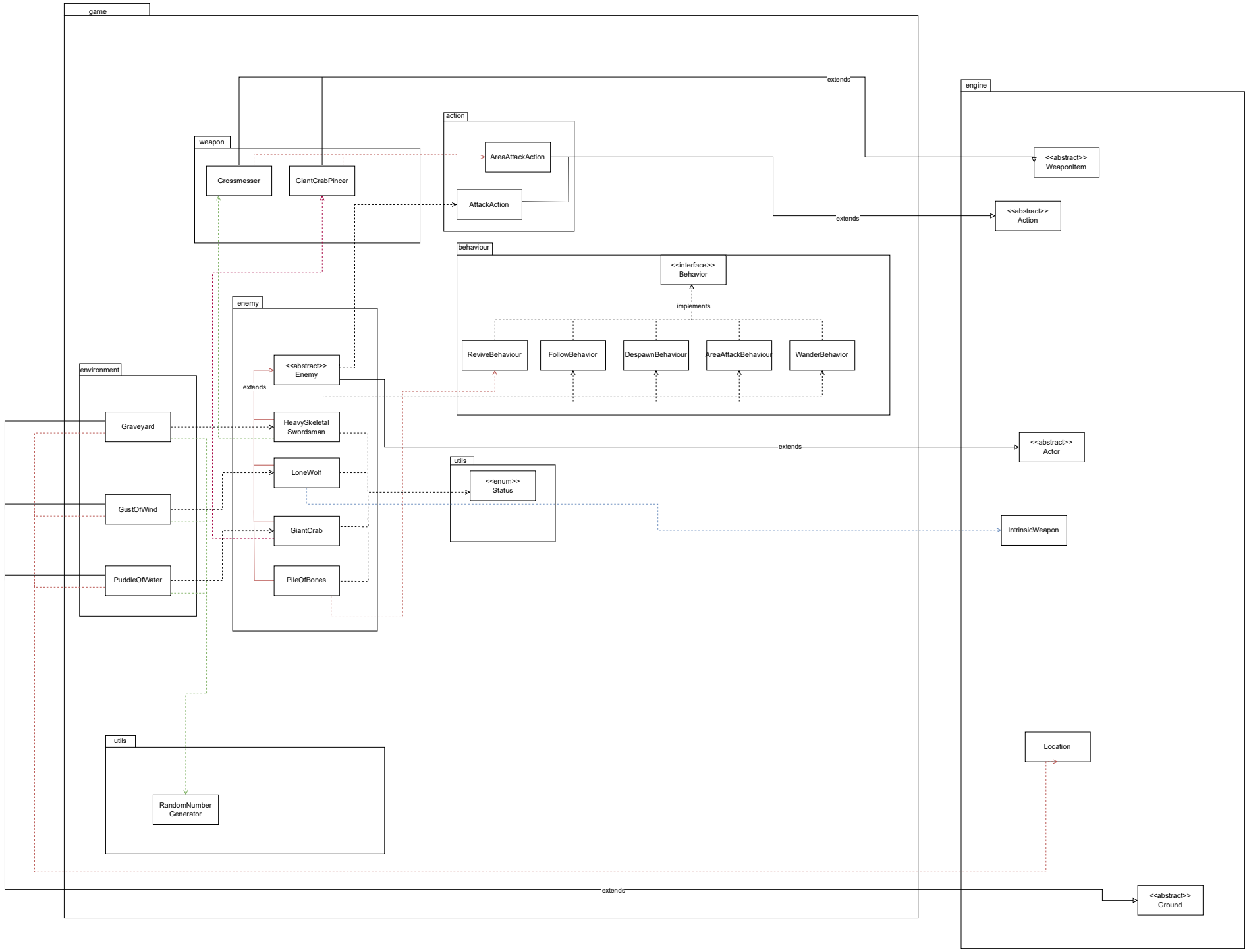
REQ 1


game
engine
weapon
Grossmesser
GiantCrabPincer
action
AreaAttackAction
AttackAction
extends
<<abstract>>
WeaponItem
<<abstract>>
Action
behaviour
<<interface>>
Behavior
implements
ReviveBehaviour
FollowBehavior
DespawnBehaviour
AreaAttackBehaviour
WanderBehavior
enemy
<<abstract>>
Enemy
environment
Graveyard
GustOfWind
PuddleOfWater
HeavySkeletal
Swordsman
LoneWolf
GiantCrab
PileOfBones
<<abstract>>
Actor
utils
<<enum>>
Status
IntrinsicWeapon
utils
RandomNumber
Generator
Location
<<abstract>>
Ground

REQ 1

A. Environments

Creating an “Environment” package to group all related environments together, making for better organisation.

