ASSIGNMENT 1

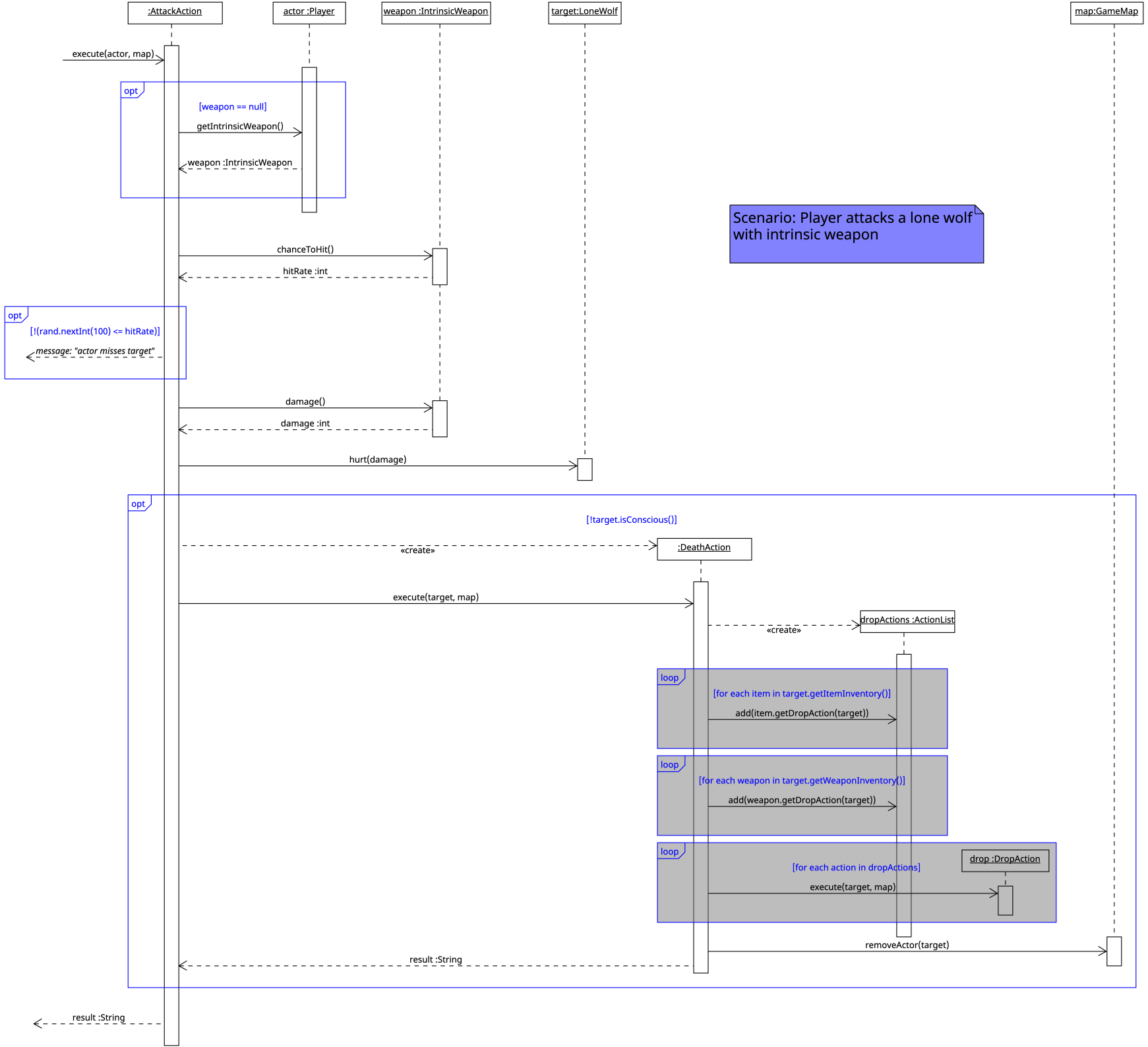
By Lab11Team1

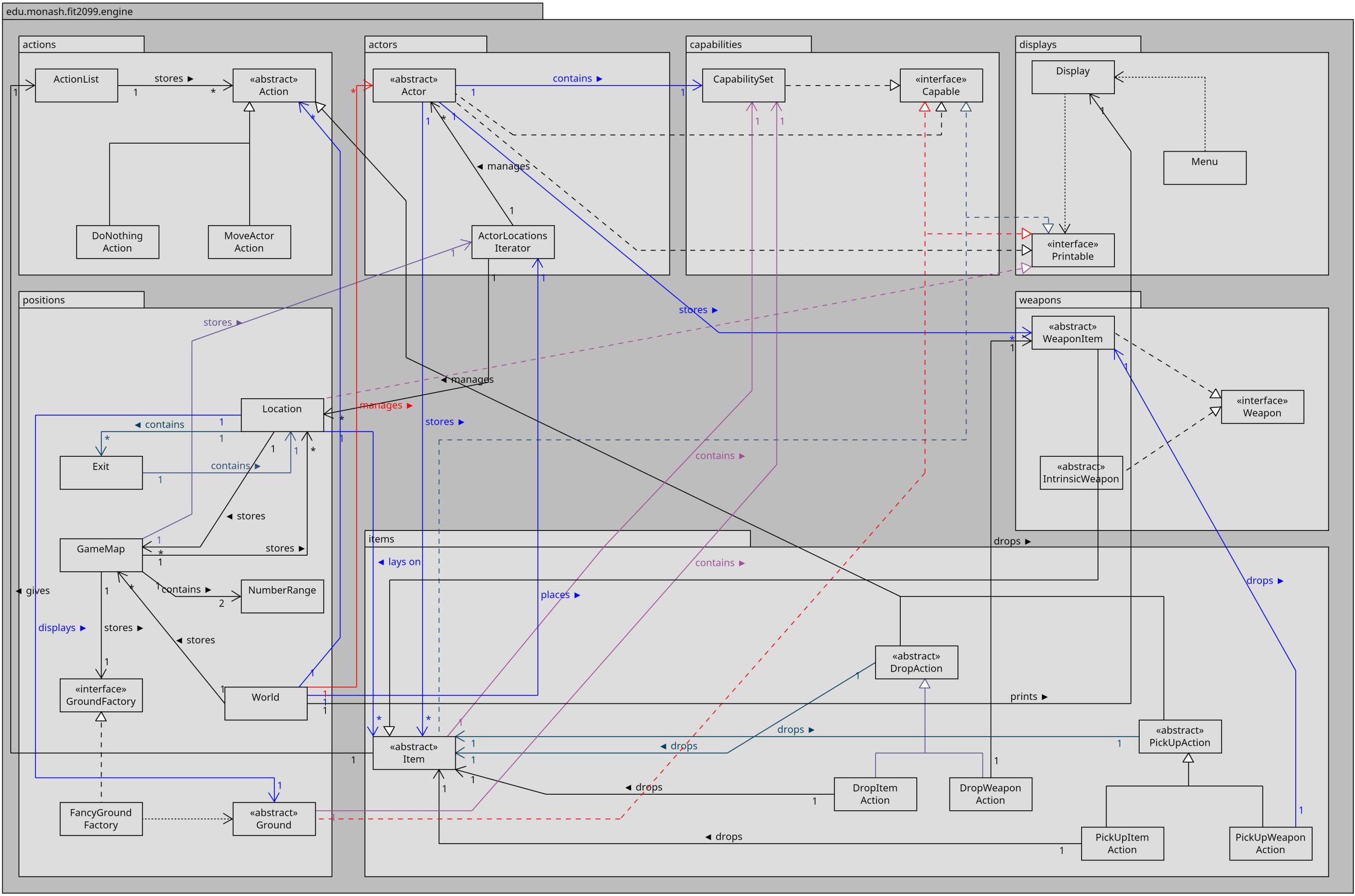
Harshath Muruganantham, Ziheng Liao, Ho Seng

