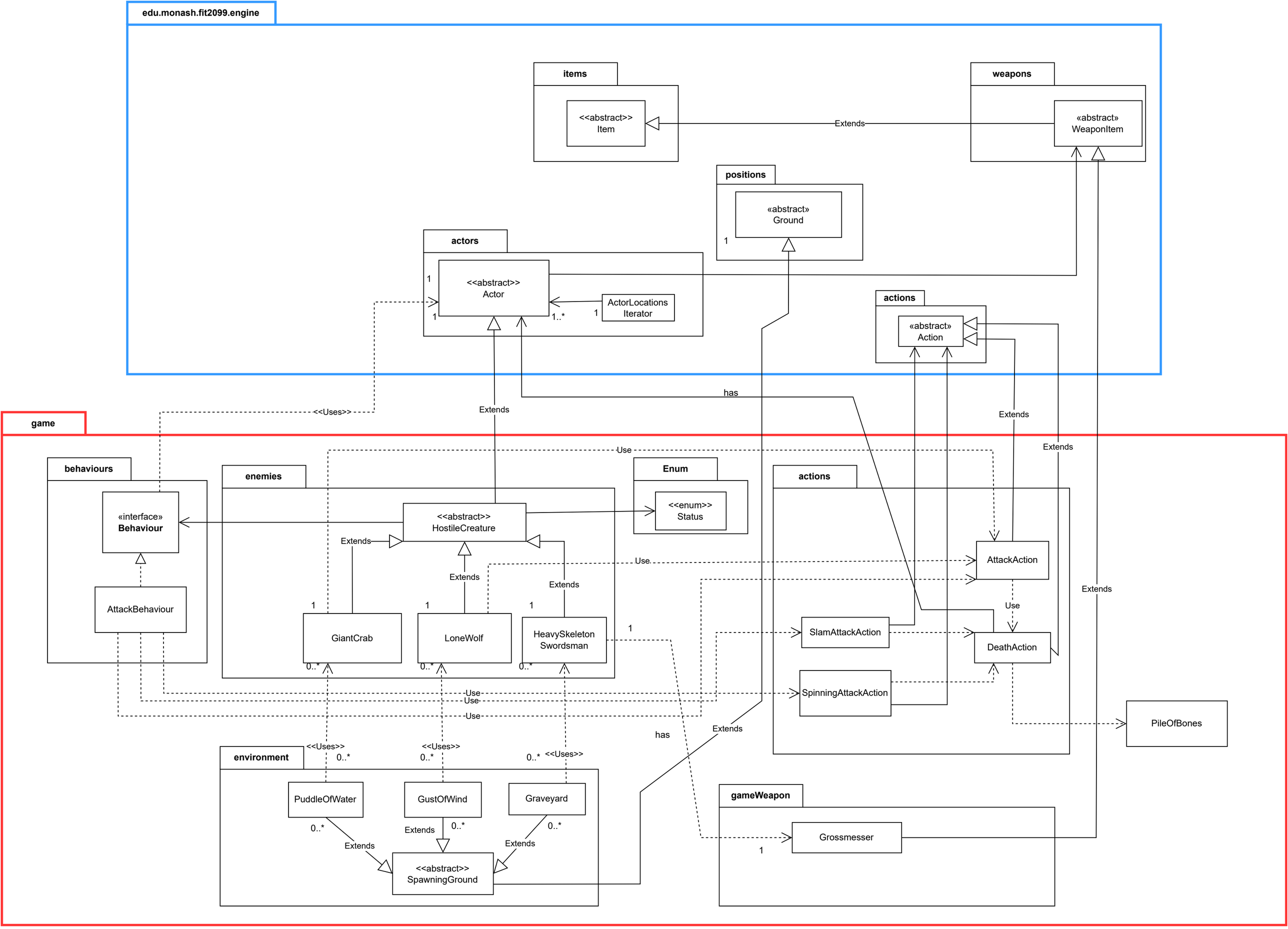

edu.monash.fit2099.engine
items
<<abstract>>
Item
Extends
weapons
«abstract»
WeaponItem
positions
«abstract»
Ground
1
actors
1
<<abstract>>
Actor
1
1..*
1
ActorLocations
Iterator
actions
«abstract»
Action
has
<<Uses>>
Extends
Extends
Extends
game
behaviours
«interface»
Behaviour
AttackBehaviour
enemies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