“Graveyard”, “GustOfWind”, and “Puddle of Water” represents three different environment classes. They are extended from an abstract class “Ground”, so that similar methods can be grouped together (DRY principle). Having an abstract class also allows for easier expansion in the future. All three of them will have a dependency with “Location” class, as they will need to know which location they are at. “Graveyard” has a dependency with “HeavySkeletalSwordsman”, “GustOfWind” has a dependency with “LoneWolf”, and “PuddleOfWater” has a dependency with “GiantCrab”. These three dependencies are there, so that each environment can spawn their respective enemies. Since the spawning of enemies are by a chance of a certain percentage, all three environments will have a dependency with “RandomNumberGenerator, to use its method to get a random percentage, so that it can calculate whether to spawn an enemy or not.

B. Enemies

Creating an “enemy” package to group all related enemy classes together, making for better organisation.

In this requirement, four enemy classes are created; “HeavySkeletalSwordsman”, “LoneWolf”, “GiantCrab”, and “PileOfBones”. These four classes extends an abstract class “Enemy”, so that they can share commonly used methods (DRY principle). Doing this, it also makes extension easier; new implementation of enemies do not need to have too much code inside of them. All 4 enemy classes also have a dependency with a enumeration “Status”. This class allows us to categorise the enemy into their respective types. By doing this, it makes it easier to differentiate different types of enemies, and we can further specify what we want to do with each type.

Our abstract “Enemy” class is dependent on “AttackAction”, as this action tells us what other actors can do to this actor. “Enemy” is also dependent on “FollowBehaviour”, “DespawnBehaviour”, “AreaAttackBehaviour”, and “WanderBehaviour”. Since we cannot possibly tell each actor what to do each time, we count of behaviours to help us make the decisions. These behaviours implements a interface “Behaviour”, that acts as a factory for all behaviours.

“HeavySkeletalSwordsman” has a dependency on its weapon “Grossmesser”, while “GiantCrab” has a dependency on its weapon “GiantCrabPincer”. Meanwhile, “Lonewolf” has a dependency on “IntrinsicWeapon”, which is basically biting. These dependencies allow the enemies to use their respective weapons.

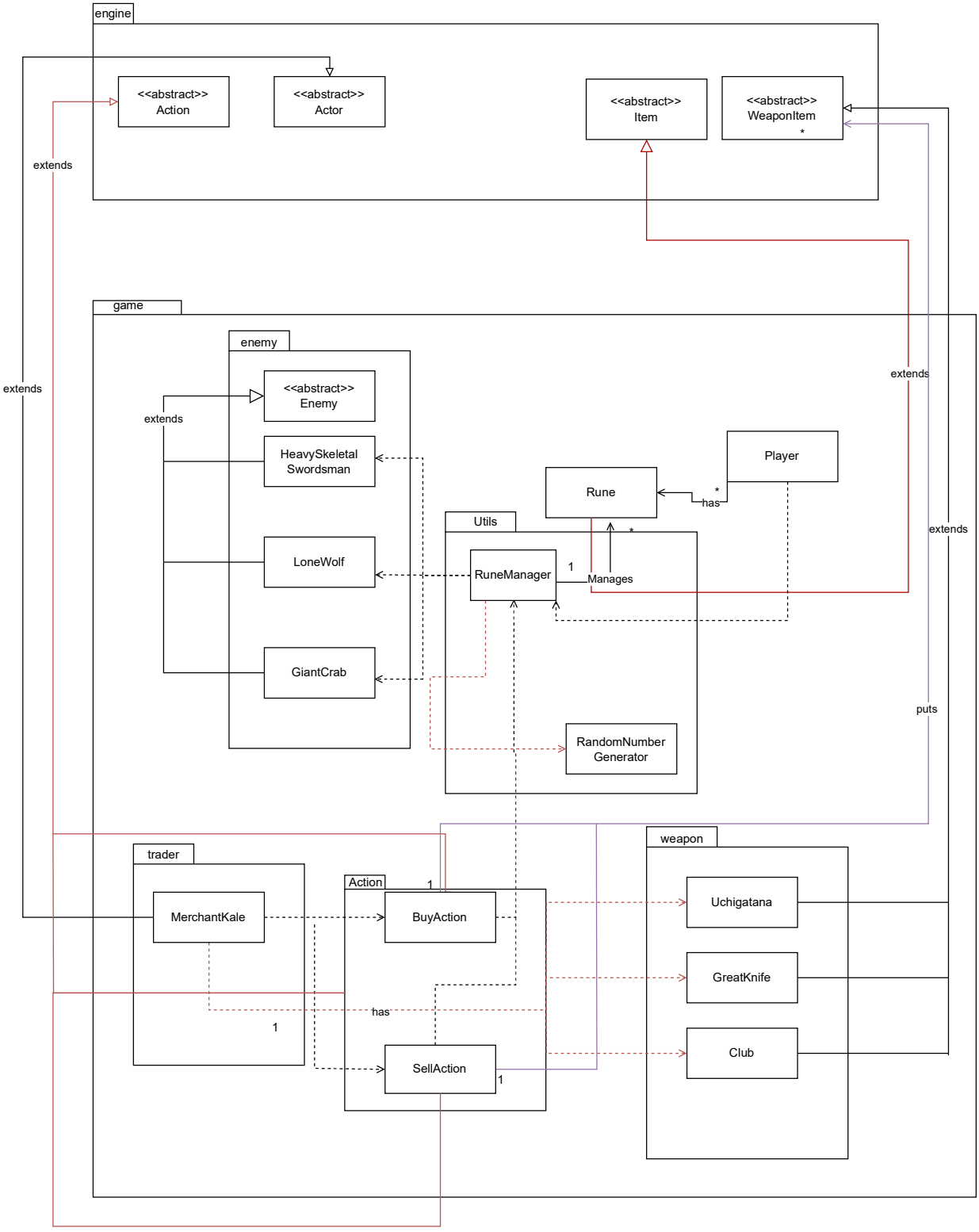
“PileOfBones” has a dependency to “ReviveBehaviour”. This allows “PileOfBones” to check the conditions for revival. By creating a behaviour for this, we allow other future enemies to use the same class for any revival.