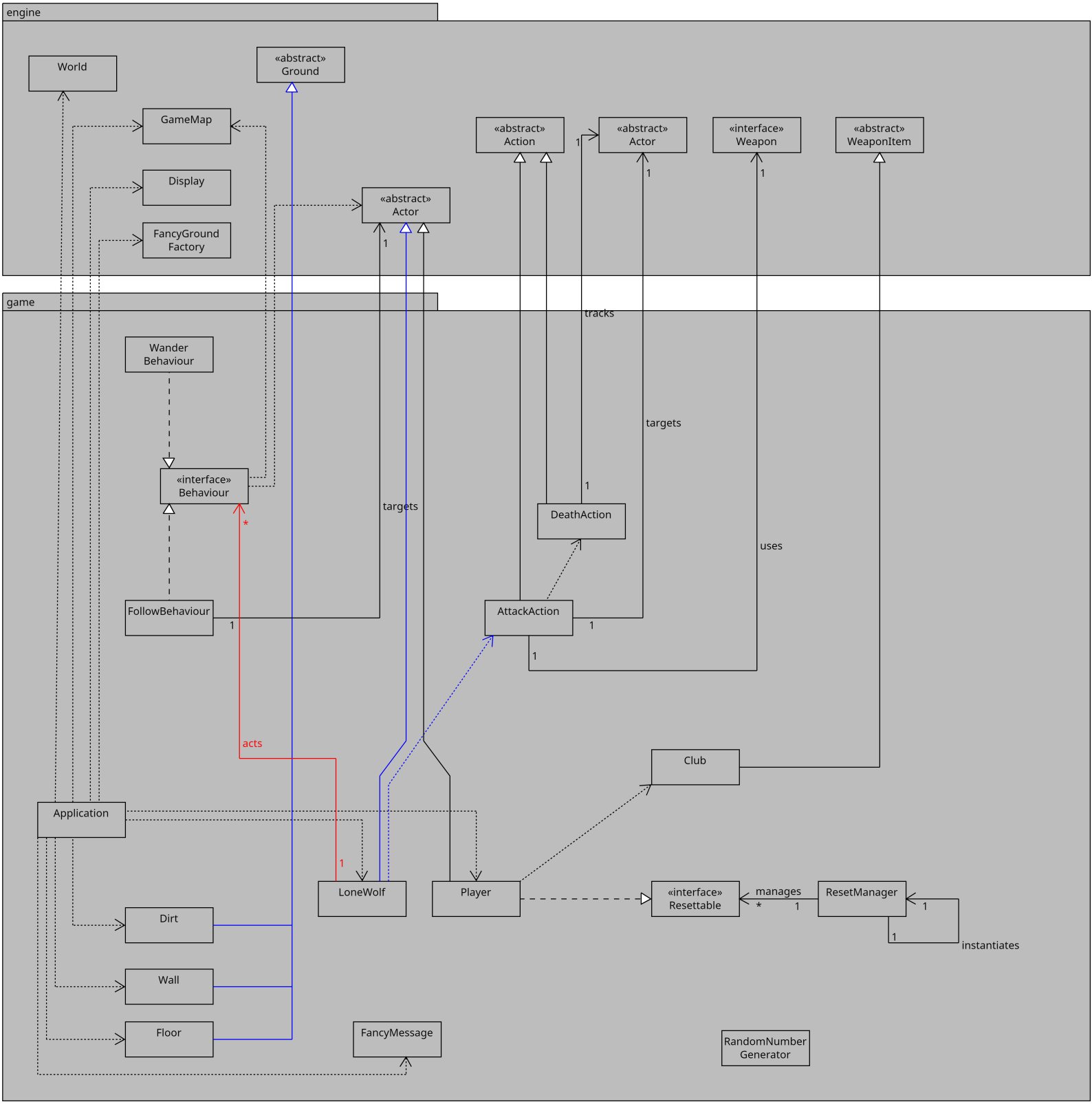
Lab11Team1

Harshath Muruganantham
Ziheng Liao
Ho Seng

Given UMLs


:AttackAction
actor :Player
weapon :IntrinsicWeapon
target:LoneWolf
map:GameMap
execute(actor, map)
opt
[weapon == null]
getIntrinsicWeapon()
weapon :IntrinsicWeapon
chanceToHit()
hitRate :int
opt
[!(rand.nextInt(100) <= hitRate)]
message: "actor misses target"
damage()
damage :int
hurt(damage)
Scenario: Player attacks a lone wolf with intrinsic weapon
opt
[!target.isConscious()]
«create»
:DeathAction
execute(target, map)
«create»
dropActions :ActionList
loop
[for each item in target.getItemInventory()]
add(item.getDropAction(target))
loop
[for each weapon in target.getWeaponInventory()]
add(weapon.getDropAction(target))
loop
[for each action in dropActions]
drop :DropAction
execute(target, map)
removeActor(target)
result :String
result :String

