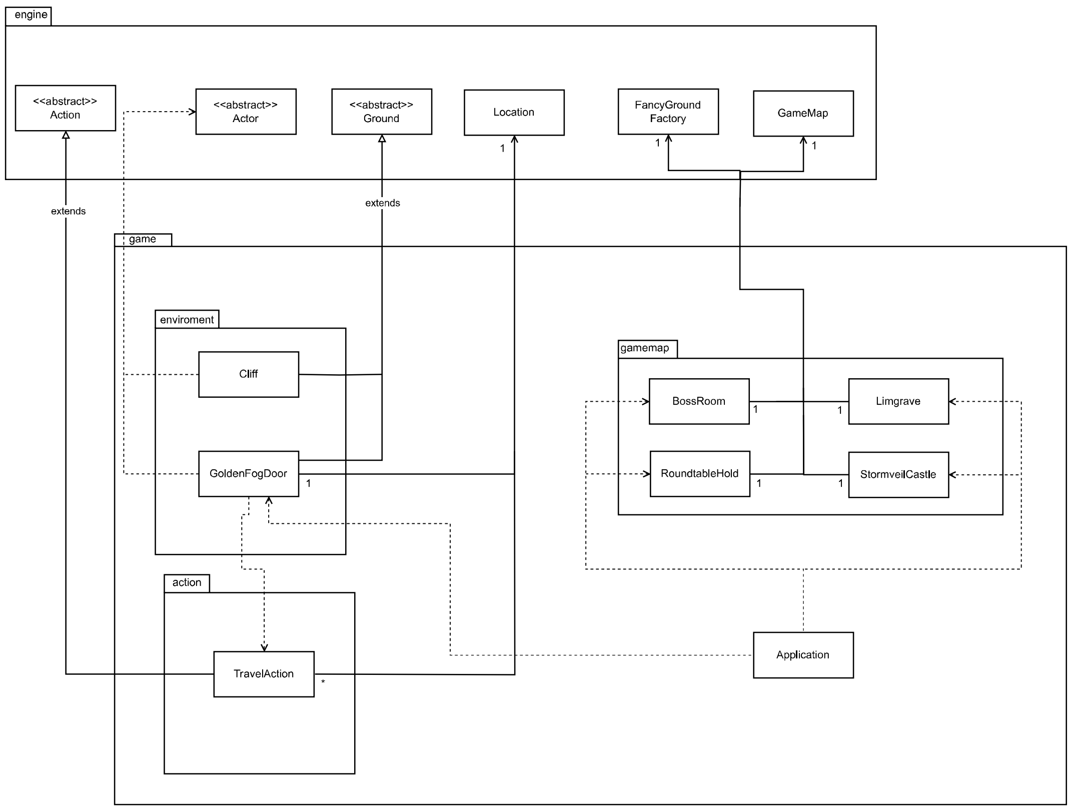
REQ1


engine
<<abstract>> Action
<<abstract>> Actor
<<abstract>> Ground
Location
FancyGround Factory
GameMap
extends
extends
1
1
1
game
enviroment
Cliff
GoldenFogDoor
1
action
TravelAction
*
gamemap
BossRoom
Limgrave
RoundtableHold
StormveilCastle
1
1
1
1
Application

REQ 1

A. New Grounds

The classes “Cliff” and “GoldenFogDoor” are added into the environment package created in assignment 2, for better organisation.

“Cliff” and “GoldenFogDoor” represents different grounds. They are extended from the abstract class “Ground”, so that similar methods can be grouped together (DRY principle). Having this abstract class also allows easier expansion in the future, as shown when extending this task from assignment 2. These two classes also have a dependency with the “Actor” class, as they need to check which actor is able to step on them.

“GoldenFogDoor” has an association with the “Location” class. This is because “GoldenFogDoor” needs to save a location, so that the player can travel to that location. It has a one-to-one relationship, as one Golden Fog Door can only have one location saved up. It also has a dependency with “TravelAction”, as it allows the player to select the travel action, when a player is on top of this ground. The class “Application” has a dependency with “GoldenFogDoor”, as the golden fog door can only be placed in the map after all gamemaps have been inserted into the world. Without inserting the gamemaps, there is no way to get the location. Thus, I have inserted all golden fog doors after inserting the gamemaps in application.