C. Weapons

The weapons “Grossmesser” and “GiantCrabPincer” has a dependency to “AreaAttackAction”. This allows them to have a special skill, which is to attack their surroundings. Creating an class for this ensures easier extension, as any future weapons can call “AreaAttackAction” to implement an area attack easily. The two weapons also extends to a “WeaponItem” class, to allow them to share any common methods and not repeat methods (DRY principle).

This implementation does not have much cons, as it follows the Single Responsibility Principle (SRP) and DRY principle mostly.

REQ 2


engine
<<abstract>>
Action
<<abstract>>
Actor
<<abstract>>
Item
<<abstract>>
WeaponItem
*
extends
game
enemy
<<abstract>>
Enemy
HeavySkeletal
Swordsman
LoneWolf
GiantCrab
extends
extends
extends
extends
Player
Rune
*
has
Utils
RuneManager
1
Manages
*
RandomNumber
Generator
puts
trader
MerchantKale
1
Action
1
BuyAction
has
SellAction
1
weapon
Uchigatana
GreatKnife
Club

REQ 2

A. Enemies

Enemies are depended on by the “RuneManager” class in its “addRuneToPlayer” method. This method is used whenever an enemy dies based on the specifications. This follows the DRY principle as we do not have to repeat the same method for different enemies.

B. Trader

Our trader, “MerchantKale” class, extends the abstract class “Actor” from the provided engine. Having it extend this class allows us to easily insert actor into our map in application. A package called “Trader” is created in the game to group all related classes together, making for better organisation.

“MerchantKale” has a dependency to the weapons he is able to sell. In our requirements said weapons are “Uchigatana”, “Club” and “GreatKnife”. The advantage over using this instead of a sellable interface is that we are able to easily extend the amount of weapons and/or items Merchant Kale can sell by just adding it to his inventory.

“MerchantKale” has dependencies to “BuyAction” and “SellAction” as it is found in his method “allowableActions”, Which gives them the ability to sell weapons and buy weapons from the “Player”.

“BuyAction” and “Sell Action extend “Action”, which allows them to use or override any and all actions when necessary. Both Actions reside within the Action Package.

C. Runes

The “Player” inherits the “Rune” class as the player creates an instance of rune and adds the runes to their inventory.

Our “Rune” class is managed by “RuneManager”. It contains a simple attribute, constructor and methods. 1 Manager manages many runes. Which follows the SRP.

Our “RuneManager” class has a private instance meaning that there should only be one instance of it throughout the program. Its main function is to manage the “Rune” Class. It allows the runes to be added to the player, shown to the user, and subtracted from the player. It follows the Single Responsibility Principle(SRP) as We use to rune manager to prevent rune from becoming a god class that controls everything rune related.