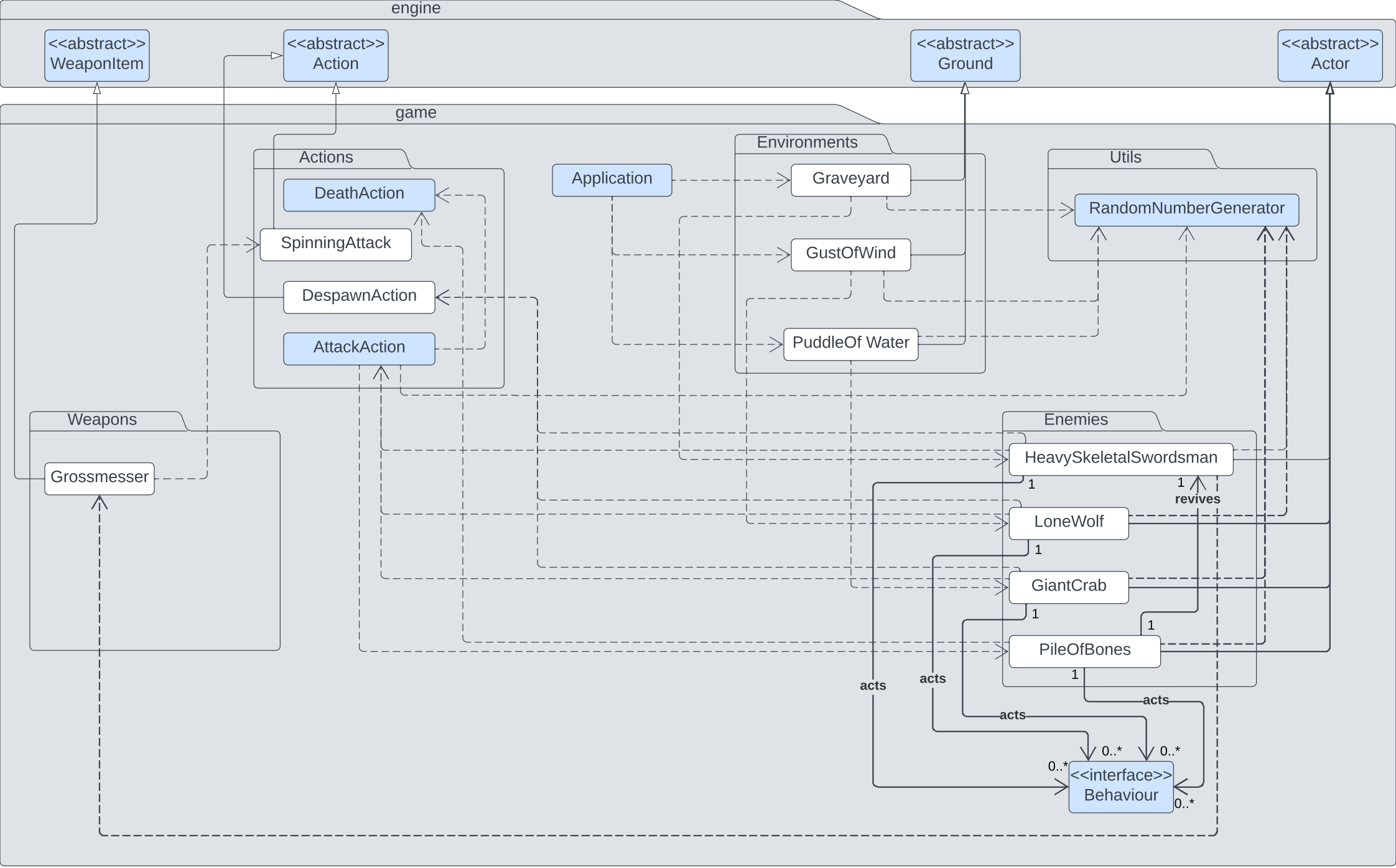

edu.monash.fit2099.engine
actions
ActionList
stores ▶
«abstract»
Action
DoNothing
Action
MoveActor
Action
actors
«abstract»
Actor
contains ▶
◀ manages
ActorLocations
Iterator
capabilities
CapabilitySet
«interface»
Capable
displays
Display
Menu
«interface»
Printable
positions
Location
◀ contains
Exit
contains ▶
GameMap
◀ stores
stores ▶
◀ gives
displays ▶
NumberRange
«interface»
GroundFactory
World
FancyGround
Factory
«abstract»
Ground
manages ▶
weapons
«abstract»
WeaponItem
«interface»
Weapon
«abstract»
IntrinsicWeapon
items
◀ lays on
places ▶
drops ▶
◀ drops
«abstract»
Item
«abstract»
DropAction
«abstract»
PickUpAction
prints ▶
DropItem
Action
DropWeapon
Action
PickUpItem
Action
PickUpWeapon
Action


engine
World
«abstract»
Ground
GameMap
Display
FancyGround
Factory
«abstract»
Actor
«abstract»
Action
«abstract»
Actor
«interface»
Weapon
«abstract»
WeaponItem
1
1
1
1
game
tracks
Wander
Behaviour
targets
«interface»
Behaviour
*
targets
1
DeathAction
uses
FollowBehaviour
1
AttackAction
1
1
acts
Club
Application
1
LoneWolf
Player
«interface»
Resettable
manages
*
1
ResetManager
1
1
instantiates
Dirt
Wall
Floor
FancyMessage
RandomNumber
Generator