***Changes are 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

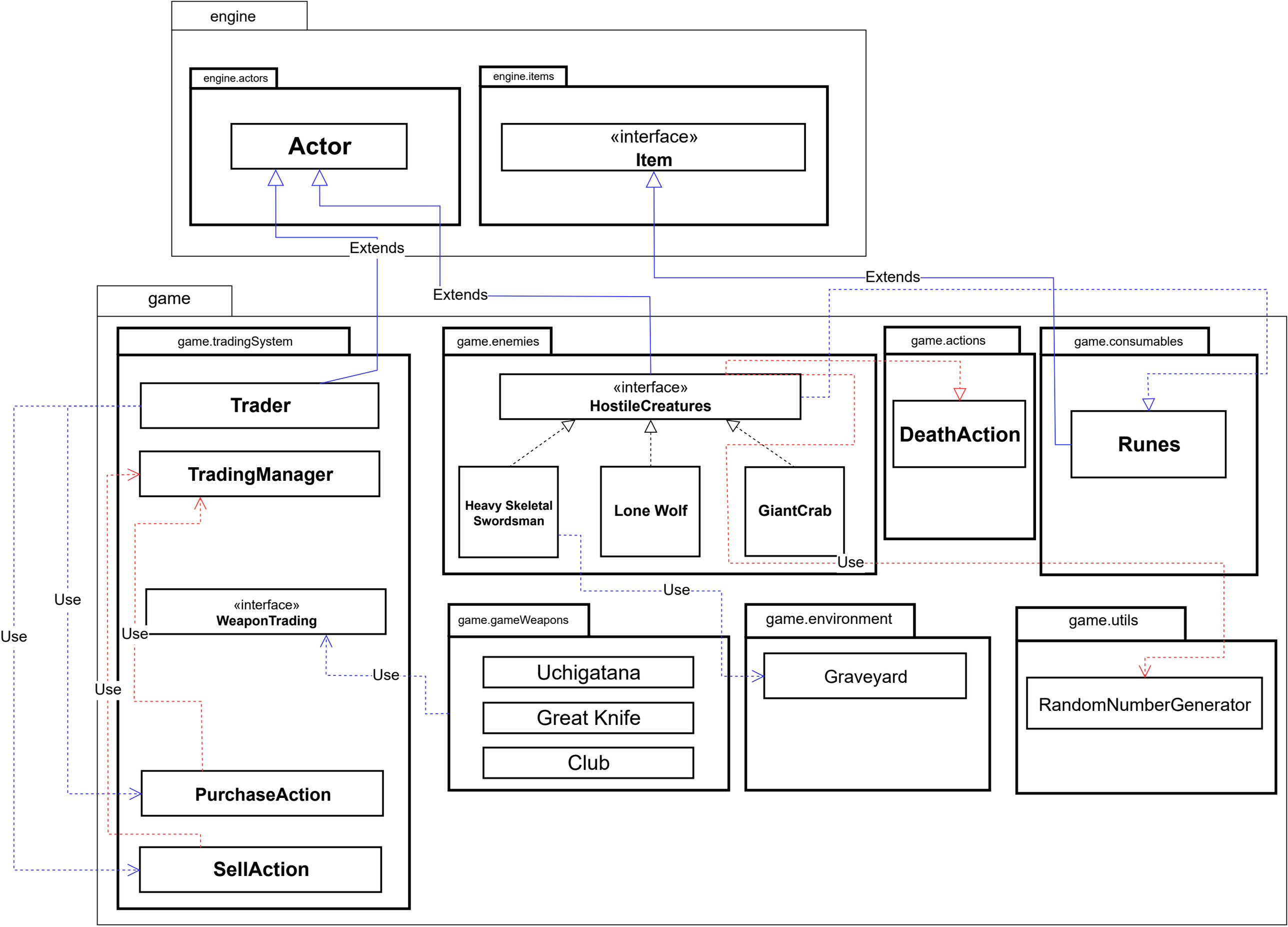
The majority of the existing classes in engine illustrated in the UML diagram has been removed (includes the capabilities package, dropWeapon and dropItem Action, Weapon Interface and Instrinsic Weapon classes). The big change to this design would include Attack Behaviour class to facilitate attack action 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.