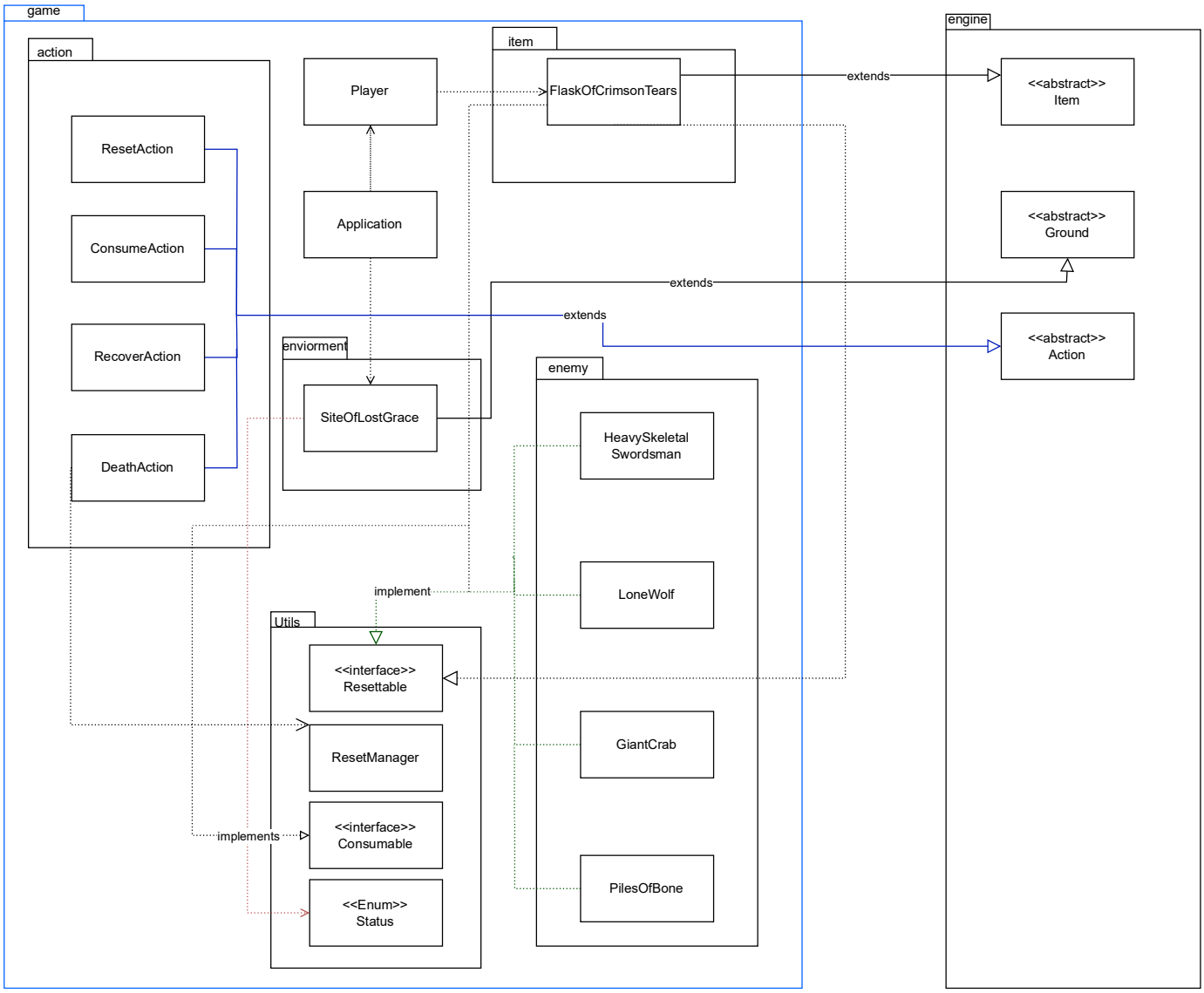
“RuneManager” has a hashmap that contains (key,value) pairs denoting (“Display Character”, (the lower bound of how many runes are dropable, the upper bound of how many runes are dropable). This is a very advantageous way of implementing this as it reduces the dependencies of having each monster carry runes and needing to check said runes everytime we kill a monster. This also allows us to easily extend our class whenever more monsters are added by putting new entries into our hashmap.

“RuneManager” has a dependency to the enemies “LoneWolf”,”GiantCrab” and “HeavySkeletalSwordsman” as they are used in the addRuneToPlayer method whenever necessary. This method is also dependant on the “RandomNumberGenerator” class to generate a random amount of runes based on a specific enemy. This method allows for the ease of transfer of runes from dead monsters to players.