Lab11Team1

Harshath Muruganantham
Ziheng Liao
Ho Seng

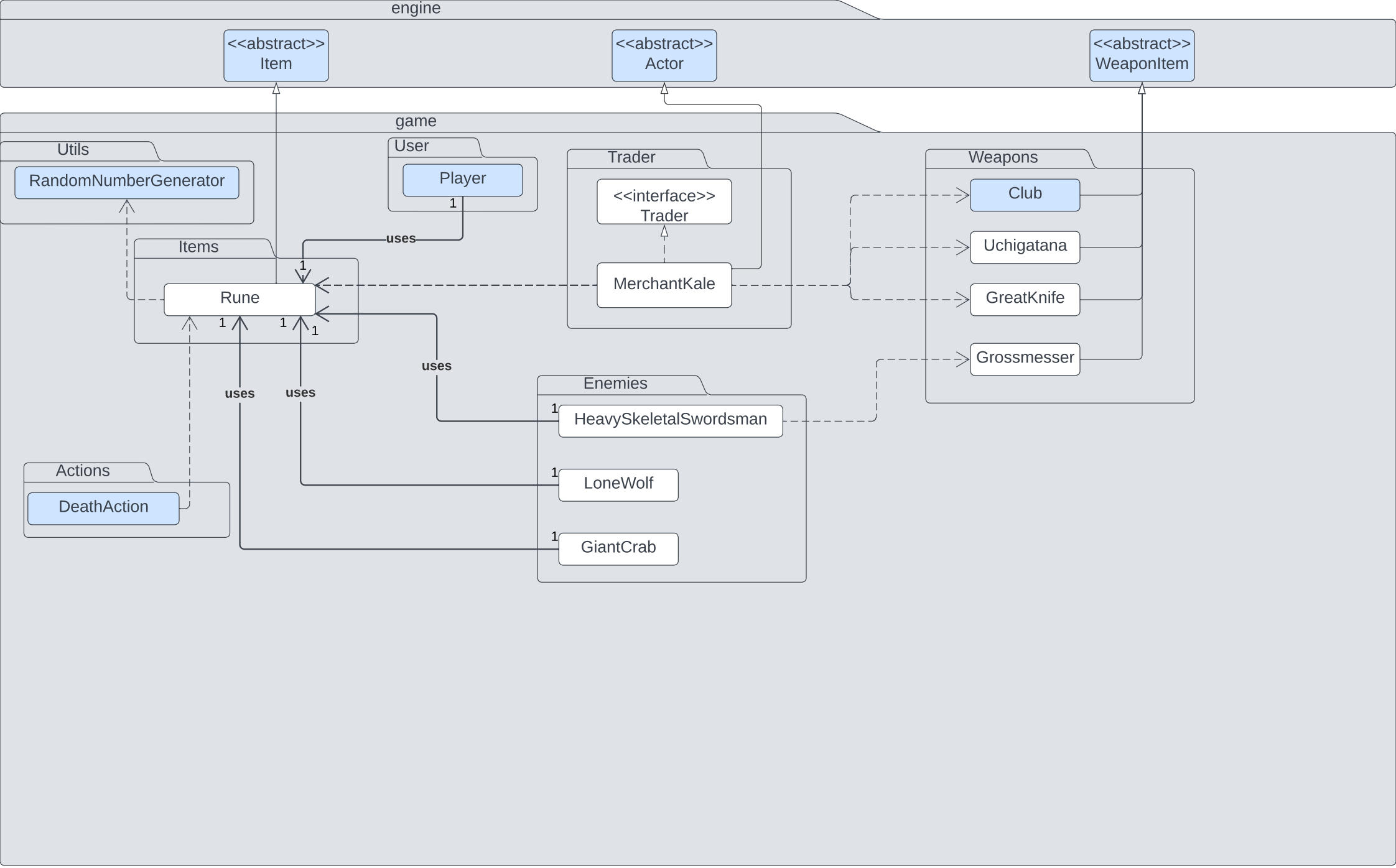
Group's UMLs



Req 1
Some relationships between entities shown on previous UML diagrams in this document are not repeated here.

OLD Classes are represented using the colour of this box.

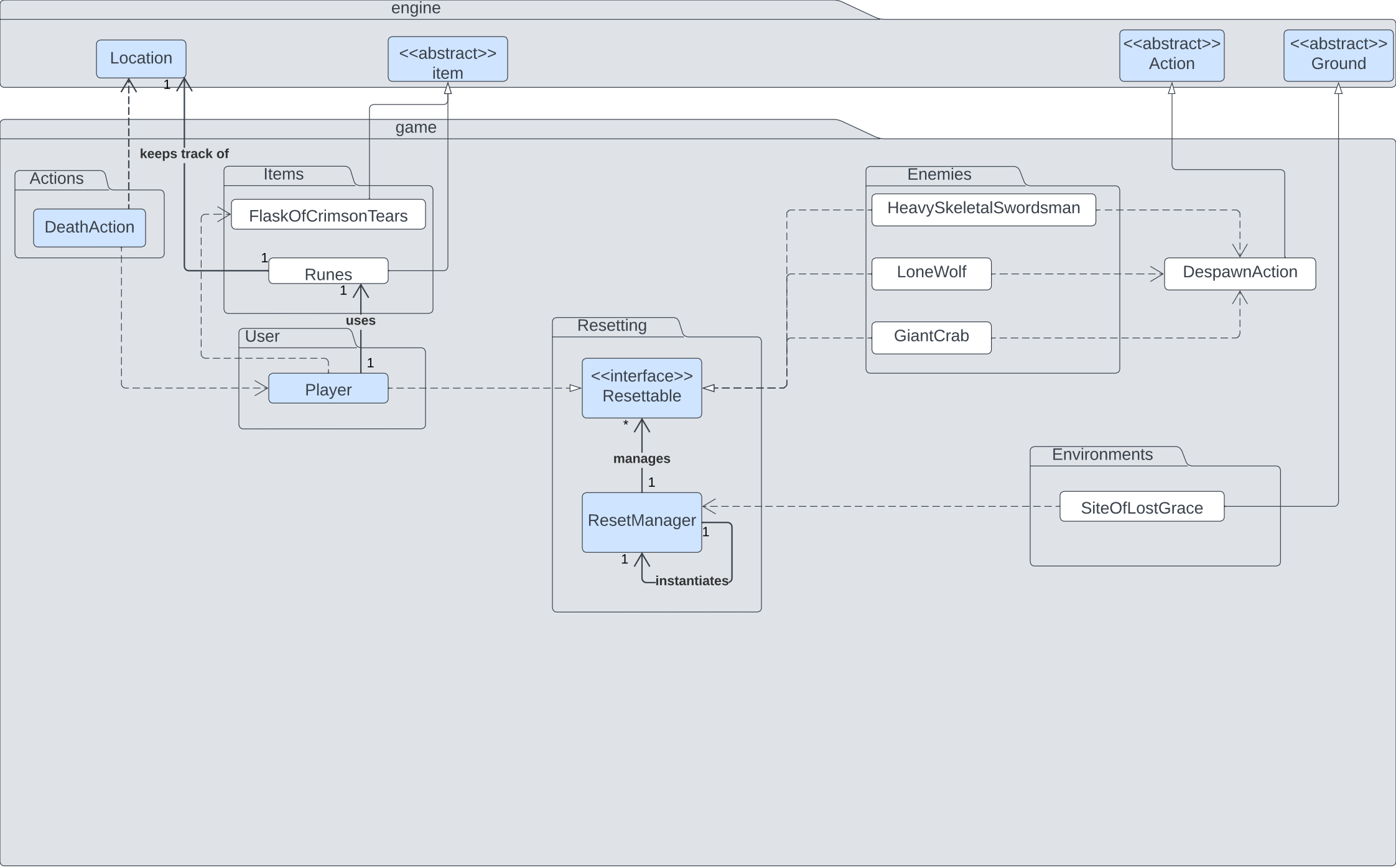
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



Req 2
Some relationships between entities shown on previous UML diagrams in this document are not repeated here.

OLD Classes are represented using the colour of this box.