engine
engine.actors
Actor
engine.items
«interface»
Item
Extends
Extends
Extends
game
game.tradingSystem
Trader
TradingManager
«interface»
WeaponTrading
PurchaseAction
SellAction
Use
Use
Use
Use
Use
game.enemies
«interface»
HostileCreatures
Heavy Skeletal Swordsman
Lone Wolf
GiantCrab
Use
Use
game.gameWeapons
Uchigatana
Great Knife
Club
game.actions
DeathAction
game.consumables
Runes
game.environment
Graveyard
game.utils
RandomNumberGenerator

REQ2 & REQ3 Introduction (Myint Myat Thura)

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.

There have been numerous, extremely good, improvements over my last design. You see, my previous design for A1 was the way it was because the ban on the use of instanceof and downcasting was not exactly known to me. In all my REQs I will be making use of interfaces to achieve polymorphism and abstraction.

REQ2 – Trader & Runes

Trader and Trading

Trading is the biggest focus of this REQ, to truly perform using total abstraction and polymorphism, we had to see how the system can be implemented where we let the objects speak for themselves instead of their respective methods having to speak for who they are. Using the popular design pattern called Visitor Pattern was considered but due to the code being extremely unreadable, we finally figured out the answer. We would implement the Singleton.

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

There are only a few weapons in the weapons that can be bought from the trader. Therefore, if we had one hundred weapons, it would be difficult to write code to buy/sell each weapon. Enters the **WeaponTrading** interface, every single tradable weapon in the game would implement this interface which has the methods called purchase and sell. These methods would, depending on the weapon, will do different things when selling. This was useful since we never actually meet the same weapon twice, the price might be different, the character might be different. Each weapon will be passed off to the **TradingManager** as "Tradable" instances and this makes it so that we do not have to specify what should be done with these instances based on their types. Tradable will then call PurchaseAction and SellAction depending on what the user decides to do.

WeaponItem--<<Uses>>--TradingManager(passing the instances (Uchigatana,GreatKnife, etc.))

TradingManager--<<stores>>--Tradable(e.g, Uchigatana, GreatKnife, Club)

TradingManager--<<Uses>>--PurchaseAction || SellAction

This implementation was improved upon the last one which does not use any interfaces, which forces the use of downcasting the instanceof which destroys the entire point of polymorphism and abstraction. This new method of handling purchases is made so that it manages purchases efficiently and follows the OOP principles.

Pros:

1. **Flexibility**: By using an interface, I can easily add new weapons to the game without having to modify my existing **TradingManager** code. This makes my code more flexible and easier to maintain, adhering to the *Open-Closed Principle*.
2. **Modularity**: By depending on an abstraction rather than a concrete implementation, I reduce the coupling between my **TradingManager** and the weapons in the game. This makes my code more modular and easier to change, following the *Dependency Inversion Principle*.
3. **Adherence to OOP principles**: My implementation follows several important Object-Oriented Programming principles such as the Single Responsibility Principle, the **DRY** principle, the *Open-Closed Principle*, and the *Dependency Inversion Principle*. This makes my code more robust and easier to understand.

Cons:

1. **Increased complexity**: Using an interface can increase the complexity of my code, as I need to define and implement the interface methods for each weapon.
2. **Potential for errors**: If a weapon does not correctly implement the **WeaponTrading** interface methods, it could lead to errors or unexpected behavior when trading that weapon.
3. **Limited control**: By depending on an abstraction rather than a concrete implementation, I have less control over the specific actions performed when trading a weapon. This could make it more difficult to implement certain features or behaviors.

Deciding how many runes to drop and when to decide that.

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

HostileCreature--<<uses>>-->DeathAction

For what the enemy drops, the requirement is that we must 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 at the Start. In the previous implementation, we thought the enemies would drop the runes then the player would be given a chance to give it up. But upon further inspection, we found a way to implement that in a way that no runes will be dropped. Instead, we would directly add the number of runes into the player's inventory. This eliminates the unnecessary use of DropItemAction since it is not dropped at all.

engine

engine.actors

ActorLocationIterator

Actor

engine.positions

GameMap

Locations

Ground

engine.action

DropItemAction

PickUpItemAction

engine.items

«interface»
Item

Use

Extends

game

game.reset

GameReset

ResetManager

<<interface>>
Resettable

game.enemies

«interface»
HostileCreatures

Player

game.environment

SiteOfLostGrace

game.consumables

FlaskOfCrimsonTears

Runes

game.actions

DeathAction

REQ3 – Grace and Game Reset

Game Reset

Resetting the game is the biggest focus of this REQ, to really perform this efficiently and *correctly* we had to use something called Singleton. This implementation would work by passing each and every actor instance as a resettable and then performing the reset all at once.