The class “TravelAction” is in the action package, for better organization. “TravelAction” is extended from “Action”. This allows similar methods to be grouped together (DRY method), and allows for easier expansion of adding new action classes. “TravelAction” also has an association with “Location”. This is because TravelAction needs to know where the player is travelling to (i.e. where to send the player to). It has a many-to-one relationship, as one TravelAction only allows the player to go to one location, but one location can have many TravelActions on it.

B. Game Maps

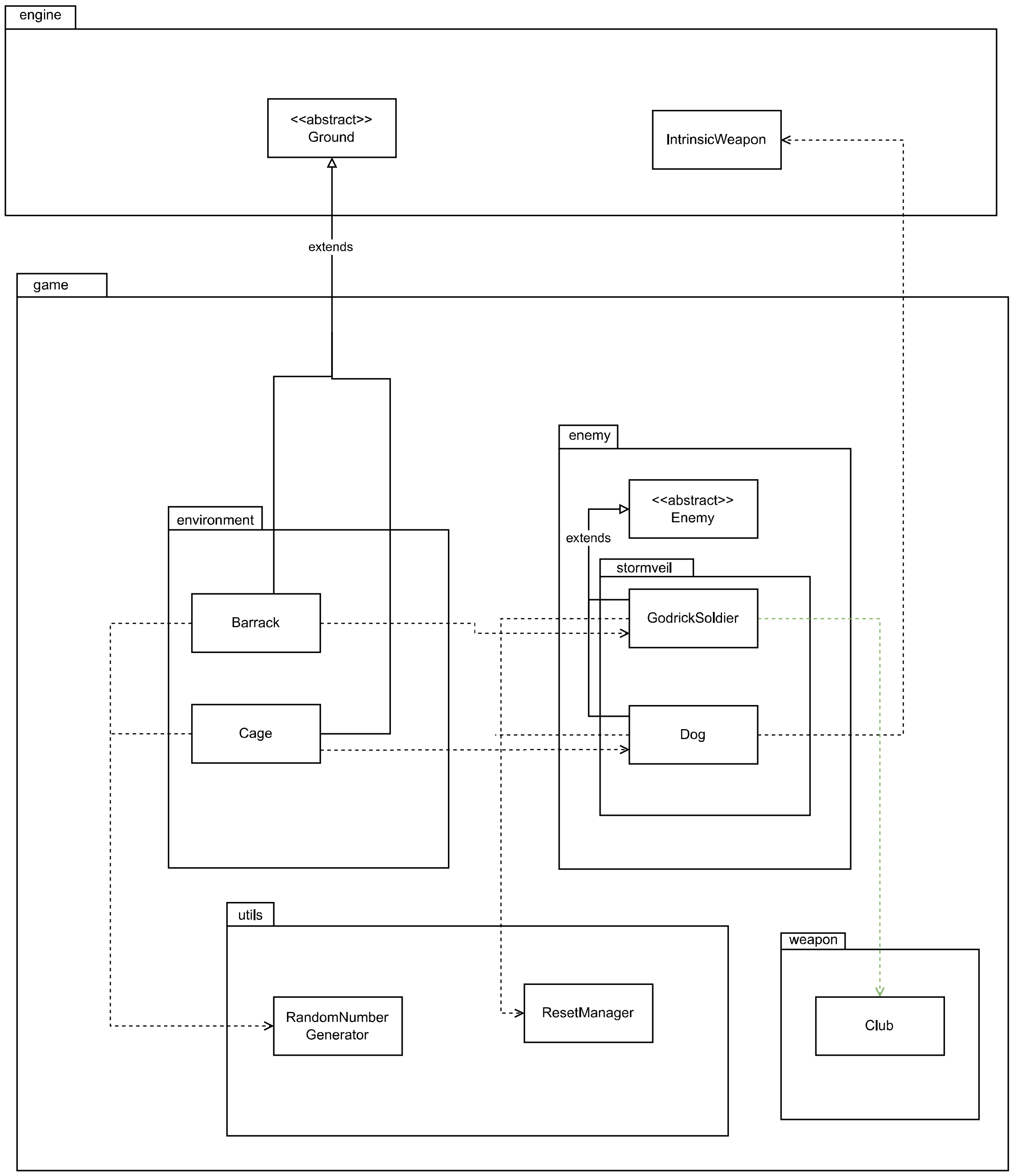
Since we now have 4 gamemaps, it is infeasible to create all of them in Application, as this will make application a superclass, breaking the Single Responsibility Principle. Thus, I have created a class for each gamemap, which are