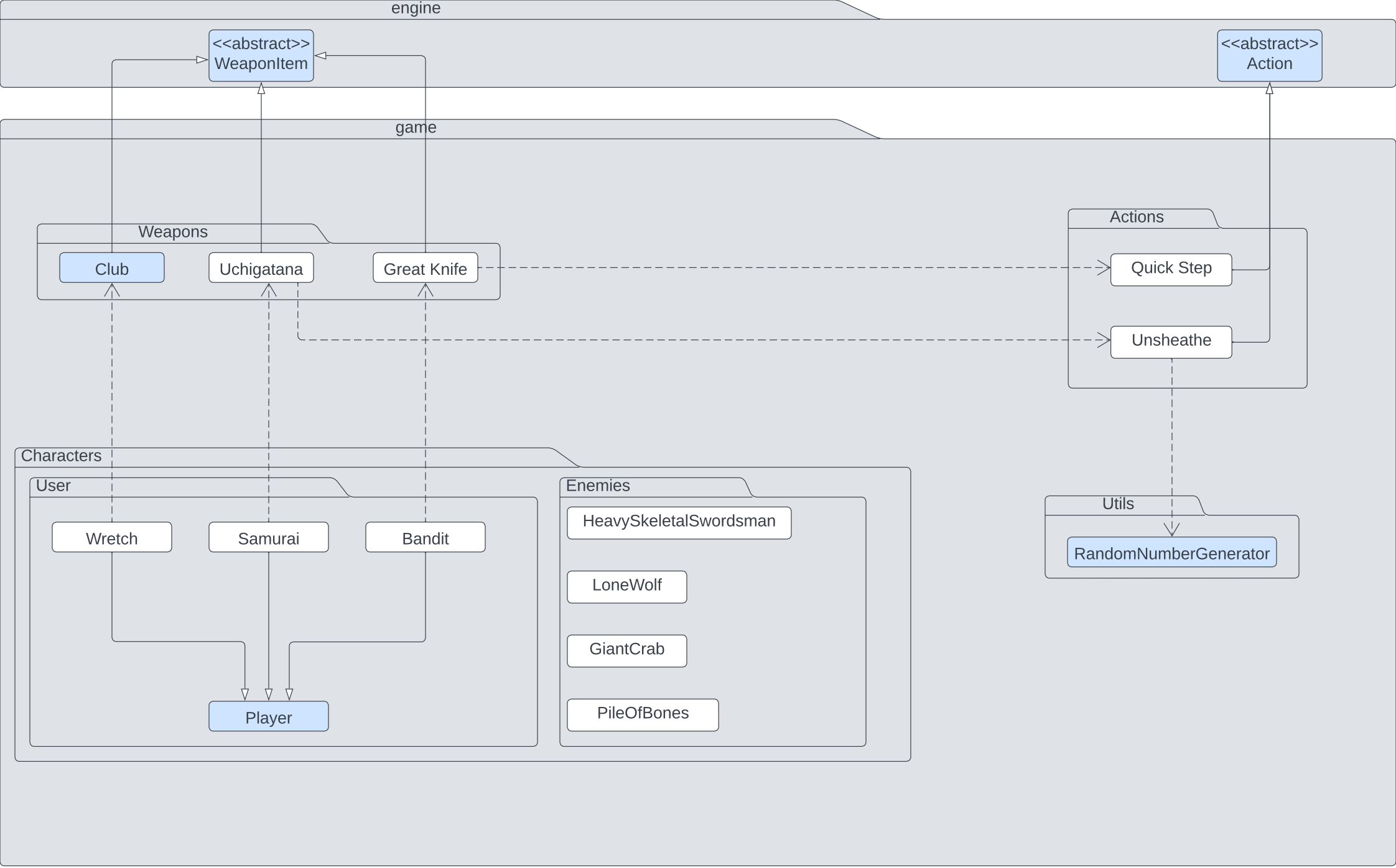
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



Req 3
Some relationships between entities shown on previous UML diagrams in this document are not repeated here.

OLD Classes are represented using the colour of this box.

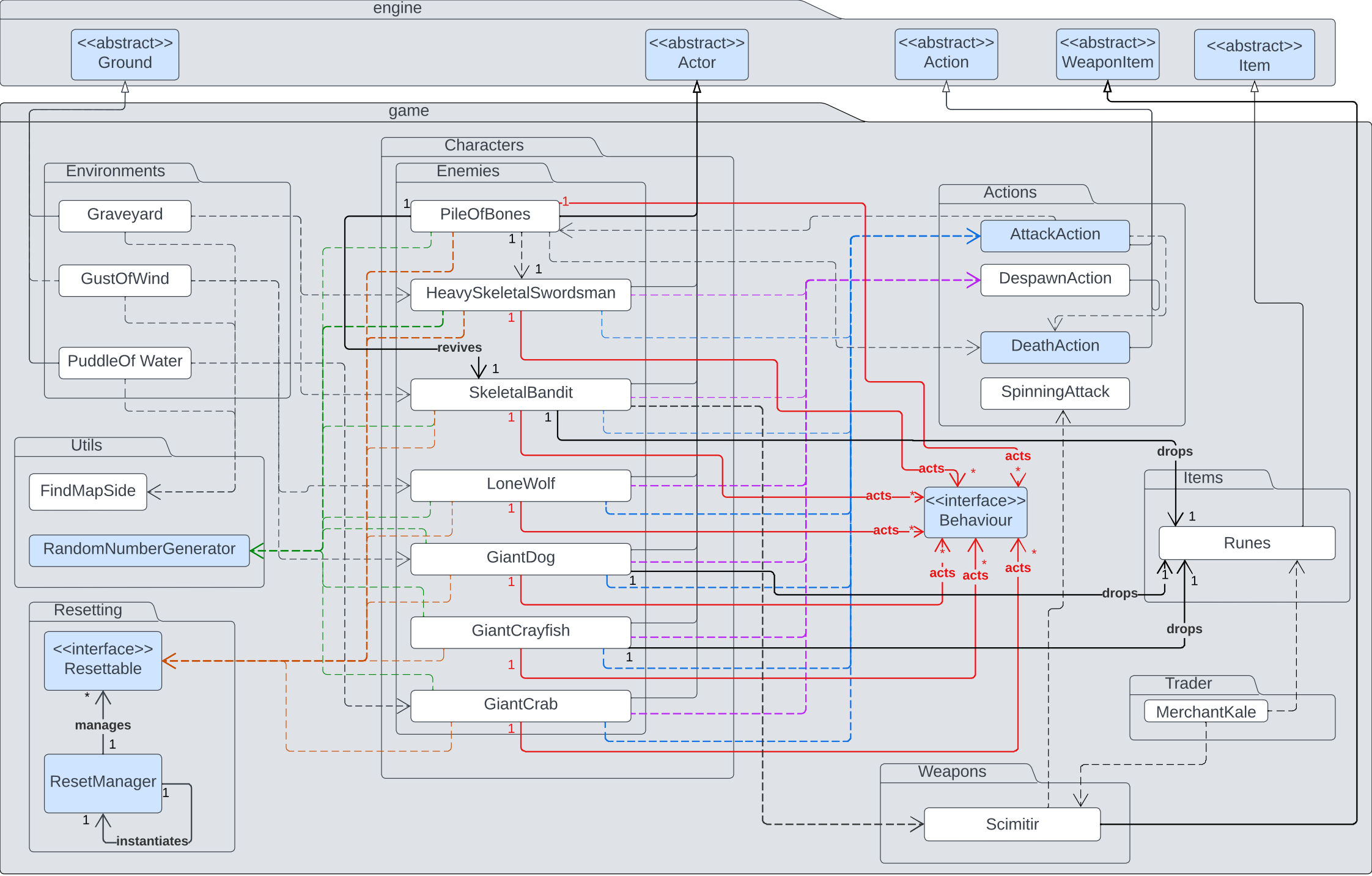
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



Req 4
Some relationships between entities shown on previous UML diagrams in this document are not repeated here.

OLD Classes are represented using the colour of this box.

CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



Req 5
Some relationships between entities shown on previous UML diagrams in this document are not repeated here.

OLD Classes are represented using the colour of this box.

CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng

Lab11Team1
Harshath Muruganantham
Ziheng Liao
Ho Seng

Design Rationale

Req 1

The diagram shows 9 new classes being added. Noticeable classes are the new enemy classes, new environment classes and Spinning Attack and Despawn Action. Some goals aimed to achieve whilst completing the implementation are:

Design Goals:

1. The new enemy characters added can re-use the old code already present and would only require special abilities to be coded in.
2. The new environments added can re-use the old code already present and would only require special abilities to be coded in.

The new enemy classes, “Heavy Skeletal Swordsman”, “Lone Wolf”, “Giant Crab” and “Pile of Bones” sit within an Enemies package inside the game package. This is done to separate the different categories of classes and to have a logical separation for ease of maintenance in future. All the four new enemy classes extend from the abstract Actor class. Since they share common attributes and methods, it is logical to abstract these identities to avoid repetitions (DRY). This also helps meet design goal 1. All new enemies have a behaviours list, suggesting any class that implements the “Behaviour” interface can be added to the behaviours list (LSP) (DIP). The enemies have a capability to attack based on a chance generated through “Random Number Generation”, explaining the enemies’ dependency on “Attack Action” and “Random Number Generation”.

When a “Heavy Skeletal Swordsman” dies, it becomes a “Pile of Bones”, which if not killed within 3 turns can revive back as “Heavy Skeletal Swordsman”. A decision was made to make “Pile of Bones” an extension of the “Actor” abstract class instead of the “Action” abstract class, due to the many similarities “Pile of Bones” shares with an “Actor”. Much like an “Actor”, the “Pile of Bones” has a specific location is stored at. However, like an “Action” the “Pile of Bones” uses a similar “tick” method to calculate the number of turns that has taken place. In the end, “Pile of Bones” was made to be an extension of the “Actor” class as it shares more similarities to “Actor” such as having a location to spawn and undergoing death action, as opposed to making it an extension to “Action” class. This allows us to meet the DRY principle and meet design goal 1, as this way we would need to repeat similar code fewer times as opposed to making “Pile of Bones” an extension of “Actor” class.

Looking at “Heavy Skeletal Swordsman” in specific, this class has a specific weapon in its “weapons list” attribute, an attribute that all Actors have (OCP). This weapon is a new class created called “Grossmesser”. Grossmesser inherits from the abstract class “Weapon Item” since they share common attributes and methods, and it is logical to abstract these identities to avoid repetitions (DRY). Grossmesser can perform a special attack called “Spinning Attack”. Since “Spinning Attack” is a special type of action that can be performed by the sword, it is extended from the abstract class “Action” to avoid code repetitions (DRY). Through creating Grossmesser as its own class and not creating it within the constructor of a heavy skeletal swordsman, we ensure that the Heavy Skeletal swordsman can use a Grossmesser without knowing what it is exactly (following Object-oriented principles) and so that a Grossmesser

can be used with by other Actors as well without having been implemented in each of the other Actor's constructor (following DRY principles).

The new environments, "Graveyard", "Gust of Wind" and "Puddle of Water" all extend from the abstract class "Ground" since they share common attributes and methods, and it is logical to abstract these identities to avoid repetitions (DRY) and helping us meet design goal 2. We avoided creating a single class containing the characteristics of all three environments, and rather decided to split them into three different classes in order to avoid creating a god class. Each of these three environments have different capabilities and spawn different enemies, so they had to be split off into different classes to follow SRP.

A decision was made to create a new class called "Despawn Action", that inherits from the action abstract class rather than to combine the de-spawning method with the "Death Action" class. This decision was made in order to better follow the OOP principles such as SRP, since death action-method requires the loot of an enemy (such as runes) to be dropped, whilst Despawn action-method requires loot to not to be dropped, and the enemy to just vanish from map.

Lab11Team1 Harshath Muruganantham
Ziheng Liao
Ho Seng

Req 2

The diagram shows 7 new classes being added. Noticeable classes are Merchant Kale, Rune, and more weapons. Some goals we will aim to achieve whilst completing the implementation are:

Design Goals:

1. To be able to add new traders without needing to modify existing code (Open/Close principle (OCP))
2. Runes should re-use as much of the old code as possible and shouldn't require break the existing functionality of the game.
3. New weapons should re-use old code as much as possible.

Rune is its own class as it is a currency of the game. It has its own functionalities which are similar to that of an item. Noticeable functionalities are that of it being able to be dropped and picked up. Therefore, the inheritance of the abstract class "Item" was chosen to avoid repetition of code (DRY). The alternative was considered where it wouldn't be inheriting from the class "Item" and functionalities such as being able to be picked up and dropped would be added into the Rune class itself. The benefits of this second method include that a rune would not be dropped by the drop items action when an enemy dies and is killed by another enemy and an enemy will not be able to pick up the rune when it kills another enemy character. However, this design was ultimately omitted as these were the only two scenarios where it would be disadvantageous for a Rune to be an "Item" object, and there are more scenarios where it would be advantageous for a Rune to be an Item. The two scenarios explained above mainly occur in Death Action method, so choosing the first method would dictate that only code in Death Action would need to me amended instead of the many classes in the game. This helps us establish easy extensibility and maintainability and helps us meet design goal 2.

The class "Merchant Kale" inherits from the abstract class "Actor" due to the attributes and methods in Actor will be used in Trader, such as itemInventory and weaponInventory can be re-used to store all inventory that the trader can sell. Like before, rewriting the same code elsewhere is bad design, so this method allows us to maintain the DRY principle. It also implements the interface "Trader", which contains methods such as selling and buying (which all traders must do). As all traders will have the same functionality and will need to implement the same functions, having this interface allows us to create new unique traders much more easily (ISP) and use this trader interface to address all traders. (LSP)

Similarly, the different types of weapons such as "Uchigatana", "Great Knife" and "Grossmesser" inherit from the abstract class "Weapon Item" as the purpose here is to take advantage of the Open/Close principle implemented from the previous developer of this code. This allows us to use methods (add item, store item, etc.) in other classes (like "Actor") without the other actors knowing specifically what Item is being stored, helping us follow Encapsulation laws. This is because, some classes (like Actor) use "Weapon Item" to dictate the type of objects in stored lists rather than mention specific weapon items. Hence, this method will allow us to re-use a lot of the old code, helping us follow DRY principle and meet Design goal 3. Furthermore, it prevents redundant code and follows DRY principles as each weapon has a hit Rate, damage, and the ability to be dropped and picked up.

Harshath Muruganantham
Ziheng Liao
Ho Seng

Req 3

The diagram shows 2 new classes being added. Noticeable classes are the Flask of Crimson Tears, and Site of Lost Grace. Some goals aimed to achieve whilst completing the implementation are:

Design Goals:

1. To add extra functionalities whilst reusing as much of the code already present in engine and game as possible.
2. Make sure that adding any classes does not break the current functionality of the game.