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

The ResetManager uses a private constructor and has a getInstance method which will only return the same instance of the ResetManager, this makes sure that every actor object will be passed into the same ResetManager. You would think that we would use different methods for different objects. Narrowing it down and looking at it realistically, we have player and we have hostile creatures. That's it, those two are the only things that can be reset. *But* we have implemented Player to also use singleton, what does that mean? It means that Player no longer has to be passed as a resettable for ResetManager, this means that the GameReset method can be called on Player in it's own way just by writing a few lines in the ResetManager. ONLY the hostile creatures will be passed to ResetManager as resettables. This allowed us to make the code much cleaner and this allowed extreme scalability for the HostileCreatures. We have considered to let Player implement Resettable interface but realistically, there will be no other actor like Player, because it only makes sense that we can only control one player.

HostileCreature--<<Uses>>--DeathAction

DeathAction--<<Uses>>--ResetManager

Player--<<Uses>>--DeathAction

ResetManager--<<Uses>>--Player

This comes to another question, what about the site of grace reset? It was evident to us that the only difference between Site of Grace Reset and Death Reset was that Site of Grace reset does not drop anything but it resets runes. This makes it seem as if the implementation can be slightly altered in the ResetManager for SOG (Site of Grace) reset. This is not the case however since these two have a huge difference in terms of design that we are simply not able to see from the surface. Because SOC reset is only activated by the player, it would make no sense getting it together with the

ResetManager class. Another thing is that, Site of Grace reset also uses numerous other classes that are not within ResetManager such as GameMap or Locations, this is to meet the requirements of respawning at the last site of grace visited when a player dies. This is where it vastly differs from DeathReset. This has helped the extendability because Site of Grace can now freely use other features from other classes without the worry of clumping up ResetManager code.

Player--<<Uses>>--Site Of Lost Grace

SiteOfLostGrace--<<Uses>>--GameReset

GameReset--<<Uses>>--GameMap

Pros:

1. **Scalability**: By only passing hostile creatures to my ResetManager as resettables, I have made my code more scalable for adding new hostile creatures to the game. This follows the *Open-Closed Principle*, allowing me to easily extend my code without modifying existing functionality.
2. **Clean code**: By separating the Site of Grace reset from the ResetManager class, I have avoided clumping up my ResetManager code and have made it easier to use features from other classes such as GameMap or Locations. This follows *the Single Responsibility Principle*, ensuring that each class has a well-defined responsibility.
3. **Adherence to OOP principles**: My implementation follows several important Object-Oriented Programming principles such as the *Single Responsibility Principle and the Open-Closed Principle*. This makes my code more robust and easier to understand.

Cons:

1. **Increased complexity**: By using singletons for both the ResetManager and the Player class, I have increased the complexity of my code. Singletons can make it more difficult to test and maintain my code.
2. **Potential for errors**: If a hostile creature does not correctly implement the Resettable interface methods, it could lead to errors or unexpected behavior when resetting that creature. This highlights the importance of following the *Liskov Substitution Principle*, ensuring that subclasses behave correctly when substituted for their base class.
3. **Limited control**: By separating the Site of Grace reset from the ResetManager class, I have less control over how resets are performed in different situations. This could make it more difficult to implement certain features or behaviors.

Deciding how runes are handled Upon Death

The runes will be dropped when the player dies, however when the player dies a second time, the runes on the ground will be dropped. This was done by implementing a single variable in DeathAction that keeps track of how many times you have died.

First Death -> DropItems -> Set Previous Death Location

Second Death -> DropItems -> RemoveItem at Previous Death Location -> Set new Previous death Location

Third Death -> Drop Items -> Remove Item at Previous Death Location -> Set new Previous death location

This was implemented in this way because it really is the only way. However to remove item at previous death location and setting new previous death location, this was not handled by secondary classes but was handled by simple attributes in Player. You might remember that we used Singleton for player, this has shown to be incredibly useful in this case, by saving the previous location in the player attribute, we can be sure that it returns the correct location every single time due to singleton, there can be no mistakes. Since the player will only implement these methods, I feel as if we have made the right decision in using singleton for player. This has made our implementation much simpler and efficient.