“BossRoom”, “Limgrave”, “RoundtableHold”, and “StormveilCastle”. The “Application” class will call these 4 classes to be inserted into the world, thus there is a dependency. These 4 classes have an association with “FancyGroundFactory”. This is because the FancyGroundFactory will go through the list of arrays, and determine what type of ground is at each location. There is a one-to-one relationship, as one map can only have one FancyGroundFactory.

The 4 classes also have an association with “GameMap” This is so that a new Gamemap can be created for each class. This means that when the class is called, a new gamemap of this class will immediately be generated. There is a one-to-one relationship, as one map can only create one new GameMap.

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

REQ 2


engine
<<abstract>>
Ground
IntrinsicWeapon
extends
game
enemy
<<abstract>>
Enemy
extends
stormveil
GodrickSoldier
Dog
environment
Barrack
Cage
utils
RandomNumber
Generator
ResetManager
weapon
Club

REQ 2

A. Grounds

The new classes “Barrack” and “Cage” are added to the environment package, for better organisation.

“Barrack” and “Cage” represents different grounds. They are extended from the abstract class “Ground”, so that similar methods can be grouped together (DRY principle). Having this abstract class also allows easier expansion in the future. These two classes have a dependency with the class “RandomNumberGenerator”, to use its method to generate a random percentage, so that they can calculate whether to spawn an enemy or not. “Barrack” has a dependency with “GodrickSoldier”, while “Cage” has a dependency with “Dog”. These dependencies are there, so that the relevelent grounds can spawn enemies.

