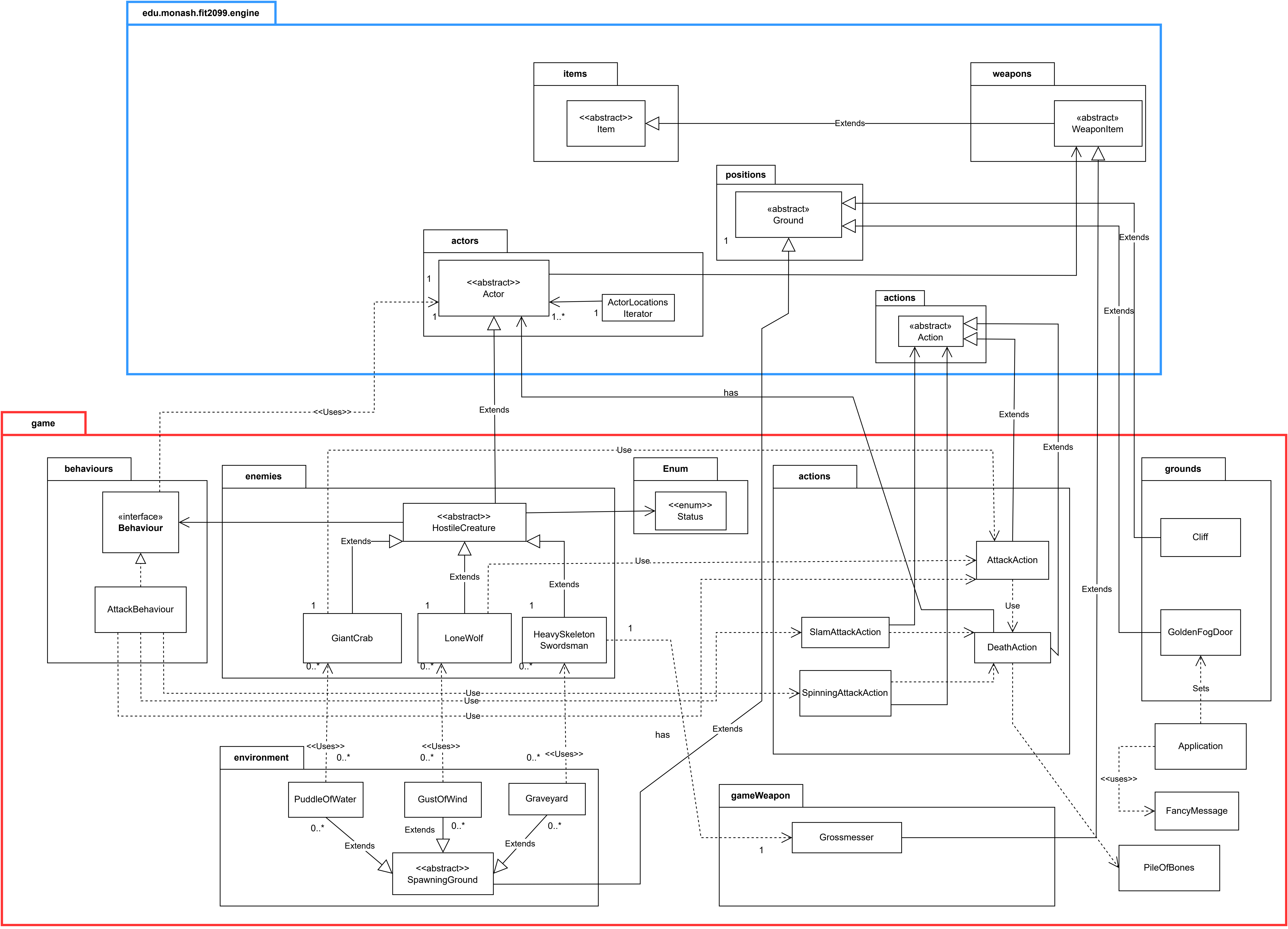

edu.monash.fit2099.engine
items
<<abstract>> Item
weapons
«abstract» WeaponItem
Extends
positions
«abstract» Ground
actors
<<abstract>> Actor
ActorLocations Iterator
actions
«abstract» Action
1
1..*
has
<<Uses>>
Use
game
behaviours
«interface» Behaviour
AttackBehaviour
enemies
<<abstract>> HostileCreature
GiantCrab
LoneWolf
HeavySkeleton Swordsman
0..*
Enum
<<enum>> Status
actions
AttackAction
SlamAttackAction
SpinningAttackAction
DeathAction
grounds
Cliff
GoldenFogDoor
Sets
Application
<<uses>>
FancyMessage
PileOfBones
environment
PuddleOfWater
GustOfWind
Graveyard
<<abstract>> SpawningGround
gameWeapon
Grossmesser

***Changes are below*
***Assignment 3 REQ extension details below*

A. Environment

Three different types of environments (graveyard, gust of wind and puddle water) are concrete classes that classified as spawning grounds. These spawning grounds share common attributes to Dirt & Floor classes where each dictates allowable actions apart from displaying their character on game map. In addition to these functions, spawning grounds have ability to create specific enemy based on type.

SpawningGround extends Ground
SpawningGround---<<uses>>--->Utils
SpawningGround inherits Ground. Spawning ground shares the common similar methods of displaying the ground character and set allowable actions. Abstraction of Ground allows for environment to be generated dynamically based on enemy spawn. Further abstraction of spawning ground gives open extension for different spawning grounds of different enemies. In spawning grounds, there are methods for despawn and spawn. These are calculated in using a static probability calculator in Utils. A dependency relationship is formed through calling an instance of Utils.

Graveyard extends SpawningGround,
GustOfWind extends SpawningGround,
PuddleOfWater extends SpawningGround
Three different spawning grounds inherit two additional common methods that every spawning ground contains, which are despawning and spawning. Calculation of these actions is based on probability on each turn and can accept various inputs of probability. Reusability of the methods adheres to DRY principle.

Graveyard---<<contains>>--->HeavySkeletonSwordsman,
GustOfWind---<<contains>>--->LoneWolf,
PuddleOfWater---<<contains>>--->GiantCrab
Each spawning ground spawns one kind of enemy. Every time an environment is called, the environment will generate an enemy object based on a probability.
An enemy object (whether null or exists) will be stored in the environment class as attribute permanently to keep track on its existence. This creates an association relationship between enemy creation and its place of creation highlights the sole purpose of breeding enemies (Single responsibility principle).

B. Enemies

Three different enemies are presented in the game as non-playable character (NPC). They exist as an Actor with hit points and items inside their inventory. The non-playable state does not restrict its motion as they contain interactable behaviours and capabilities.

HostileCreature extends Actor
HostileCreature inherit Actor class as the three different hostile creatures have identical methods and attributes (just not playable by user). The core state of Actor gives HostileCreature class full necessary adaptation (Liskov's Substitution Principle) for extending itself to creatures with specific behaviours and capabilities.

HostileCreature---<<has>>--->Status,
HeavySkeletonSwordsman---<<uses>>--->Status,
LoneWolf---<<uses>>--->Status,
GiantCrab---<<uses>>--->Status
Hostile creature can attack other hostile creature but not of their own kind (except Giant Crab and Heavy Skeleton Swordsman). Status enumeration helps to identify actors that are hostile to enemies. The status is kept as an attribute of abstract class so other type of hostile creatures can have access to its status when called for comparison in individual classes. Enumeration class helps to prevent magic strings error since status type can be represented as String. To further distinguish attack on hostile creatures' type, other attribute of enemy (display character) would be used for identification. An alternative of creating a new enumeration class of attack type would made subclasses tight on coupling while not utilising enemies' common attributes as identification.

