawn
AreaAttackAction
enemyFactory
<<interface>>
EnemyFactory
implement
EastMapEnemyFactory
WestMapEnemyFactory
environments
<<abstract>>
Environment
Barrack
surface
ToxicLandFill
actors
Player
enemies
GodrickSoldier
return
return
items
Runes
use
RuneManager