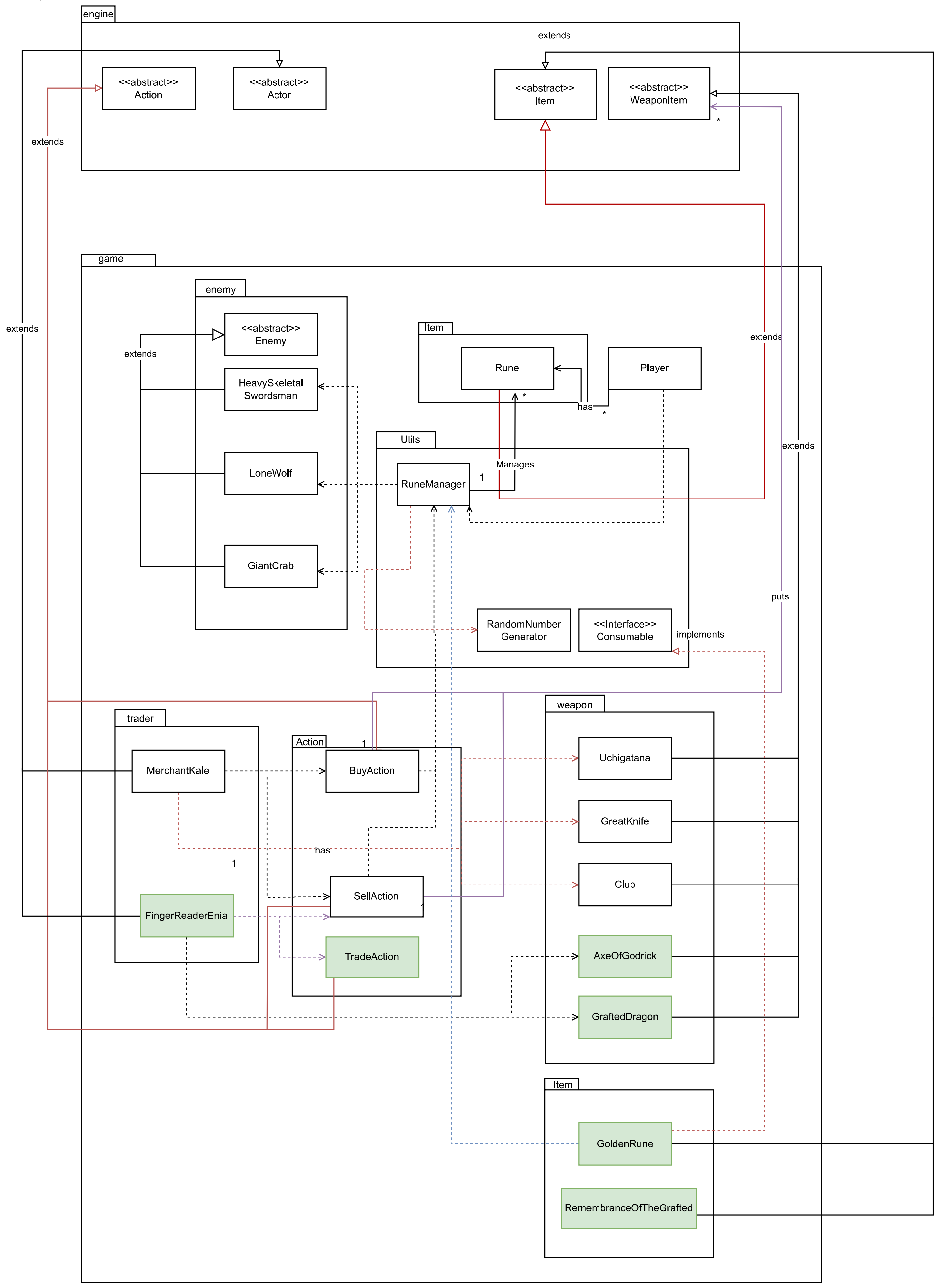
B. Enemies

There are two new enemy classes, “Dog” and “GodrickSoldier”. They are added to the package “stormveil” for better organisation, as both of them live in stormveil castle. These two classes are extended from the abstract class “Actor”, so that similar methods can be grouped together (DRY principle). Having an abstract class also allows easier expansion in the future. “Dog” has a dependency with “IntrinsicWeapon”, to allow Dog to attack other actors. Meanwhile, “GodrickSoldier” has a dependency with “Club”, to allow Godrick Soldier to attack other actors. Since the Heavy Crossbow weapon is optional, I have modified it so that Godrick Soldier carries a Club instead.

“GodrickSoldier” and “Dog” both have a dependency with “ResetManager”. This is so that they will get wiped off the map when the game resets. ResetManager is created specially to manage the resetting of the game, so that it follows the Single Responsibility Principle.

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

REQ 3


engine
extends
<<abstract>> Action
<<abstract>> Actor
<<abstract>> Item
<<abstract>> WeaponItem
*
extends
game
enemy
<<abstract>> Enemy
extends
extends
HeavySkeletal Swordsman
LoneWolf
GiantCrab
Item
Rune
*
has
*
Player
extends
Utils
RuneManager
Manages
1
extends
RandomNumber Generator
<<Interface>> Consumable
implements
puts
trader
MerchantKale
FingerReaderEnia
1
has
Action
BuyAction
1
SellAction
TradeAction
weapon
Uchigatana
GreatKnife
Club
AxeOfGodrick
GraftedDragon
Item
GoldenRune
RemembranceOfTheGrafted

REQ 3

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.

We now have a new Trader, “FingerReaderEnia”, whom 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.

“FingerReaderEnia” has a dependency to the weapons he is able to sell. In our requirements said weapons are “GraftedDragon” and “AxeOfGodrick”. 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 “TradeAction” and “SellAction” as it is found in his method “allowableActions”, Which gives them the ability to sell weapons and buy weapons from the “Player”.

The new “TradeAction” extends “Action” in the engine, which allows it to use any override methods when necessary. Following the SRP rule.