HeavySkeletonSwordsman extends HostileCreature,
LoneWolf extends HostileCreature,
GiantCrab extends HostileCreature
Three types of hostile creatures inherit HostileCreature class since they share attributes commonly exist among hostile creatures.

HeavySkeletonSwordsman---<<has>>---> Behaviour,
LoneWolf---<<has>>--->Behaviour,
GiantCrab---<<has>>---> Behaviour,
WanderBehaviour---<<implements>>--->Behaviour,
FollowBehaviour---<<implements>>--->Behaviour
Hostile creatures have three behaviours implemented in their classes as attributes. These behaviours act as collection of action that cannot be chosen by the user. Directly, these behaviours are part of the creatures and are associated. Interface of Behaviour is chosen as attribute class as it can take in three other behaviours. Two separate behaviours implement the method of getAction, where in their own classes, the method is tailored to the structure of motion. Follow behaviour gives enemies action to follow actors within their bound and Wander behaviour gives enemies to move on itself.

LoneWolf---<<uses>>--->AttackAction,
GiantCrab---<<uses>>--->AttackAction,
GiantCrab---<<uses>>--->SlamAttackAction,
SlamAttackAction---<<uses>>--->AttackAction
Giant Crab can carry out attack on a sole target and area attack but only sole target attack for Lone Wolf. For a sole target attack, AttackAction is called as instance through allowable

actions for later play turn. Creating an instance would be a dependency relationship. This occurs to Lone Wolf and Giant Crab. However, Giant Crab has a unique area attack that slams any actors around its surroundings. Hence, a SlamAttackAction instance would be called in allowable actions for later use. The SlamAttackAction would consist of multiple of sole target attack, meaning multiple of AttackAction. SlamAttackAction can be made anew with new attack methods instead of utilising AttackAction but elements needed for attack would be similar to AttackAction and it will be repetitive (DRY principle). One drawbrack would be memory use of object call. If there are little enemies, reusing the codes would benefit than need for optimization.

HeavySkeletonSwordsman---<<uses>>--->PileOfBones
Heavy skeleton swordsman can turn into pile of bones once it gets killed. Pile of bones can revive back to swordsman if not hit successfully in three turns. In this case, piles of bones cannot be considered as a standalone Actor fully but as extension of Heavy Skeleton Swordsman. Heavy skeleton swordsman would create an instance of piles of bones as a temporary object to count the number of hits and proceed to death action or revival depending on successful hits. The use of dependency relation is sufficient in keeping track of progress.

ActorLocationsIterator---<<uses>>--->Actor
Enemies can be despawned at each turn and so removing a hostile creature from a location on a map can be done through remove method.

AttackAction extends Action,
DeathAction extends Action,
AttackAction---<<uses>>--->DeathAction,
AttackAction---<<has>>--->Actor,
DeathAction---<<has>>--->Actor,
AttackAction---<<has>>--->Weapon
Hostile creatures have to abilities to attack other enemies with certain conditions. However, the procedure of attack remains similar for all enemies regardless. Attack and death action inherits Action class as Action class provides needed abstract methods for definition of attack (Dependency Inversion Principle). Attack action needs actor as target and weapon of choice for combat, hence the need of keeping these as attributes for accessing the target counterattack behaviour and weapons attack behaviour (e.g., hit points). Hence, they have association relationship. Once a targeted hostile creature has been defeated after an attack, an instance of Death Action would be called. Death action only requires dependency relationship to perform any item drop from target and updates target existence in the game map.

C. Weapons

HeavySkeletonSwordsman---<<has>>--->Grossmesser,
WeaponItem---<<implements>>--->Weapon,
Grossmesser extends WeaponItem,
WeaponItem extends Item

Heavy skeleton swordsman carries a weapon item called Grossmesser. This weapon becomes an attribute for the swordsman with association relationship. Grosmesser needs methods and attributes in Weapon Item abstract class to be classified as one (Dependency Inversion Principle). Therefore, Grossmesser inherits Weapon Item. Weapon Item shares similar methods in Weapon interface (Weapon Item implements Weapon) and extends itself as Item that an Actor can have.

HeavySkeletonSwordsman---<<uses>>--->AttackAction,
AttackAction---<<uses>>--->Grossmesser,
HeavySkeletonSwordsman---<<uses>>--->SpinningAttackAction,
SpinningAttackAction---<<uses>>--->AttackAction,
Grosmesser is a weapon used by HeavySkeletonSwordsman for attack. It can be used for sole target attack or area attack. For sole target attack, HeavySkeletonSwordsman would call an instance of AttackAction with Grossmesser object, and it establishes a dependency relationship for the attack and the use of Grosmesser weapon object. Alternatively, HeavySkeletonSwordsman may perform a SpinningAttackAction with Grossmesser. SpinningAttackAction would be created as an object in action list and utilises multiple call of AttackAction with Grossmesser as an object parameter. Like SlamAttackAction, we achieve DRY principle through reusing the codes from AttackAction. Same drawback goes to larger use of memory space if many enemies exist around.

Changes to REQ1

The majority of the existing classes in engine illustrated in the UML diagram has been removed (includes the capabilities package, dropWeapon and dropItem Action, Weapon Interface and Instrinsic Weapon classes). The big change to this design would include Attack Behaviour class to facilitate attack action carried out by NPC hostile creatures to their enemies and proper implementation of Pile of Bones that was missing last time.

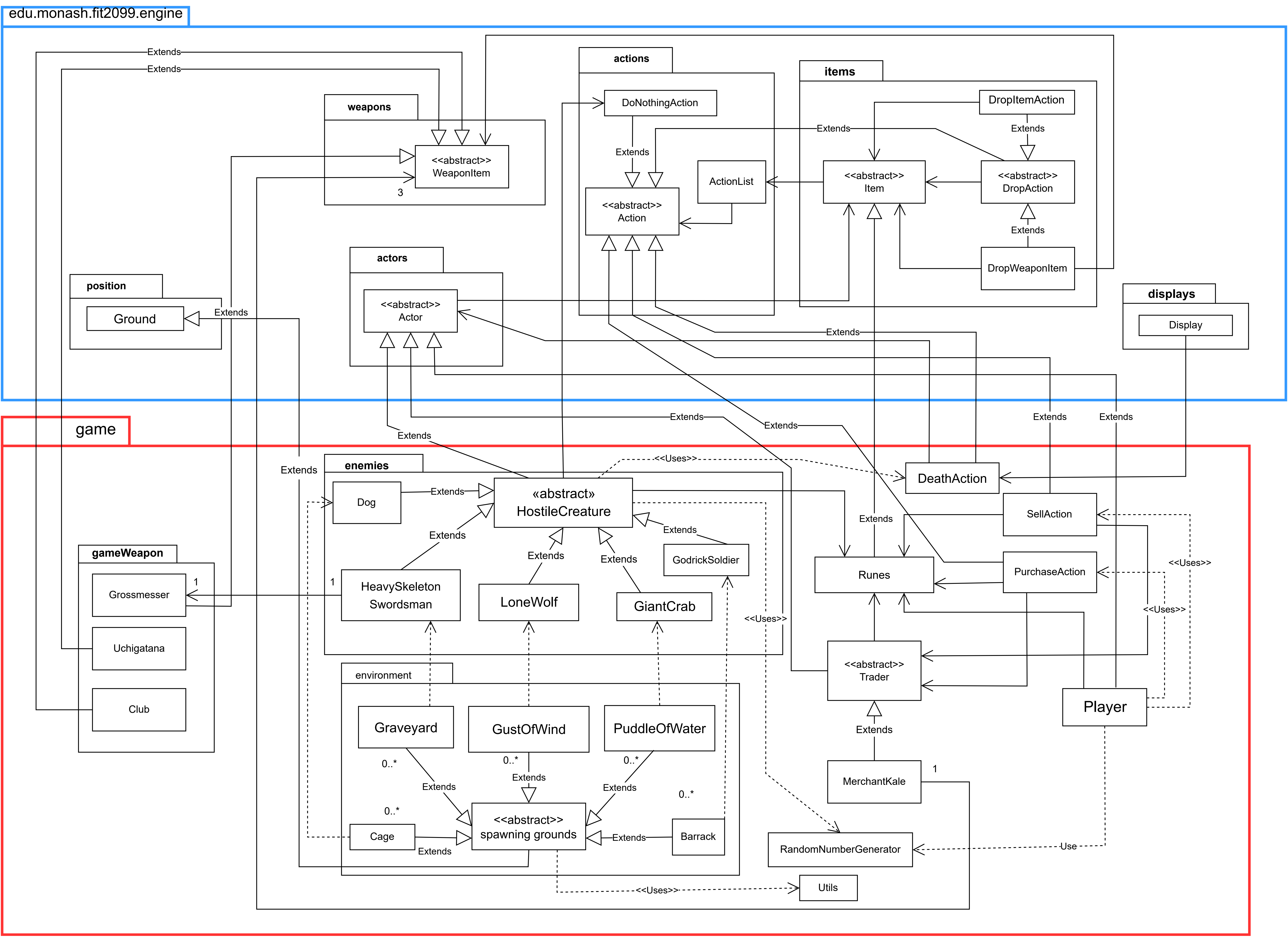
Some dependency/relationship arrows have been removed due to design change or new classes that made the previous relationship redundant.