"Flask of Crimson Tears" extends from "Item" abstract class since they share common attributes and methods, and it is logical to abstract these identities to avoid repetitions (DRY). Making "Flask of Crimson Tears" a subclass of the Item class allows us to take advantage of the functionality of the Item class to implement the adding of the item to the Player's inventory and to allow for further modification in the future. Although implementing the Flask as a subclass of the Item class will allow the Flask to be dropped as it is an item and the Item class has the DropItemAction method, the functionality gained from the Item class is greater and we can simply override the DropItemAction method in the Flask subclass to not drop the item upon Player death or rest at Site of Lost Grace. This method was also chosen instead of having "Flask of Crimson Tears" as just an attribute of the player class to improve upgradeability of code in the future. Through making the "Flask of Crimson Tears" a separate object, we better adhere to object-oriented programming principles, as we now allow more than one character (i.e., player) to be able to use the "Flask of Crimson Tears" without specifically knowing what it does or what it is. This allows us to ensure that in future revision of the game, more characters can use "Flask of Crimson Tears" without the need to change existing code (OCP). This also allows us to assign a specific purpose to "Flask of Crimson Tears" helping us better follow SRP and meet design goal 2.

Resettable interface will be implemented in all classes that needs to be resettable (ISP). Currently, this includes the "Player" class, and all the classes in the Enemies package. We use this interface as we can customise the result of the Reset method to be specific to the Class that is being reset. Since the Player class and the Enemy Actor classes need to be reset in different ways, this interface acts as a way to let us know this object needs to be reset but allows the particular object to dictate how it can be reset, perfectly fitting OOP principles. The Reset Manager class will be used to store a list of all resettable classes to ensure all the resettable classes are reset upon Player death or Player's rest at Site of Lost Grace.

The "Site of Lost Grace" extend from the abstract class "Ground" since they share common attributes and methods, and it is logical to abstract these identities to avoid repetitions (DRY). The "Site of Lost Grace" has a dependency with "Reset manager", as when a player enters the "Site of Lost Grace", the player will be given an option to reset the game. "Reset Manager" class contains an attribute, an Array List of classes implementing the "Resettable" Interface (LSP) (DIP). If the player decides to exercise the option of resetting the game, the list of all resettable in "Reset Manager" will be inputted into a loop and each object in that list will be reset. Enemies, upon spawning, will be added to this Array List of they are Resettable. Through this method, OOP principles are followed as Reset Manager isn't made aware of any particular

objects and only that the objects implement the method “Resettable”, and the “Site of Lost Grace” is not made aware of any particular objects in the game when resetting, allowing for Encapsulation. This particular method was chosen as opposed to storing all objects in an Array List in “Site of Lost Grace” and manually checking and resetting each object. The proposed second method does not follow proper OOP principles as the “Site of Lost Grace” is made aware of all Actor objects in the game, which breaks Encapsulation laws and the game would have to employ very similar code to be executed in “Death Action” when a player dies, which creates unnecessary code repetition breaking DRY principles.

When a player dies, runes are dropped in the player’s previous location. After adding the previous location’s “Location” as an attribute to a “Rune” object and then game is reset, and the “Rune Object” is dropped in the player’s previous location before death (currently stored as an “Location” attribute in the Rune). Doing this allows us to meet one of our design goals of reusing given code as much as possible. The “Location” class is already present in the Engine package and fits our need perfectly. So, this class is reused in order to avoid repetition of code and follow DRY principles. More implementation details follow here:
If the player visits the Rune’s dropped location again, they can pick up the rune. If the player undergoes “Death Action” (not resetting at “Site of Lost Grace”), “Death Action” method accesses at the player's rune “if dropped” attribute (newly created; set to true right after the Rune is dropped in “Death Action”). If that attribute is false (which it shouldn’t be), nothing happens. If it is true, “Death Action” accesses the Rune’s location attribute, and employs the remove item method and removes the rune object from this location. Since the Rune is extended from the abstract class “Item”, the rune will contain the remove item method within itself, sparing us the trouble of coding that in, and helping us adhere to the OOP principle of DRY (Don’t Repeat Yourself). This also allows us to achieve or design goal of reusing given code as much as possible. This method is chosen, instead of creating a new class for representing a dropped rune, as this method allows us to reuse as much of the existing code as possible, helping us meet our design goal and adhere to DRY principles.

Harshath Muruganantham
Ziheng Liao
Ho Seng

Req 4

NOTE: Player classes is referring to Samurai, Bandit or Wretch and not the class "Player".
The diagram shows the player taking on 3 new player classes and 2 new weapons classes new unique skills being added to the game. Some goals we will aim to achieve whilst completing the implementation are:

Design Goals:

1. The new Weapons classes added can re-use the old code already present and would only require special abilities to be coded in.
2. The new Player classes added can all reuse the old code already present and would only require special abilities to be coded in.
3. The new Player classes can work with existing code.

The classes, "Wretch", "Samurai", and "Bandit" all extends from the Player class. Since these new classes modify the hit points and the starting inventory of the Player, we extend them from the Player class as this allows for existing code to be reused, meeting our design goals 2 and 3, and the DRY principle. This approach was chosen over just modifying the "Player" class each time one of the new characters was chosen in order to better follow the SOLID principle of SRP and to ensure ease of code maintenance and code simplicity.

Each of these new player classes has a specific weapon type added to their inventory upon summon, as showed in the UML. These weapons all extend from the "Weapon Item" abstract class since they share common attributes and methods, and it is logical to abstract these identities to avoid repetitions (DRY). Looking at "Uchigatana" and "Great Knife" in particular, they each have the ability to perform a special action. Since "Uchigatana" and "Great Knife" both extend from "Weapon Item", they both have the GetSkill() method, that has the ability to gets the Action from the weapon. So, we can simply add their special actions to the unique Weapon. This simplifies implementation as we are able to use existing methods to get the implementation that we need, helping us better follow DRY principle.

The new abilities discussed in the previous paragraph are "Quick Step" and "Unsheathe". "Quick Step" and "Unsheathe" but extend from the "Action" abstract class. This is done because the weapons that use these abilities have a special attribute in their parent class that allows us to store special abilities as "Action" class attributes. Therefore, extending "Quick Step" and "Unsheathe" from the Action class, allows us to use existing code in the engine package, helping us meet our design goal number 1, and follow DRY principle and OCP. If we had instead created "Quick Step" and "Unsheathe" as their own classes, not extending from the "Acton" abstract class, we would have had to add in new code to the weapons specifically, and we would not be able to follow the OCP principle, as if another weapon in the future needed to use these abilities, it would also have to be coded in specifically in that weapons constructor, which is undesirable.

Harshath Muruganantham
Ziheng Liao
Ho Seng

Req 5

This UML diagram shows the addition of new enemy characters, new weapon items and adds a new twist to the side of map an enemy character can be spawned. Some goals aimed to achieve whilst completing the implementation are:

Design Goals:

1. The new enemy characters added can re-use the old code already present and would only require special abilities to be coded in.
2. The new Weapons classes added can re-use the old code already present and would only require special abilities to be coded in.