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.

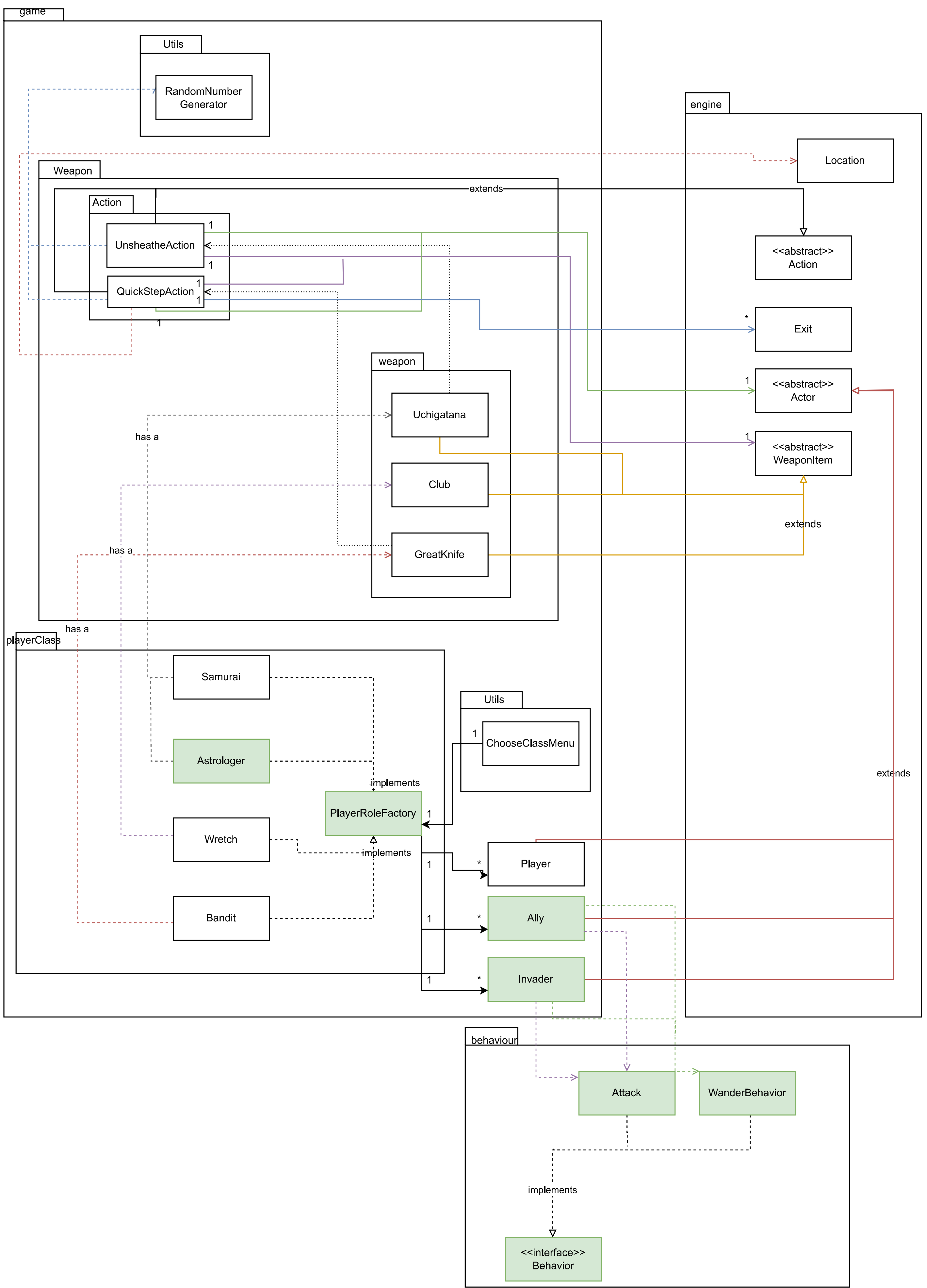
“SellAction” is also dependant on the RuneManager as it calls the “RuneManager” instance within its execute override method to help it manage how the runes flow into the “Players” inventory. Having a separate class from “BuyAction” follows the SRP principle.