“BuyAction” is also dependant on the RuneManager as it calls the “RuneManager” instance within its execute override method to help it manage how the runes flow from the “Players” inventory. Having a separate class from “SellAction” follows the SRP principle.

“SellAction” is also dependant on the RuneManager as it calls the “RuneManager” instance within its execute override method to help it manage how the runes flow into the “Players” inventory. Having a separate class from “BuyAction” follows the SRP principle.

REQ 3


game
action
ResetAction
ConsumeAction
RecoverAction
DeathAction
Player
Application
item
FlaskOfCrimsonTears
enviorment
SiteOfLostGrace
enemy
HeavySkeletal Swordsman
LoneWolf
GiantCrab
PilesOfBone
Utils
<<interface>> Resettable
ResetManager
<<interface>> Consumable
<<Enum>> Status
implement
implements
extends
extends
extends
engine
<<abstract>> Item
<<abstract>> Ground
<<abstract>> Action

REQ 3

A. Flask of Crimson Tears

Creating an “item” package to group all related items together improves organisation.

“FlaskOfCrimsonTears” is a special item that helps the player gain HP, hence it will have an extend relationship with “Item” abstract class from the engine package. Player will spawn with this item and it allows the player to consume twice. “FlaskOfCrimsonTears” will also be implementing resettable where after reset “count” get back to 2. The pros of doing such an implementation are that it is easily manageable and for future extensions, we can just make changes within the “FlaskOfCrimsonTears” class without touching other classes. The cons will be that if we have many potions in the future, such an approach without an abstract parent class can violate DRY rule.

B. Site of Lost Grace

“SiteOfLostGrace” is a unique ground, which it be extended from the abstract “Ground” class in the engine package. Since it is a ground type, it needs to be shown on the map for the player to rest on, The “Application” class will use it and place it onto the map, hence the dependency relationship. Our approach will be using a “Status” enum class to differentiate which actor can enter this particular ground. The pros of this approach are that such an implementation are that it is easily manageable and for future extensions, we can just make changes within the “SiteOfLostGrace” class without touching other classes. The cons will be in the future expansion, we might need a new abstract class to be the parents of these unique land so that we don't break DRY rule.

C. Game Reset

When a player is near “SiteOfLostGrace”, the player can choose to rest on it and the whole game will be reset. This is done by “SiteOfLostGrace” using the “ActorLocationsIterator” class to get the player's location and see if the player is on it. “SiteOfLostGrace” then can use the “RestAction” extended from the “Action” class in the engine to reset the game if the play decides to rest on this unique ground.