AttackBehaviour---<<uses>>--->AttackAction,SpinningAttackAction,SlamAttackAction
After a better understanding of engine, it is decided to dedicate a separate method in interacting NPC hostile creature attack action. Player would carry out attacks in action list while NPC hostile behaviour would obtain attack behaviour as default. Attack behaviour would be the priority of enemy, followed by follow behaviour and wandering. The choice of going for behaviour over setting conditions in allowable actions in each hostile creature remains complex to configure with existing behaviours and NPC cannot choice actions. Since player has only simple attack style, it is best to emphasize design on attack style from NPCs. Hence by nature, designing attack behaviour is the way-to-go. In Attack Behaviour, we can specifically determine which attack style for the NPC attacker onto its enemy (whether that enemy be NPC creatures or player). Through series of conditions and looping all exits available, we find the attack action to be executed if any. Overall, this design enables us to further extend any new attack style for NPC creatures since NPC creatures have diverse attack behaviours. With each creation of attack action style, it has dependency relationship for the three attack styles.

DeathAction---<<uses>>--->PileOfBones
Pile of Bones can be formed after Heavy Skeleton Swordsman is killed. The nature of printable display for the game map shows the implementation of pile of bones as actor makes sure the converted form is shown. With an actor onsite, it is automatically uncrossable for others and suspectable for attack by others. The existing features of Actor makes it possible to achieve the two forementioned requirement. With that, heavy skeleton swordsman would be removed and replaced with pile of bones. The placement of pile of bones in death action ensures the effect happens swiftly instead of replacing it during location tick. DeathAction has a dependency relationship with Pile of Bones.

Assignment 3 REQ Extension (no optional)

Cliff extends Ground,
GoldenFogDoor extends Ground,
Application---<<has>>--->GoldenFogDoor
Application---<<uses>>--->FancyMessage
Cliff is the new ground on a map and instantly kills player when stepped on it. It inherits Ground as Cliff is like subclasses of Ground where, for example, it can be traversed over with restrictions like Floor. Golden fog door also inherits Ground due to similarities in giving ability for player to move to another map. As total number of doors are limited in the world and setting automated destination to the doors is very complex, they will be set at precise location of a map under Application class manually. The other maps (represented by list of String) will be added to world in Application class as well. When Site of Lost Grace is discovered, fancy message of “LOST OF GRACE DISCOVERED” will be revealed.


edu.monash.fit2099.engine
weapons
<<abstract>> WeaponItem
3
actions
DoNothingAction
<<abstract>> Action
ActionList
items
DropItemAction
<<abstract>> Item
<<abstract>> DropAction
DropWeaponItem
actors
<<abstract>> Actor
position
Ground
displays
Display
Extends
game
enemies
Dog
«abstract» HostileCreature
HeavySkeleton Swordsman
LoneWolf
GiantCrab
GodrickSoldier
DeathAction
SellAction
PurchaseAction
Runes
<<abstract>> Trader
MerchantKale
Player
gameWeapon
Grossmesser
Uchigatana
Club
1
environment
Graveyard
GustOfWind
PuddleOfWater
<<abstract>> spawning grounds
Cage
Barrack
0..*
RandomNumberGenerator
Utils
<<Uses>>
Use

REQ2 & REQ3 Introduction (Myint Myat Thura)

**Assignment 3 REQ 2 features extension below

Many of my design choices for REQ2 and REQ3 (which I also did) will revolve around the DRY principle. I believe that non-repetitive code is a necessity for clean and professional software. Most of my implementation decisions will mostly involve extremely good scalability/extension.

REQ2 – Trader & Runes

Runes and Weapons dropping

For runes, we can see that as per instructions they are the main currency of the game (for players). Every actor in the game will carry runes, however, only players can use them in any way they want. This allows us to see something. To best adhere to DRY principle, the goal is to reduce repetitive code. There is no better way to do this than to group the similarities and split the differences.

Brief walkthrough on implementation design – (Single Responsibility Principle, DRY principle)

To understand this more, we can see that runes can be dropped and used. Who can use the runes? ONLY the player, who can drop the runes? EVERYONE. When does everyone drop the runes? When they die. Knowing this, the most efficient way to accomplish dropping runes is to associate dropping the runes with death action. Each actor has an ActionList which stores action objects, in this case DropAction

ActionList --<<stores>>--> Action (e.g., DropAction & DeathAction)

DropAction --<<has>>--> Item

Runes extends Item

This means that when someone dies, it always calls the DropAction inside the actionlist. This is following the Single Responsibility Principle and Don't Repeat Yourself principle. By associating the dropping of runes with the death action and having a dropAction inside the actionlist for each actor, we are reducing repetitive code and ensuring that each class has a specific responsibility.

In the case of enemies with weapons dropping weapons, we could easily manage this by making runes and weapons part of their inventory. Then calling the dropAction for every item in their inventory.

Pros –

1) Associating the dropping of runes with the death action and having a DropAction inside the actionlist for each actor reduces repetitive code and follows the **Don't Repeat Yourself (DRY) principle.**

2) Having a specific DropAction for each actor follows **the Single Responsibility Principle (SRP)** and makes the code more modular and easier to understand.

3) Grouping similarities and splitting differences allows for more **efficient code and easier maintenance.**

4) No matter what the enemies have in their inventory, the rule is that everything in the inventory

will always be dropped. Therefore, adding weapons and runes to inventory would make it **extremely scalable and efficient** in terms of utilizing DropAction to its full potential.

Cons –

1) We are implementing this method with the assumption that all the enemies will drop something on death. If there was ever a function that allows enemies to drop things even while not dead, this would force us to link dropAction with other classes, not just deathAction.
2) If the enemy were to not drop anything upon death, the associative relationship between deathAction and dropAction becomes redundant.