A new item “GoldenRune” has been introduced in assignment 3, It extends “Item” abstract class in the engine and implements “Consumable” in the utils package. This allows us to use any methods within the “Item” abstract class and allow the player to consume the golden rune.

When a golden rune is consumed a random number of runes is generated using “Random” and is added to the players inventory. This means that the “GoldenRune” class would have to have a dependency to the “RuneManager”

We have added the Item, “RemembranceOfTheGrafted” and the weapons, “AxeOfGodrick” and “GraftedDragon” as placeholders as we are not planning on implementing the optional tasks.

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
playerClass
Samurai
Astrologer
Wretch
Bandit
PlayerRoleFactory
implements
implements
Utils
ChooseClassMenu
1
1
1
1
1
Player
Ally
Invader
*
*
*
engine
Location
<<abstract>> Action
Exit
*
<<abstract>> Actor
1
<<abstract>> WeaponItem
1
extends
extends
behaviour
Attack
WanderBehavior
implements
<<interface>> Behavior

REQ 4

A. Classes Combat Archetypes

“PlayerRoleFactory” is a new interface we have made in requirement 3 to make the classes classes be used by allies and invaders. This allows us to easily create or give classes to our respective actors. (We have removed the extension from respective class classes to accommodate this new feature.)

Four classes are added to represent three different starting classes; “Samurai”, “Wretch”, “Bandit” and “classes”. The classes will be surrounded by a Class package for better organisation.

These classes implement the “PlayerRoleFactory” class therefore implying that we have successfully implemented the Don’t Repeat Yourself (DRY) rule. We can easily extend the amount of classes by just having a new class for said class to implement PlayerRoleFactory to adopt its methods.

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 invoke the “chosePlayerRole” override method that resides within the respective classes. This means that the “ChooseClassMenu” will have a dependency to the “PlayerRoleFactory”. 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.

The downside of this approach is that we have to manually create new classes and it is not automated. This can be proven tedious and problematic as it will be hard to keep track of, for example, 1000 classes.

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” and “Astrologer” have 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).

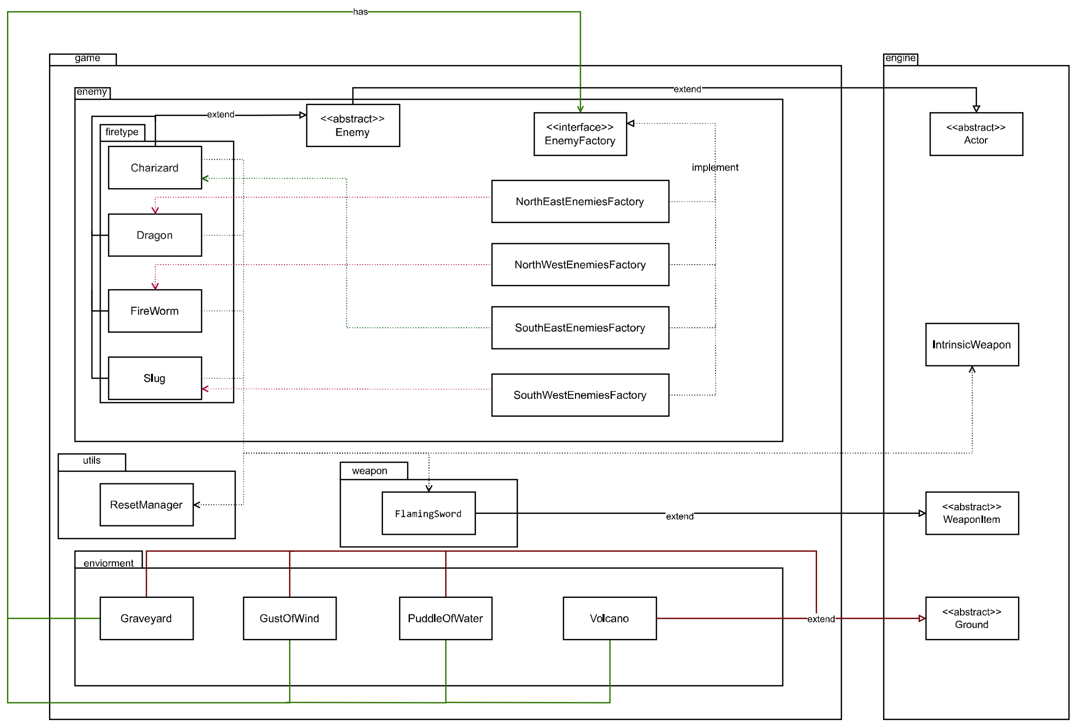
The “UnsheatheAction” class allows the user to do double damage of the weapon that wields the skill at a 60% hit rate. This is accomplished by having an inheritance relationship with “Actor”, to know the target we are attacking, the “WeaponItem”, So we know which weapon is using the skill, and a dependency 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)