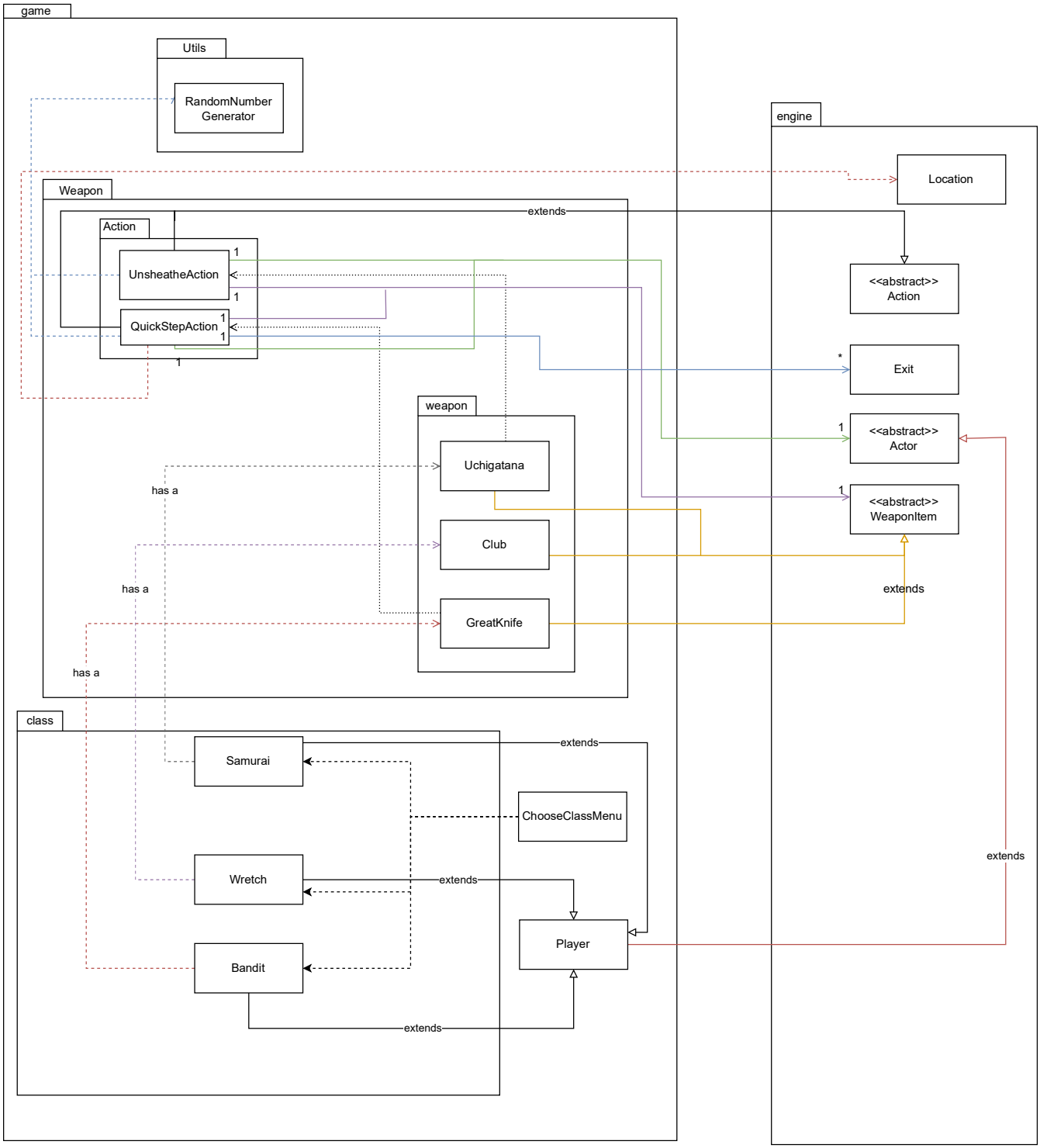
Game can be reset using two ways, when the player has died or when they chose to rest on the site of lost grace. All actors during this phase of the game development can be reset and hence all the enemies and players implemented the “Resettable” interface. And all the enemies that can be spawned will despawn. “HeavySkeletalSwordsman” and its dead state(piles of bones), “LoneWolf”, and “GiantCrab” will all implement the “Resettable” interface. The pros of this approach are that it removes many repeated codes if we were to write the despawn method in every enemies class, following SOLID and DRY principles.

“DeathAction” will be using the “ResetManager” to reset the game.

D. Runes

In our implementation, Runes will not need to have any relations with other classes.

REQ 4


game
Utils
RandomNumber Generator
Weapon
Action
UnsheatheAction
QuickStepAction
1
1
1
1
1
extends
weapon
Uchigatana
Club
GreatKnife
has a
has a
has a
class
Samurai
ChooseClassMenu
Wretch
Bandit
Player
extends
extends
extends
engine
Location
<<abstract>> Action
Exit
<<abstract>> Actor
<<abstract>> WeaponItem
*
1
1
extends
extends

REQ 4

A. Classes Combat Archetypes

Three classes are added to represent three different starting classes; “Samurai”, “Wretch” and “Bandit” classes. These classes are extended from the “Player” class, as they share common attributes therefore implying that we have successfully implemented the Don’t Repeat Yourself (DRY) rule. We can easily extend by making a new java class and just extend “Player” whenever we want to have new classes. The classes will be surrounded by a Class package for better organisation.

The User will select which class they would like to be at the start of the game via the menu in the console. This is done by the “ChooseClassMenu”. Depending on which class the user picks the menu will set the “Player” as a new instance of said class. This means that the “ChooseClassMenu” will have a dependency relationship to all 3 classes. This way of implementation follows the Single Role Principle (SRP) as we have a separate class for choosing roles using a menu. The “ChooseClassMenu” is located in the Utils class.

B. Weapons

Three Weapons, the “Uchigatana”, “Club” and “GreatKnife” are added to the “Weapon” package for better organisation. These weapon classes are depended on by their respective classes;

a. “Samurai” has a dependency to “Uchigatana”.
b. “Wretch” has a dependency to “Club”.
c. “Bandit” has a dependency to “GreatKnife”.

Each weapon extends the “WeaponItem” abstract class, so that they can share common methods (DRY rule) .