<u>Deciding how many runes to drop and when to decide that.</u>

HostileCreatures --<<uses>>--> RandomNumberGenerator

For what the enemy drops, the requirement is that we have to use a random number generator with a range. However, instead of deciding the rune drops at the moment of death, at the generation of these hostiles, we can assign them a random number for runes to be dropped upon death. This way, our DeathAction calls the least number of methods possible making the code more efficient. This can be accomplished by having a dependency relationship between HostilesCreature abstract class and random number generator class.

<u>Trader</u>

<u>Brief Walkthrough on implementation design – (Single Responsibility Principle, DRY Principle)</u>

Merchant Kale will be the trader for this particular game. At the moment, merchant Kale would not keep track of purchased and sold items in an inventory. Therefore, purchasing and selling would be carried by the player with merchant Kale as a trader with prices of weapons. Trader inherits from the Actor abstract class and merchant Kale inherits from the Trader class. The player will have an association relationship with a new class called PurchaseAction and SellAction which inherits from Action abstract class. And merchant Kale will hold weapon items and runes as attributes to give details of trade prices. Each weapon will be assigned a number of runes.

Actor --<<stores>> Item

Trader extends Actor

Trader --<<has>>--> Runes

Player has the ability to interact with the trader using the PurchaseAction and choosing an item, when the PurchaseAction has been called, sellAction from the trader will be called. This back-and-forth action between the Player and the trader creates a trading exchange.

PurchaseAction && SellAction --<<uses>>--> Trader

Player --<<Uses>>-- purchaseAction && sellAction

This design follows the Single Responsibility Principle, each class will have a specific responsibility and reason to change. For example, trader will manage weapons and assign runes. While PurchaseAction and SellAction have the responsibility of handling the purchase and sell actions respectively.

Pros –

1) The use of inheritance and abstraction allows for **reusability and easier maintenance**, for example, if a change is made to the Actor abstract class, it will reflect in all classes that inherit from it.
2) The use of separate classes for different actions (PurchaseAction and SellAction) follows the **Single Responsibility Principle (SRP)** and makes the code more modular and easier to understand.
3) The use of a dependency relationship between the Player and PurchaseAction classes allows for a **clear separation of concerns** and makes it easier to add new actions in the future.

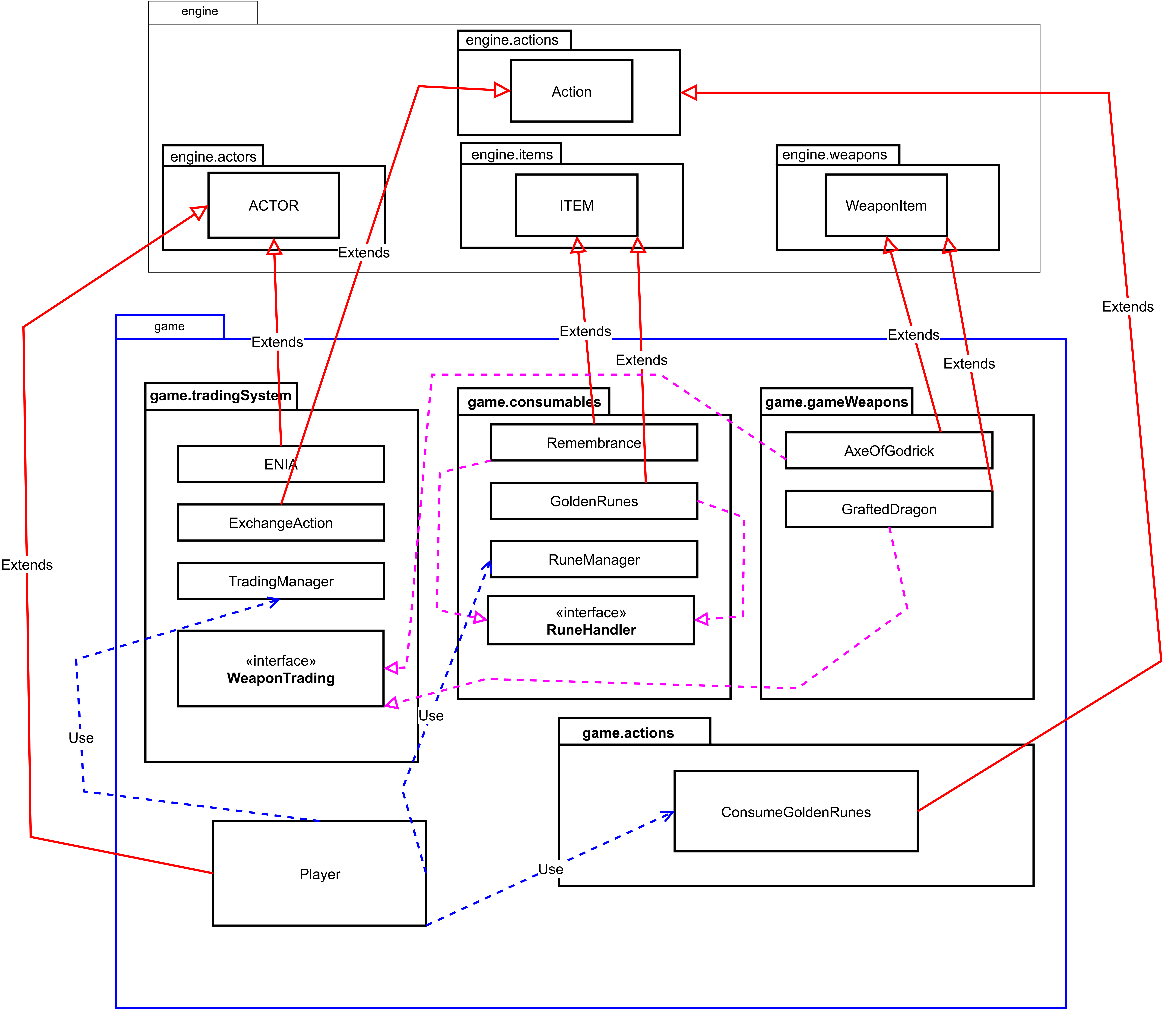
Cons –

1) As new objects and classes are added in the future, there may be a need to create additional Action classes for each new type of interaction. This could lead to an increase in the number of classes and potentially make the code **more difficult to manage.**
2) There may be **some redundancy in the code** if similar actions are performed by different classes. For example, if both the Player and Trader have similarities in methods for interacting with their inventories, there may be some overlap in functionality.

Assignment 3 REQ 2 features extension (no optional)

Cage extends SpawningGround
Cage --<<has>>-->Dog
Dog extends HostileCreature
Barrack extends SpawningGround
Barrack --<<has>>--> GodrickSoldier
GodrickSoldier extends HostileCreature

The two new spawning grounds of Stormveil Castle each spawns Stormveil Castle hostile creatures. Cage and Barrack inherits the existing SpawningGround abstract class due to similarities in functionality. This abstraction in SpawningGround provides a useful extension in features(Open-closed Principle) that does not require much modification or building functionality from scratch ,saving time and possible code repetition if they are similar. As opposed to spawning enemies from either side of the map, Dog and Godrick Soldier do not have direction restriction, hence creating a new instance is sufficient. Using enemy factory classes is only necessary when direction requirement is needed, sticking to Single Responsibility Principle. As shown in REQ 1 for attack behaviours and status for hostile creatures, Dog and Godrick Soldier will have the same attack behaviours and Stormveil Castle status to identify them in AttackBehaviour class as they are hostile creatures but only differ in stats for hit point and spawning chance.


engine
engine.actions
Action
engine.actors
ACTOR
engine.items
ITEM
engine.weapons
WeaponItem
game
game.tradingSystem
ENIA
ExchangeAction
TradingManager
«interface»
WeaponTrading
game.consumables
Remembrance
GoldenRunes
RuneManager
«interface»
RuneHandler
game.gameWeapons
AxeOfGodrick
GraftedDragon
game.actions
ConsumeGoldenRunes
Player
Extends
Extends
Extends
Extends
Extends
Extends
Extends
Extends
Extends
Use
Use
Use

REQ3 – Godrick The Grafted

Before I go on, there are some major changes to talk about that are not directly related to A3. These are great changes that shaped the overall assignment. First are Runes, runes were implemented in a sloppy way back in A2, but we have made it so that it suits the assignment instructions and the principles of OOP. Runes for this reason did not work back in A2, the player was unable to pick up runes or gain runes from enemies, this was fixed by creating an interface and a *Rune Handler* to do this.