Player--<<Uses>>--DeathAction
DeathAction--<<Uses>>--DropAction
DropAction--<<Uses>>--GameMap
DropAction--<<Uses>>--Location
DeathAction--<<Uses>>--Player (using the location attribute from player)

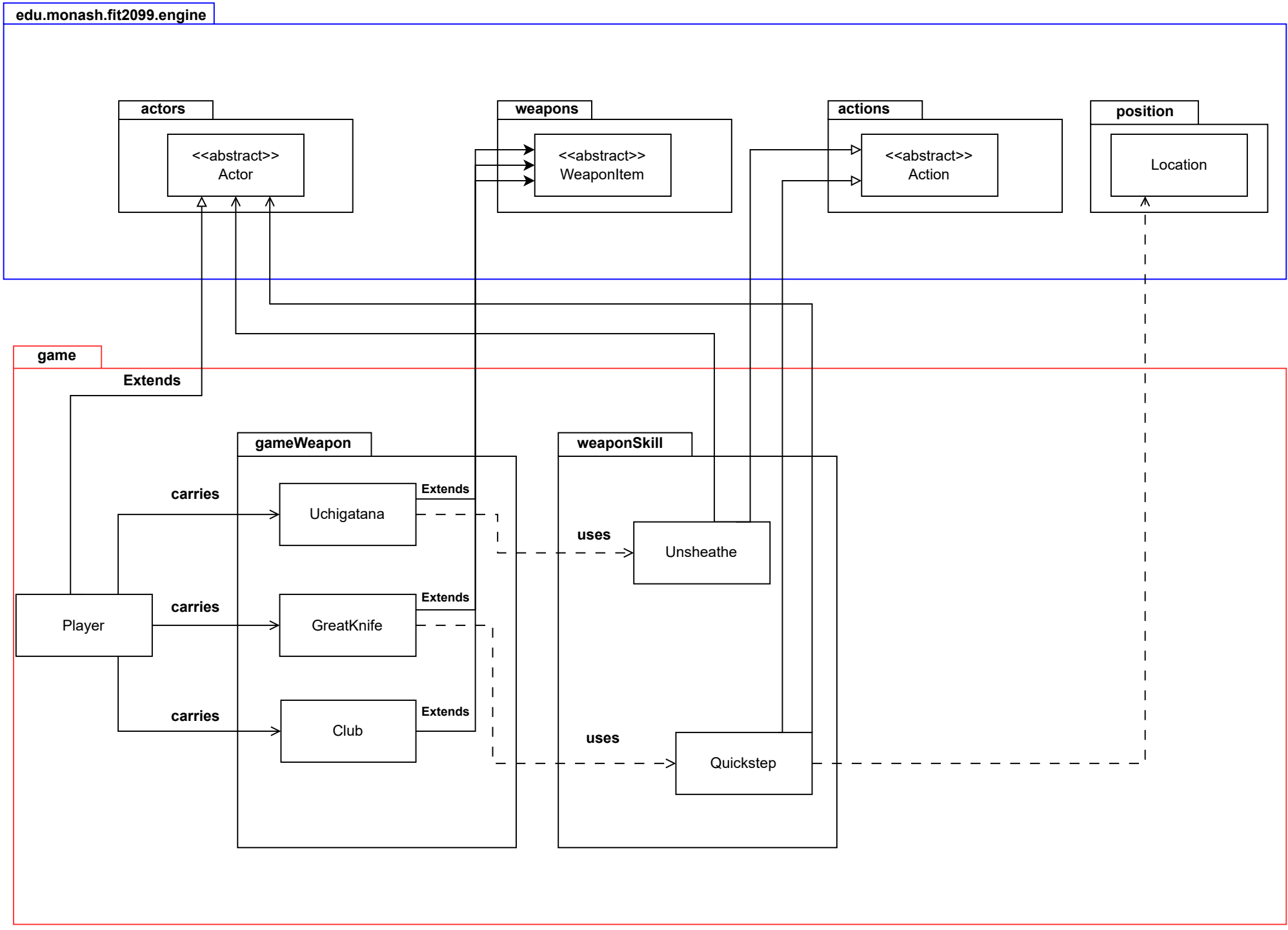
Pros:

1. **Simplicity**: By implementing a single variable in my DeathAction to keep track of the number of times the player has died, I have made my implementation simple and efficient. This follows the *KISS (Keep It Simple, Stupid) principle*, which encourages simplicity in design.
2. **Efficiency**: By using *a singleton* for the Player class and saving the previous death location as an attribute in the Player, I have ensured that the correct location is returned every time. This has made my implementation more efficient.