The “Uchigatana” has a special skill known as Unsheathe. Thus we have created an “UnsheatheAction” class in the “Action” package. The “Uchigatana” class has a dependency to the “UnsheatheAction” class. While the “UnsheatheAction” class extends the “Action” abstract class allowing it to share their common methods (DRY rule).

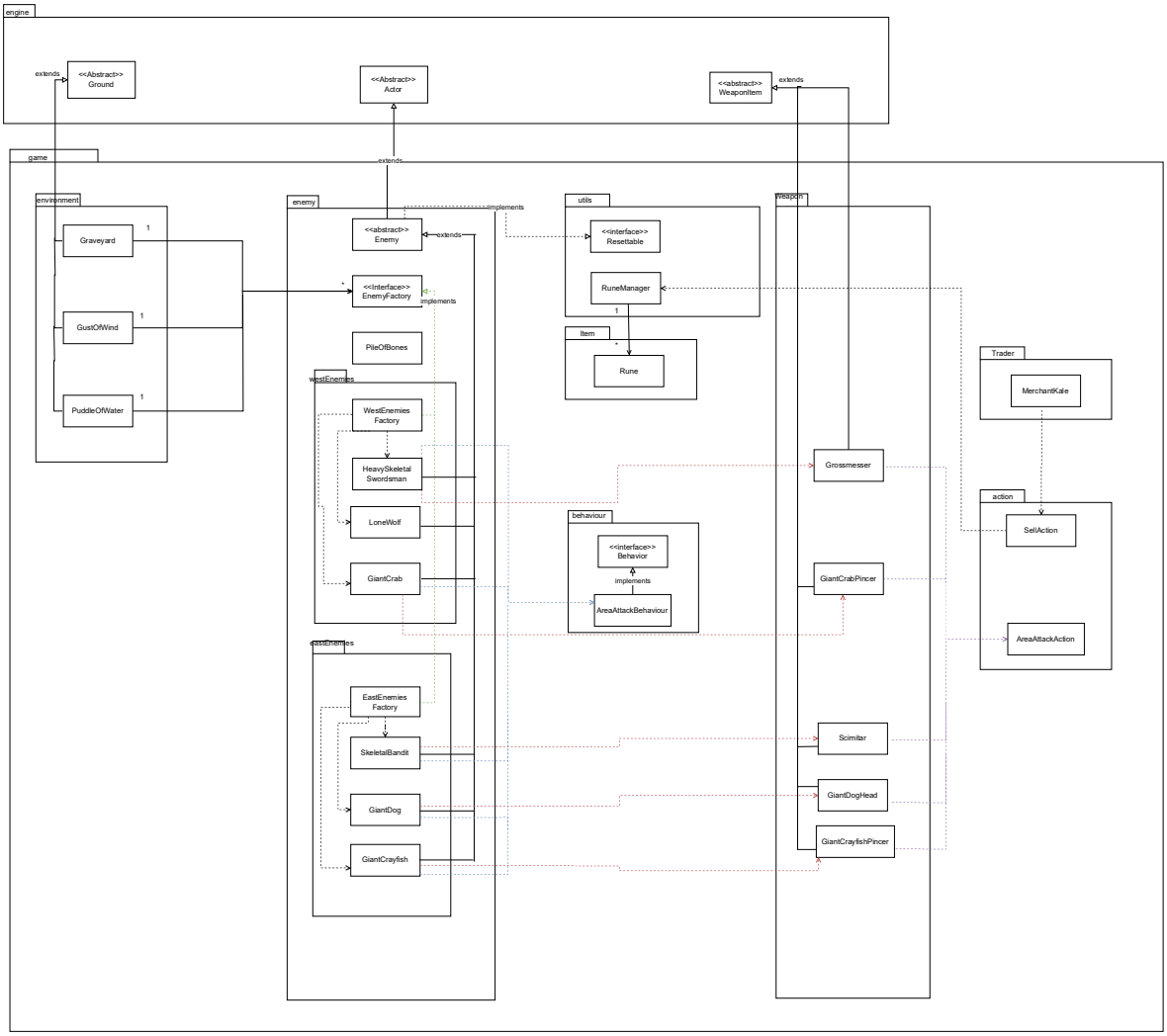
The “UnsheatheAction” class allows the user to do double damage of the weapon that wields the skill at a 60% hit rate. This is accomplished by

having an inheritance relationship with “Actor”, to know the target we are attacking, the “WeaponItem”, So we know which weapon is using the skill, and a dependency relationship to “RandomNumberGenerator” so we know if the attack has missed or not.