Combat Classes, back in A2, we implemented this with just sloppy if-else statements, however this has been changed and Combat Classes now have their own *classes*. They use an interface and a *ClassManager* which allowed for Allies, Invaders and Player to implement each class better.

Now onto the real tasks of REQ3 in A3.

Axe of Godrick and Grafted Dragon

These weapons were implemented just like any other weapons, but they have slightly changed features when it comes to buying and selling. This will be talked about more in the section of new trader Enia.

Golden Runes

Golden Runes were to be scattered on the ground and gives a player the ability to generate random runes in a given range upon consumption.

Brief walkthrough on implementation design – (Single Responsibility Principle, DRY principle)

Golden Runes just like the other normal Rune implements an *interface* called *RuneHandler*. *RuneHandler* has one method called *action*. There will be another class called *RuneManager*, this class is basically a singleton implementing class which will handle all the rune-like objects and perform actions accordingly. For example, for normal runes, it would take the amount of runes that object holds and then add it to the player's total rune. Then it will be removed from the inventory. This makes it so that the player will always have updated runes, but it will never hold a rune object in the inventory to avoid further complication; consider if there were 100 rune objects in the inventory. We are not going to be using them all, we just need the rune amount, so why not take them out? For Golden runes, we are simply just going to generate a number of runes based on the range that is given, then we will simply remove it from the inventory.

GoldenRunes--<<Implements>>--RuneHandler

GoldenRunes--<<Uses>>--RandomNumberGenerator

RuneManager--<<Uses>>--GoldenRunes

Player--<<Uses>>-- RuneManager

Pros:

1. **Single Responsibility Principle:** I implemented the RuneHandler interface and the RuneManager singleton class to promote the *Single Responsibility Principle* by separating concerns and ensuring that each class has a specific responsibility.
2. **Encapsulation:** I used RuneManager to handle all rune-like objects and perform actions accordingly to promote *encapsulation* by hiding the implementation details of how runes are managed from other classes.
3. **Simplified Inventory Management:** I removed rune objects from the inventory after their amount has been added to the player's total rune count to help simplify the inventory management and avoid complications.

Cons:

1. **Difficulty in Testing:** The use of a singleton for the RuneManager class can make it difficult to test and can introduce global state into my application.
2. **Limited Flexibility:** The removal of rune objects from the inventory after their amount has been added to the player's total rune count could limit future flexibility if I decide to add additional functionality or attributes to rune objects.
3. **Unpredictability:** The generation of a random number of runes for Golden Runes could introduce unpredictability into the game, which may not be desirable in all cases.

Remembrance of the Grafted

Remembrance of the Grafted is an item that forces the new trader to implement a new type of trading, *exchange*. It can also be sold for 20000 runes. Elaborated on section for new trader.

Finger Reader Enia

Finger Reader Enia is a new type of trader that allows you to purchase 2 new weapons called Axe of Godrick and Grafted Dragon. But the difference is, instead of runes, it will only accept Remembrance of the Grafted to be *exchanged* for.

Brief walkthrough on implementation design – (Single Responsibility Principle, DRY principle)

This new trader implements the same methods of Kale but instead of having the same items, it will only have two items. Exchange action is a new action that has been created, basically instead of acting on runes, it acts on the items that exist in the player's inventory. This is linked with how we implemented runes, because we will be holding no runes, we can basically assume that the only items that we will have in our inventory is Remembrance of the Grafted, this makes it easy to check if the item is in our inventory. Instead of using interfaces and such, we simply call the size of the inventory, if it is more than 0, then we are sure that we have Remembrance of the Grafted in the inventory. We chose this implementation because it requires nothing else and we are able to reuse the existing methods in the Player class, making the code extremely efficient. The trader will still implement the WeaponTrading interface and the singleton class TradingManager.

Enia--<<Extends>>--Actor

Enia--<<Uses>>--TradingManager

ExchangeAction--<<Extends>>--Action

Enia--<<Uses>>--ExchangeAction

Player--<<Uses>>--ExchangeAction

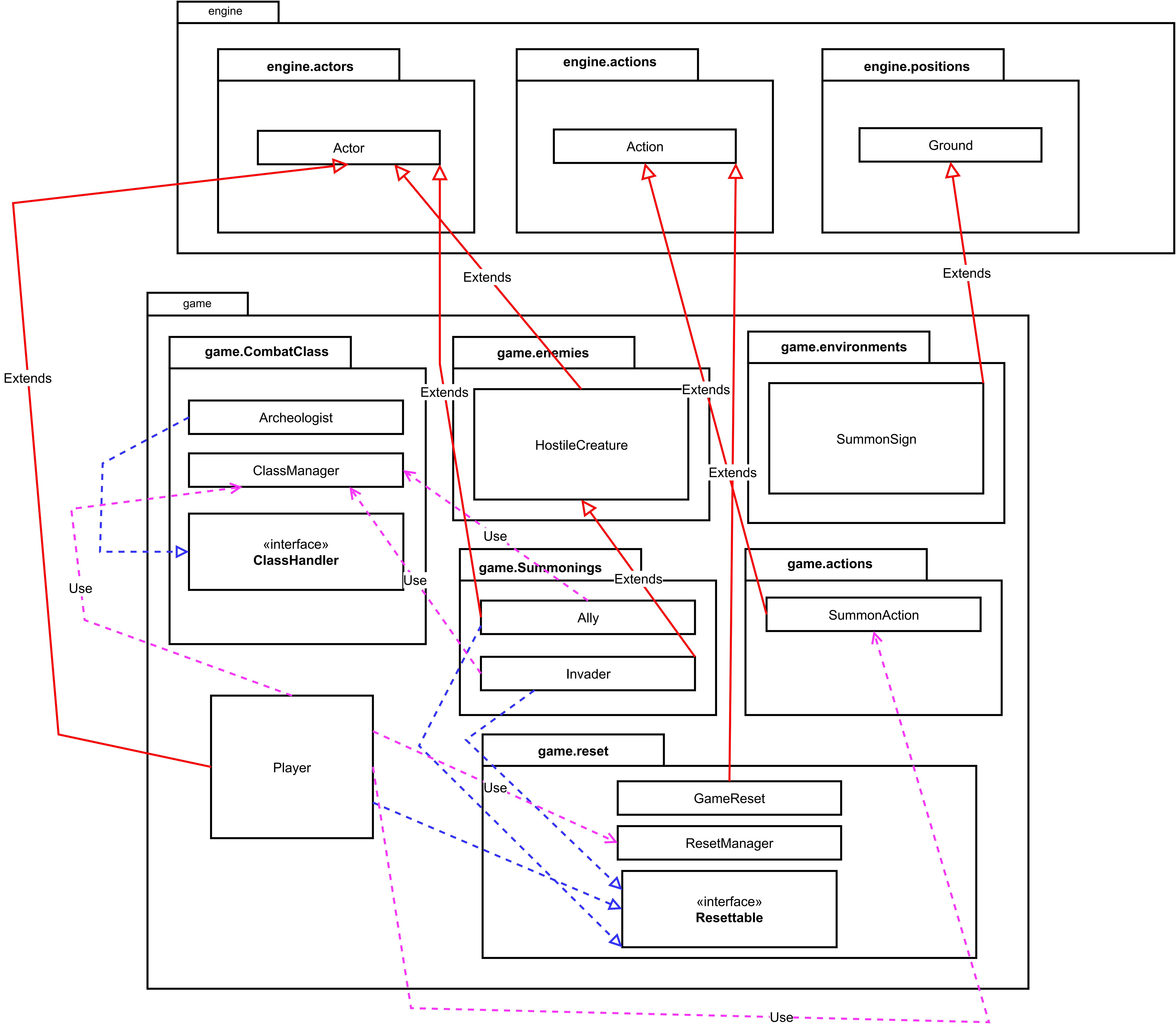
Pros:

1. **Code Reuse (Inheritance):** I chose this implementation because it requires nothing else and I am able to reuse the existing methods in the Player class, making the code extremely efficient. This is an example of the *inheritance* principle in OOP, where a new class can inherit the properties and methods of an existing class to promote code *reuse*.
2. **Simplified Inventory Management (Encapsulation):** By assuming that the only items that we will have in our inventory are Remembrance of the Grafted, I made it easy to check if the item is in our inventory. This is an example of the *encapsulation* principle in OOP, where we *hide the internal details* of an object and only expose what is necessary to reduce complexity and increase *maintainability*.

3. **Efficiency (Polymorphism):** The trader will still implement the WeaponTrading interface and the *singleton* class TradingManager, allowing for efficient code reuse. This is an example of the *polymorphism* principle in OOP, where objects of different classes can be treated as *objects of a common superclass* or interface.

Cons:

1. **Limited Flexibility (Open-Closed Principle):** By only having two items and assuming that the only items in our inventory is Remembrance of the Grafted, we may *limit future flexibility* if we decide to add additional functionality or items to the game. This goes against the *Open-Closed Principle* in OOP, which states that software entities should be open for extension but closed for modification.
2. **Difficulty in Testing (Dependency Inversion Principle):** The use of a singleton for the TradingManager class can make it difficult to test and can introduce global state into my application. This goes against the *Dependency Inversion Principle* in OOP, which states that high-level modules should not depend on low-level modules, but both should depend on abstractions.
3. **Tight Coupling (Law of Demeter):** By relying on the size of the inventory to determine if Remembrance of the Grafted is in our inventory, we may introduce *tight coupling* between the trader and the Player class. This goes against the *Law of Demeter* in OOP, which states that an object should only communicate with its immediate neighbors and should not have knowledge of the inner workings of other objects.


engine
engine.actors
Actor
engine.actions
Action
engine.positions
Ground
game
game.CombatClass
Archeologist
ClassManager
«interface»
ClassHandler
game.enemies
HostileCreature
game.environments
SummonSign
game.Summonings
Ally
Invader
game.actions
SummonAction
game.reset
GameReset
ResetManager
«interface»
Resettable
Player
Extends
Extends
Extends
Extends
Extends
Extends
Extends
Use
Use
Use
Use
Use

REQ4 – A Guest from Another Realm

This REQ4 uses the improvements from the Combat Classes that I have previously talked about in the REQ3 rationale. This made the entire implementation more efficient and abiding by OOP principles.

Death Action has also been reimplemented in a way, so that the Player drops runes at the previous location before death and not at the location of death. This makes it so that it can be implemented well with death from cliff.

Do note the multiple changes and improvements that we have made from A2 implementation.

Astrologer

The Astrologer weapon, since we did not do the optional, uses the Uchigatana weapon from the Samurai class.

Summoning Sign

The requirement states that the Summoning sign must find an appropriate exit so that the ally/invader may be spawned safely. So, instead of finding the exits of the summoning sign, we basically made it so that the player must be directly on top of the summon sign to use it. Then we used the available exits of the player to spawn the ally/invader which basically does the same thing without writing extra code, this allowed us to reuse the code given in the engine therefore making it more efficient.

Ally and Invader

Ally and Invader works the same except that the Invader has so much in common with hostile creatures in the way that they attack Player, so the only difference is that instead of directly inheriting Invader from Actor, we inherit it from the Hostile creature class that we created. Ally will inherit directly from actor.

Brief walkthrough on implementation design – (Single Responsibility Principle, DRY principle)

Ally and Invader are to be spawned by the Summon Sign, the summon sign uses the random generator class to give each a 50% chance of spawning. When they are spawned, they use our newly created ClassManager singleton class to select a random role for them, then they are given the health points and they are given the appropriate weapons. We use the same ClassManager singleton class for both Player and these summoned creatures. This makes it so that our code is efficient and reuses the code. They differ a bit from summon creatures though, because they do not disappear when the player rests at site of grace we have to implement a different method for them in the ResetManager singleton class. At the creation of Ally and Invader, they are passed as instances to the ResetManager class. Then we make it so that they are only removed when the player dies. Therefore, we are then again making use of the existing classes for the code.

Pros:

1. **Code Reuse (Inheritance):** I used the same ClassManager *singleton* class for both Player and these summoned creatures to make my code efficient and reuse the code. This is an example of the *inheritance* principle in OOP, where a new class can inherit the properties and methods of an existing class to promote code *reuse*.
2. **Efficiency (Polymorphism):** At the creation of Ally and Invader, they are passed as instances to the ResetManager class. This allows for efficient code reuse by making use of the existing classes for the code. This is an example of the *polymorphism* principle in OOP, where objects of different classes can be treated as objects of a *common superclass or interface*.
3. **Simplified Management (Encapsulation):** By implementing a different method for Ally and Invader in the ResetManager *singleton* class, I made it so that they are only removed when the player dies. This is an example of the *encapsulation* principle in OOP, where we hide the internal details of an object and only expose what is necessary to *reduce complexity and increase maintainability*.

Cons:

1. **Limited Flexibility (Open-Closed Principle):** By using the same ClassManager *singleton* class for both Player and these summoned creatures, we may limit future *flexibility* if we decide to add additional functionality or attributes to either class. This goes against the *Open-Closed Principle* in OOP, which states that software entities should be open for extension but closed for modification.
2. **Difficulty in Testing (Dependency Inversion Principle):** The use of singletons for the ClassManager and ResetManager classes can make it difficult to test and can introduce global state into my application. This goes against the *Dependency Inversion Principle* in OOP, which states that high-level modules should not depend on low-level modules but both should depend on abstractions.
3. **Tight Coupling (Law of Demeter):** By passing Ally and Invader as instances to the ResetManager class at their creation, we may introduce tight coupling between these classes. This goes against the *Law of Demeter in OOP*, which

states that an object should only communicate with its immediate neighbors and should not have knowledge of the inner workings of other objects.