3. **Adherence to OOP principles**: My implementation follows several important Object-Oriented Programming principles such as the *Single Responsibility Principle* and the *Open-Closed Principle*. This makes my code more robust and easier to understand.

Cons:

1. **Increased complexity**: By using a singleton for the Player class, I have increased the complexity of my code. *Singletons* can make it more difficult to test and maintain my code, potentially violating the *Dependency Inversion Principle* by tightly coupling my code to a specific implementation.
2. **Potential for errors**: If the DeathAction variable is not correctly updated or if the previous death location attribute in the Player is not correctly set, it could lead to errors or unexpected behavior when resetting runes. This highlights the importance of following the *Liskov Substitution Principle*, ensuring that subclasses behave correctly when substituted for their base class.
3. **Limited control**: By handling the removal of items at the previous death location and setting the new previous death location using simple attributes in the Player, I have less control over how these actions are performed. This could make it more difficult to implement certain features or behaviors.


edu.monash.fit2099.engine
actors
<<abstract>>
Actor
weapons
<<abstract>>
WeaponItem
actions
<<abstract>>
Action
position
Location
game
Extends
gameWeapon
weaponSkill
carries
Uchigatana
Extends
uses
Unsheathe
Player
carries
GreatKnife
Extends
carries
Club
Extends
uses
Quickstep

REQ 4: Weapons and Unique Skills

Player extends <<Abstract>> Actor

The Player Extend <<Abstract>> Actor serves as the parent class for the Samurai and Bandit classes, which both inherit from it. This design allows for the implementation of unique characteristics and abilities for each class, while still maintaining a common set of behaviours and attributes that all players share.

Overall, the use of an abstract class for the Player Extend Actor promotes modularity and flexibility in the game design, enabling easier addition of new player classes in the future if desired.

Player ----<<carries>>----→Club
Player ----<<carries>>----→Uchigatana
Player ----<<carries>>----→Great Knife

The above design shows a composition relationship between the Player class and three different weapon classes, Club, Uchigatana, and Great Knife. This design follows the OOP principle of composition, where the Player class is composed of various weapons classes. One of the major advantages of this design is that it allows for easy addition of new weapons in the future. For example, if the game developer decides to introduce a new weapon such as a bow or a spear, it can be easily added to the Player class without affecting the existing codebase. This makes the design flexible and extensible.

In summary, the design of having the Player class composed of various weapon classes using the composition principle is a flexible and extensible design that allows for easy addition of new weapons in the future. However, it may become complex if there are too many weapon classes to manage and may require a lot of code changes if the weapon properties or methods change frequently.

Uchigatana-----<<uses>>----->Unsheathe

Great Knife-----<<uses>>----->Quickstep

The design shows a clear relationship between the Uchigatana and the Unsheathe skill and the Great Knife and the Quickstep skill. One pro of this design is that it promotes modularity and reusability of code. The skills can be used by different weapons, and new weapons can easily be added by simply implementing the "uses" relationship with the appropriate skill. A con of this design is that it may become complex if there are too many weapons and skills to manage. Additionally, if a weapon has multiple skills, then the design may become more complicated, and the implementation may be more challenging.

Unsheathe extends Action

Quickstep extends Action

The design of the Unsheathe and Quickstep classes extending the Action class follows the Open-Closed Principle of OOP, which states that classes should be open for extension but closed for modification. By creating separate classes for these actions, we can easily add new actions in the future without having to modify the existing code.

One potential extension of this design would be to add more actions that the player can perform, such as blocking or dodging. By creating separate classes for each action, we can keep our code modular and easy to maintain.