The “GreatKnife” has a special skill known as Quick Step. Thus we have create a “QuickStepAction” class in the “Action” package. The “GreatKnife” class has a dependency to the “QuickStepAction” class. While the “QuickStepAction” class extends the “Action” abstract class allowing it to share their common methods (DRY rule).

The “QuickStepAction” class allows the user to do the same damage as the weapon that wields the skill at the same rate. However, It also allows the wielder to immediately move to an adjacent location right after the attack. This is accomplished by having an inheritance relationship with “Actor”, to know the target we are attacking and the player we are moving, the “WeaponItem”, so we know which item will be using the skill, the “Location”, so we know where the player is currently at, the “Exit”, so we know the possible exits that the player may move and if there is an actor in an adjacent location, and finally a dependency relationship to “RandomNumberGenerator”, so we know if the attack has missed or not. (Note: the player will move regardless if the attack has hit)

REQ 5


engine
<<Abstract>>
Ground
<<Abstract>>
Actor
<<abstract>>
WeaponItem
extends
game
environment
Graveyard
GustOfWind
PuddleOfWater
enemy
<<abstract>>
Enemy
<<Interface>>
EnemyFactory
PileOfBones
westEnemies
WestEnemies
Factory
HeavySkeletal
Swordsman
LoneWolf
GiantCrab
eastEnemies
EastEnemies
Factory
SkeletalBandit
GiantDog
GiantCrayfish
implements
utils
<<interface>>
Resettable
RuneManager
item
Rune
behaviour
<<interface>>
Behavior
AreaAttackBehaviour
weapon
Grossmesser
GiantCrabPincer
Scimitar
GiantDogHead
GiantCrayfishPincer
Trader
MerchantKale
action
SellAction
AreaAttackAction

Req 5

A. Enemies

Our main “Enemy” abstract class implements Resettable as all enemies within the scope of our current requirements are able to be reset. All enemies are actors therefore the “Enemy” class also extends the abstract “Actor” class in the engine.

In this requirement, we are introduced to three new enemy classes; “SkeletalBandit”, “GiantDog”, and “GiantCrayfish”. They are extended by the “Enemy” class along with the enemies from the previous requirements. “LoneWolf”, “GiantCrab” and “HeavySkeletalSoldier” classes. As they are part of enemies. Since we are extending an abstract class we are able to use all the common methods meaning DRY principle is followed. The enemies are spawned in from their respective Grounds which have an inheritance to the interface “EnemyFactory” class in the enemy package, Following the Single Role Principle(SRP).

The values of the runes that are dropped by the enemies are stored in the “Rune” class in the item package and managed by the “RuneManager” class. This follows the SRP as we dont make Rune into a god class. Which also makes it easier to extend as we are able to just add more information into our manager class.

“SkeletalBandit”, “GiantDog”, “GiantCrayfish”, “GiantCrab”, and “HeavySkeletalSwordsman” all have a dependency relationship to “AreaAttackBehaviour” which implements the interface “Behaviour”. This follows the Dependancy inversion principle (DIP).

Something new that was introduced was the West and East Region. Therefore in the enemy package we have created 2 other packages “WestEnemiesFactory” and “EastEnemiesFactory”. Both factories implement the interface “EnemyFactory” class. This follows the SRP.

B. Weapons

All weapons are weapon items therefore all weapons extend the “WeaponItem” abstract class in the engine allowing them to share common methods and therefore follow the DRY principle.

Including the previous requirements we have 5 weapons depended only enemies, those are;

a. “HeavySkeletalSwordsman” depending on “Grossmesser”
b. “GiantCrab” depending on “GiantCrabPincer”
c. “SkeletalBandit” depending on “Scimitar”
d. “GiantDog” depending on “GiantDogHead”
e. “GiantCrayfish” depending on “GiantCrayfishPincer”

Within this requirement, all weapons depended on by enemies, “GiantCrayfishPincer”, “GiantDogHead”, “Scimitar”, “GiantCrabPincer”, “Grossmesser”, all are able to attack surrounding actors. Therefore, they all have a dependency to the “AreaAttackAction” class in the “Action” package. This follows the DRY principle as we do not make each weapon have the implementation within their respective classes.

Finally, Some weapons , namely the “Scimitar” and “Grossmesser” can be sold. The information about the weapons and their rune price are all stored in the “RuneManager” which is depended on by “SellAction” class. “MerchantKale” has a dependency to the “SellAction” making him able to be sold to. Advantageously,
This allows for ease of extension as we are able to input new information easily into the “RuneManager” as they come. This follows the SRP as we do not make each class too heavy with functions.