Ally--<<Extends>>--Actor

Invader--<<Extends>>--HostileCreature

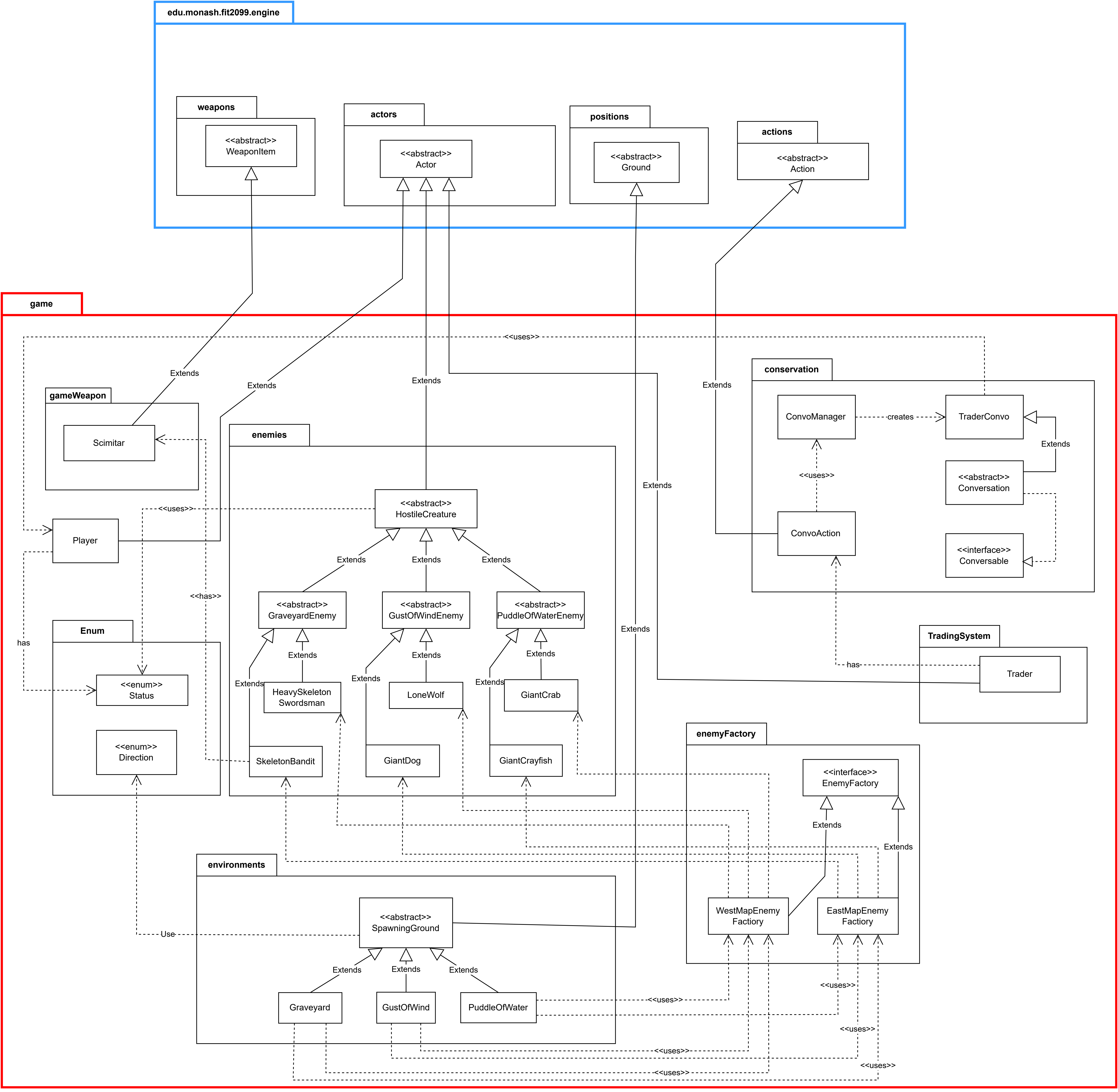
ResetManager--<<Uses>>--Ally || Invader

Player--<<Uses>>--GameReset

Player--<<Uses>>--SummonAction

SummonAction--<<Creates>>--Ally || Invader

Ally || Invader--<<Uses>>--ClassManager


edu.monash.fit2099.engine
weapons
<<abstract>> WeaponItem
actors
<<abstract>> Actor
positions
<<abstract>> Ground
actions
<<abstract>> Action
game
<<uses>>
Extends
gameWeapon
Scimitar
Player
enemies
<<abstract>> HostileCreature
<<abstract>> GraveyardEnemy
<<abstract>> GustOfWindEnemy
<<abstract>> PuddleOfWaterEnemy
HeavySkeleton Swordsman
LoneWolf
GiantCrab
SkeletonBandit
GiantDog
GiantCrayfish
<<has>>
has
Enum
<<enum>> Status
<<enum>> Direction
conservation
ConvoManager
creates
TraderConvo
<<abstract>> Conversation
ConvoAction
<<interface>> Conversable
TradingSystem
Trader
enemyFactory
<<interface>> EnemyFactory
WestMapEnemy Factory
EastMapEnemy Factory
environments
<<abstract>> SpawningGround
Use
Graveyard
GustOfWind
PuddleOfWater

***Changes are below*
***Assignment REQ creative requirements below. Changes to enemy factory is explained below.*

The game introduces map partition, more enemies and additional weapon is introduced. Gamemap has been divided into West (Right half) and East (Left half). Three environments in each half spawn different enemies but carries some similar characteristics. For example, Graveyard on the West spawns HeavySkeletonSwordsman while Graveyard on the East spawns SkeletalBandit but they are still the same type regardless of their geographic difference. An additional weapon is introduced for SkeletonBandit called Scimitar.

The additional features model around design rationale of REQ 1. As a result, features would be modularised with existing package and classes to prove the extendibility of REQ 1 (Open-closed principle) and obey the importance for abstraction (Dependency inversion principle).

A. Environment

SpawningGround---<<has>>--->Location,
SpawningGround---<<has>>--->Direction,
The partition of GameMap for spawning grounds can be done with identification of the location of ground. We inquire a parameter of Location in the constructor of spawning ground and identify the direction from the x and y coordinates. The identified direction would be set with an enum type of Direction in spawning ground Direction type attribute. An alternative would be to create two separate maps, but this would require a combination of map for better experience or peripheral methods to transition character movement which can be complicated. Spawning ground has an association relationship with Location as an attribute to identify the direction and Direction enum type is used to prevent Magic Strings (String can be used but may cause error).

HostileCreature---<<has>>--->Direction,
HostileCreature---<<has>>--->Status
Hostile creatures have Direction enum type to help spawning grounds for setting the correct enemy set. Every hostile creature has a Direction on the game map and so it has association relationship. Status enum type tracks which type of enemy an enemy can attack. However, further identification would require display character when performing attack action. Using Direction enum type avoids Magic Strings error.

SkeletonBandit extends HostileCreature,
GiantDog extends HostileCreature,
GiantCrayfish extends HostileCreature,
SkeletonBandit---<<uses>>--->Status,
GiantDog---<<uses>>--->Status,
GiantCrayfish---<<uses>>--->Status,
The three additional enemies extend Hostile Creature class due to similarities with the three other existing enemies. They inherit the class and establish association relationship. The dependency relationship of Status type is for identification of other enemies' Status.

SkeletonBandit---<<uses>>--->PileOfBones
Just like HeavySkeletonSwordsman, SkeletonBandit can turn to Pile of Bones once killed. The temporary existence of Pile of Bones meant creation of Object to track the progress for revival or death action.This usage is dependency relationship.