The downside of this approach is that we have to manually create new Weapons and skills is not automated. This can be proven tedious and problematic as it will be hard to keep track of, for example, 1000 weapons with 1000 different skills.

REQ 5


has
game
engine
enemy
extend
<<abstract>>
Enemy
<<interface>>
EnemyFactory
<<abstract>>
Actor
firetype
extend
Charizard
implement
NorthEastEnemiesFactory
Dragon
NorthWestEnemiesFactory
FireWorm
SouthEastEnemiesFactory
IntrinsicWeapon
Slug
SouthWestEnemiesFactory
utils
weapon
ResetManager
FlamingSword
extend
<<abstract>>
WeaponItem
enviorment
Graveyard
GustOfWind
PuddleOfWater
Volcano
extend
<<abstract>>
Ground

REQ 5

A. New sides of map
The map is now being separated into 4 sides, with top left being labeled as “north west”, top right being labeled as “north east”, bottom left being labeled as “south west” and bottom right being labeled as “south east”. Each side of the map will have their respective EnemiesFactory, which will generate different type of enemies based on the location of the ground in that specific map. The “NorthEastEnemiesFactory”, “NorthWestEnemiesFactory”, “SouthEastEnemiesFactory” and “SouthWestEnemiesFactory” will all implementing the “EnemyFactory” interface. The advantages of doing so is that it is easily manageable and making further changes fast and trobleless. We can just add in new methods inside the “EnemyFactory” class and implementing the method inside each sides of the enemies facotry concrete classes since each location of ground will spawn different type of enemies. By doing so it obey the SOLID priciple.

B. New enemies
There will be 4 new enemies with “Fire” type and are located in the “firetype” package. They are “Charizard”, “Dragon”, “FireWorm” and “Slug”. The 4 different types of enemies will spawn respectively with the location of the map, north, east, south and west. Each of the enemies classes will extend the abstract Enemy class, by doing so we follow the DRY rule. The pros of this approach is that when we want to make adjustments to the enemies in the future, we will not have to go into each enemies concrete classes to make changes, only changes will be made in the Enemy abstract class. The 4 enemies will also uses “ResetManager” and “IntrisicWeapon” to help despawn and reset everything once the player is dead or resting and to get thier default weapon.

C. New ground
A new ground called “Volcano” is made to spawn all the “Fire” type enemies. Just like all the other ground classes, it extend the abstract “Ground” class from the engine and uses the “EnemyFacotry” to spawn the respective enemies for different sides of the map. All the previous ground class has also been update to use the 4 new enemies factory class for the 4 different sides. The pros of such approach is that we do not need to make changes in the ground class if we wanted to make adjustments on which enemies is being spawn at which side, all the adjustments is being done in the enemies factory classes and hence following SOLID principles.

D. New weapon
A new weapon called “FlamingSword” is beening made for the “Fire” type enemies. Where it directly extends from the abstract “WeaponItem” class in the engine package and is being use by all the Fire type enemies via a dependency relationship. When the “Fire” type enemies die they will drop the “FlamingSword” onto the ground and

player can choose to pick it up. By resuing the “WeaponItem” class, we avoid having to create many same methods, following the DRY rule.