The main advantage of this design is that it allows for a clear separation of concerns and easy extensibility. However, one potential disadvantage is that it can lead to an increase in the number of classes, which can make the code more difficult to understand and navigate. Overall, the design of the Unsheathe and Quickstep classes extending the Action class is a good example of the Open-Closed Principle, which promotes a modular and extensible design.

Club extends WeaponItem

Uchigatana extends WeaponItem

Great Knife extends WeaponItem

The design of Club, Uchigatana, and Great Knife extending WeaponItem follows the OOP principle of inheritance. By extending the WeaponItem class, each of these subclasses inherits the properties and behaviors of the parent class. This makes it easier to add new weapon items in the future, as they can simply extend the WeaponItem class and inherit its properties and behaviors.

One potential advantage of this design is that it allows for easier maintenance of the code. Changes to the properties or behaviors of the WeaponItem class will be inherited by its subclasses, reducing the amount of code that needs to be changed. Additionally, code reuse is promoted by reducing the amount of duplicate code. One potential disadvantage of this design is that it can lead to tight coupling between the parent class and its subclasses. Changes to the parent class can have unintended consequences for its subclasses, and vice versa.

Additionally, subclassing can result in an excessive number of classes, making the code more difficult to understand and maintain. Overall, the design of Club, Uchigatana, and Great Knife extending WeaponItem is a good example of inheritance, but care must be taken to avoid tight coupling and class proliferation.

Quickstep-----<<uses>>-----> Location

The Quickstep class uses the Location class to determine the player's movement during combat. This design follows the Dependency Inversion Principle (DIP) of OOP by depending on an abstraction (Location class) instead of a concrete implementation.

This allows for flexibility and easy modifications in the future, as any class that implements the Location interface can be used instead.

In terms of extension, the Quickstep class can be extended to include different types of movement techniques, such as dodging, rolling, or backstepping, by creating additional classes that implement the Location. The Location class itself can also be extended to include more methods that cater to different types of movements. One potential drawback of this design is that it may be overly complex for a simple movement mechanism. It may also require additional classes and interfaces to be created, which can lead to a larger codebase and potential maintenance issues. However, the benefits of flexibility and scalability may outweigh these concerns

Player carrying weapons:

Pros:

-Encapsulates the player's inventory and makes it easy to manage.

-Allows for a clear relationship between the player and their weapons.

-Provides a clear path for future expansion if more types of weapons are added.

Cons:

-Can become unwieldy if the inventory becomes too large.

-Might need to be refactored if a new type of weapon doesn't fit the current inheritance hierarchy.

Uchigatana and Great Knife using Unsheathe and Quickstep:

Pros:

-Allows for unique and specific functionality for each weapon. Encapsulates weapon-specific logic, making it easy to manage.

-Allows for clear and concise code, making it easy to read and maintain.

Cons:

-Can be difficult to manage if more weapons and skills are added in the future.

-May need to be refactored if a weapon does not fit the current inheritance hierarchy.

Club, Uchigatana, and Great Knife extending WeaponItem:

Pros:

-Encapsulates weapon-specific logic, making it easy to manage.

-Allows for easy addition of new weapons.

-Provides a clear inheritance hierarchy, making it easy to understand.

Cons:

-May become unwieldy if the inheritance hierarchy becomes too deep.

-May need to be refactored if a new type of weapon does not fit the current inheritance hierarchy.

Quickstep using Location:

Pros:

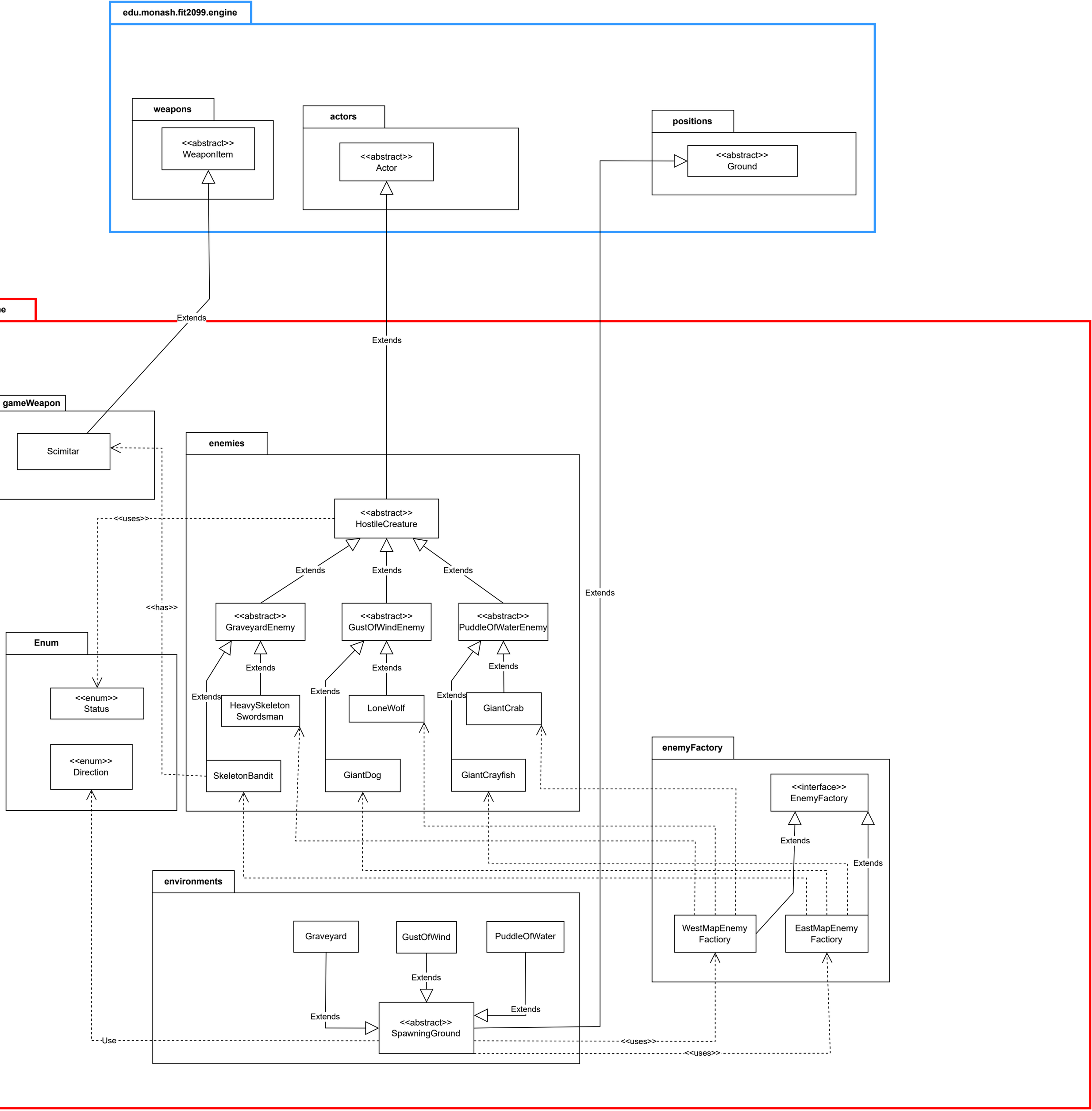
-Encapsulates location-specific logic, making it easy to manage.

-Allows for easy expansion if more location-based skills are added.

-Provides a clear separation of concerns between the location and the skill.

Cons:

May need to be refactored if the location logic changes. Can become difficult to manage if too many location-based skills are added


edu.monash.fit2099.engine
weapons
<<abstract>>
WeaponItem
actors
<<abstract>>
Actor
positions
<<abstract>>
Ground
game
Extends
Extends
Extends
gameWeapon
Scimitar
enemies
<<abstract>>
HostileCreature
Extends
Extends
Extends
<<uses>>
<<has>>
<<abstract>>
GraveyardEnemy
<<abstract>>
GustOfWindEnemy
<<abstract>>
PuddleOfWaterEnemy
Enum
Extends
Extends
Extends
Extends
Extends
Extends
<<enum>>
Status
<<enum>>
Direction
HeavySkeleton
Swordsman
LoneWolf
GiantCrab
SkeletonBandit
GiantDog
GiantCrayfish
enemyFactory
<<interface>>
EnemyFactory
Extends
Extends
environments
WestMapEnemy
Factiory
EastMapEnemy
Factiory
Graveyard
GustOfWind
PuddleOfWater
Extends
Extends
Extends
<<abstract>>
SpawningGround
Use
<<uses>>
<<uses>>

***Changes are 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.