SkeletonBandit---<<has>>--->Scimitar,
SkeletonBandit---<<uses>>--->AttackAction,
SkeletonBandit---<<uses>>--->SpinningAttackAction,
SpinningAttackAction---<<uses>>--->AttackAction,
AttackAction---<<uses>>--->Scimitar
Skeleton Bandit is the same type of enemy from graveyard but on the East side. Unlike Heavy Skeleton Swordsman, Skeleton Bandit carries Scimitar. This weapon belongs to the enemy, and it has an association relationship with the enemy. Skeleton Bandit attacks the same as Heavy Skeleton Swordsman. Skeleton Bandit can opt for AttackAction with Scimitar object or call SpinningAttackAction method with multiple AttackAction with Scimitar objects. The approach adheres to the one for HeavySkeletonSwordsman due to similar attack method codes and prove of reusing SpinningAttackAction benefit code reusability (DRY principle).

GiantCrab---<<uses>>--->SlamAttackAction,
GiantCrab---<<uses>>--->AttackAction,
GiantCrayfish---<<uses>>--->SlamAttackAction,
GiantCrayfish---<<uses>>--->AttackAction,
SlamAttackAction---<<uses>>--->AttackAction,
Giant Crab is same type of Lone Wolf but attacks like GiantCrayfish (the type of Puddle of Water). Both Giant Crab and Giant Crayfish attack via sole target attack or area attack via SlamAttackAction. Sole target attack consists of calling AttackAction object to actionlist, giving dependency relationship. The area attack is through SlamAttackAction which calls itself to action list and create multiple instances of AttackAction on all nearby enemies. Approach is like SpinningAttackAction to reduce repeated codes and allow modularity in the attack execution with repurposing methods from AttackAction class (extendibility).

Scimitar extends WeaponItem
MerhantKale---<<has>>--->WeaponItem
Scimitar can be sold by Merchant Kale and purchased by the player. Since Merchant Kale keeps weapon item as goods (association relationship), Scimitar inherits WeaponItem allows Merchant Kale to sell it. The inheritance of abstract class of WeaponItem makes it extendable for further weapon items (Open-closed principle). The inheritance is an association relationship as it derives as an Item that can be used as weapon for damage. Merchant Kale stores WeaponItem class objects for sale.

The design rationale for rest of the links within the game remain similar. Purchase Action is still the same approach but with additional item for purchase or sale. Interaction within the game is similar as there is no modification to the logic of game but only extension of game elements.

Changes to REQ 5

Majority of the general classes in engine package has been removed due to redundancy. Exisiting features from REQ 1 design would not be repeated in the REQ 5. This is reflected in removal of attack style for the new hostile creatures as they still have same attack behaviours to determine the attack action/style needed. As for trading of Scimitar (a new weapon), it will be an additional weapon item by Merchant Kale and something player can sell for in sell action. The new addition to the design would be the creation of NPC hostile creatures from spawning grounds on two different sides.

HeavySkeletonSwordsman extends GraveyardEnemy,
SkeletonBandit extends Graveyard,
LoneWolf extends GustOfWindEnemy,
GiantDog extends GustOfWindEnemy,
GiantCrab extends PuddleOfWaterEnemy,
GiantCrayfish extends PuddleOfWaterEnemy
Each spawning group has two types of enemies. They belong in the same enemy type, just spawn on either east or west side of the spawning ground. To minimise code repetition, three abstract classes of spawning ground enemy type are created to merge any similarities. This new layer offers new set of identification in status and follows Single Responsibility Principle. Future enemy of the same type can inherit any of the abstract classes and only define the unique values in its own class. It can help to reduce dependency on each enemy class.

GraveyardEnemy extends HostileCreature,
GustOfWindEnemy extends HostileCreature,
PuddleOfWater extends HostileCreature
The three enemy type classes inherit the functionalities in hostile creature class. Hostile creature class still manages most of the action in enemy play turn and allowable actions.

EastMapEnemyFactory implements EnemyFactory,
WestMapEnemyFactory implements EnemyFactory
The east and west enemy factory classes implements newEnemy function in EnemyFactory interface. The newEnemy function is designed to return a hostile creature based on direction of spawning ground. EnemyFactory can be extended in the future with more new methods for implementation but restricted modification due to its abstraction (Open-closed principle).

EastMapEnemyFactory---<<uses>>--->SkeletonBandit,GiantDog,GiantCrayfish
WestMapEnemyFactory---<<uses>>--->HeavySkeletonSwordsman,LoneWolf,GiantCrab
In each of the enemy factory, it will return a new instance of enemy based on the location of spawning ground provided and the spawning ground type itself.

SpawningGround---<<uses>>--->WestMapEnemyFactory,EastMapEnemyFactory
Inside spawning ground spawnHostileCreature method, new instance of either WestMapEnemyFactory or EastMapEnemyFactory is called based on direction of spawning ground. The subclasses of spawning ground will call the spawnHostileCreature method and utilises a broader search of enemy factory to determine enemy instance. Since all subclasses

of spawning grounds needs to know which enemy to spawn, calling a common search prevent code repetition in subclasses and reduces tight coupling on each subclass to determine on their own which creature to spawn.

SpawningGround---<<uses>>--->Direction
To determine direction of a spawning ground, it calls a static direction method from Utils class and returns a enum direction type (EAST or WEST). Having enum type is preferable over returning direction in String since the value is final and easily prevents magic string issue.

SkeletonBandit---<<has>>--->Scimitar
Skeleton bandit would have a scimitar as a weapon in its inventory. Scimitar would be considered as weapon item to be linked with Skeleton Bandit and it can be removed from the enemy. A new Scimitar instance is created and added to Skeleton Bandit inventory.

Changes to REQ 5 (for assignment 3)
The restrictive uses of generating enemies from West/East Enemy factory alone did not help the spawning of Stormveil castles enemies as they do not have direction requirement. Though Stormveil Castle enemies can adhere to the requirements of West/East for creating new instances, such practice forces class to use the generative tool (which does not follow Interface Segregation Principle). Hence, enemy factory related class and abstraction will only be tied to enemies that have direction. In the case of Stormveil Castle enemies, new instances will be called directly instead of going through Enemy Factory.

Assignment 3 REQ creative requirement
Conversation implements Conversable,
TraderConvo extends Conversation,
TraderConvo---<<uses>>--->Player,
Player---<<uses>>--->Status,
ConvoManager---<<creates>>--->TraderConvo,
ConvoAction extends Action,
ConvoAction---<<uses>>--->ConvoManager,
Trader---<<has>>--->ConvoAction
For this creative requirement, a short conversation feature is created between player and trader. Trader will converse with player with narration and player will respond through numbered options for response. This design takes into account of future extension of more conversation with other friendly actors or anyone and use of OOP and abstraction to easily segregate heavy and complex functionalities.

Conversable interface is created to store the two actions in a dialogue (which are talk and respond). An argument scene that involves more than dialogue , as an example, can utilise this conversable interface (Open-closed Principle) to act out a scene in the black actions.

The idea of having an abstract class of Conversation is to branch out to convos with different actors instead of implementing Conversable interface directly. This method helps ConvoManager to return different convos that are under Conservation abstract class

(Dependency Inversion Principle), without needing to set different convo getter. Through speaker display character, ConvoManager returns the right instance of convo class.

To perform conversation, it is done through ConvoAction which inherits Action class due to its integration within the game in play turn. Player will have a Talking Status to allow conversation. Trader would have ConvoAction in allowable actions for actors with Talking status to initiate (this case , the player). Sticking to Single Responsibility Principle, ConvoAction would execute the flow of a conversation solely. It uses ConvoManager to choose Convo class and does the talk and respond execution.

In the trader convo, trader will ask player for 5 runes for a conversation ; hence the use of Player instance to decrease runes amount.