The new enemy classes added, “Skeletal Bandit”, “Giant Dog” and “Giant Crab” all extend from the “Actor” abstract class since they share common attributes and methods, and it is logical to abstract these identities to avoid repetitions (DRY) and help us meet design goal 1. All new enemies have a behaviours list, suggesting any class that implements the “Behaviour” interface can be added to the behaviours list (LSP) (DIP). The enemies have a capability to attack based on a chance generated through “Random Number Generation”, explaining the enemies’ dependency on “Attack Action” and “Random Number Generation”.

Similar to requirement 1 design rationale, when a “Skeletal Bandit” dies, it becomes a “Pile of Bones”, which if not killed within 3 turns can revive back as “Skeletal Bandit”.

Looking at “Skeletal Bandit” in specific, this class has a specific weapon in its “weapons list” attribute, an attribute that all “Actors” have (OCP). This helps us meet or design goal 2. This weapon is a new class created called “Scimitir”. Scimitir inherits from the abstract class “Weapon Item” since they share common attributes and methods, and it is logical to abstract these identities to avoid repetitions (DRY). Grossmesser can perform a special attack called “Spinning Attack”. Since “Spinning Attack” is a special type of action that can be performed by the sword, it is extended from the abstract class “Action” to avoid code repetitions (DRY). Through creating Scimitir as its own class and not creating it within the constructor of “Skeletal Bandit”, we ensure that the “Skeletal Bandit” can use a Scimitir without knowing what it is exactly (following Object-oriented principles) and so that a Scimitir can be used with by other Actors as well without having been implemented in each of the other Actor’s constructor (following DRY principles).

These new enemies also share the same mechanism of resetting as discussed in detail in requirement 3 design rationale.

Scimitir can be traded by “Merchant Kale” in a similar way as discussed in requirement 2 design rationale.

A new mechanism in this requirement, not previously in any requirements is the ability for the environment to spawn different enemies depending on the location that the enemy will be spawned in (i.e., east, or west). We have made a decision to implement this feature by creating a new class called “Find Map Side” in utils. A few implementation details on how this works follows. In each turn, the environment class has the ability to spawn a new enemy. We will first be using “Random Number Generator” to get a specific location to spawn an enemy in

terms of a grid slot in the map (aaXaa). This location return will be sent to “Find Map Side” class which would return if the specific location inputted is on the right or the left side of the map. Then depending on the string returned (“right” or “left”) a different animal will be spawned in that location. Doing it this way was chosen, rather than implementing a similar method to “Find Map Side” in each of the environment classes, as this follows the principle of DRY, allowing us to create clear and maintainable code.

Lab11Team1

Harshath Muruganantham
Ziheng Liao
Ho Seng

Contribution Log

Task/Contribution(~30 words)	Contribution type	Planning Date	Contributor	Status	Actual Completion Date	Extra notes	
Draft UML Diagram for Requirement 1	UML diagram	02/04/2023	EVERYONE	DONE	03/04/2023	We went over the code whilst completing the UML diagram. Total time taken to complete this was 3 hours. UML diagram did not have packages. Was a rough draft on the ideas that team members had.	
Draft UML Diagram for Requirement 2	UML diagram	03/04/2023	EVERYONE	DONE	04/04/2023	Went over code whilst completing UML. Discussed design options, particularly about the implementation of the Rune class. Discussion involved around whether Rune would inherit from Item class or not. Ultimately decided to inherit from Item. Light discussion over Trader took place as well.	
Draft UML Diagram for Requirement 3	UML diagram	04/04/2023	EVERYONE	DONE	12/04/2023	Further Discussion about UML diagram, particularly about how Runes should be implemented. Discussion revolved around how rune would be decided to be dropped and removed from the map. Ultimately decided on a Boolean attribute inside Rune to determine whether Rune should be removed or remain for Player to be picked up.	
Draft UML Diagram for Requirement 4	UML diagram	05/04/2023	EVERYONE	DONE	12/04/2023	Code was looked over as this Task was tied into how certain Classes were going to be implemented in Task 2 and 3. Discussion involved the implementation of weapon to each of the player classes (Samurai, etc). Briefly went over why Player could not inherit from the player classes. Concluded that Java would not allow such implementation and it would be a poor design choice.	
Draft UML Diagram for Requirement 5	UML diagram	10/04/2023	EVERYONE	DONE	12/04/2023	Short discussion as Task 5 was more about extending and modification without implementing anything new. Also decided on the packages that each class was going to fit into. Discussion also involved whether or not to include REQ 1 UML implementation. Decided to include it to show design decisions were consistent with REQ 1. As a result, coloured arrows were necessary for readability.	
Carefully note down the implementation details for Req 1 / Draft initial Design Rationale for Req 1	Discussion	02/04/2023	EVERYONE	DONE	03/04/2023	Whilst working on the UML diagram, we also thought about how we were going to implement it via code. A lot of time was spent on finding what methods each class had. Time was also spent on looking at the engine UML diagram to assist with a better understanding of the code. This thought process helped was intended to help with our design rationale later on.	
Carefully note down the implementation details for Req 2 / Draft initial Design Rationale for Req 2	Discussion	03/04/2023	Ho	DONE	04/04/2023	Majority of time was spent looking at the Item class to determine the methods that it had that would help with the implementation of Rune. As discussion revolved around deciding whether or not for Rune to inherit from Item, pseudocode was written to assist with deciding how it would be implemented both with and without the inheritence of Item.	
Carefully note down the implementation details for Req 3 / Draft initial Design Rationale for Req 3	Discussion	04/04/2023	EVERYONE	DONE	12/04/2023	Discussion mainly involved around the implementation of the interface Resettable. Most of the time was spent deciding how to reset the different Actors as Player do not despawn like the enemies. Also discussed whether or not DeathAction should be used as part of the reset function. The decision to have ResetManager as a dependency or an attribute to SiteOfLostGrace was lightly mentioned. Group unanimously agreed that it should be a dependency	

Task/Contribution(~30 words)	Contribution type	Planning Date	Contributor	Status	Actual Completion Date	Extra notes	
Carefully note down the implementation details for Req 4 / Draft initial Design Rationale for Req 4	Discussion	05/04/2023	Ho	DONE	12/04/2023	Discussion lasted 15 minutes. Group came quickly to an agreement on how to add new player classes. Since there weren't many possible options to add the player classes, we decided to have it inherit from player with its own attributes and weapon. Also concluded pretty quickly that the actions for each weapon should be a dependency.	
Carefully note down the implementation details for Req 5 / Draft initial Design Rationale for Req 5	Discussion	10/04/2023	Ho	DONE	12/04/2023	This requirement was thought to be extremely similar to that of requirement 1. Discussion did not revolve around design, but rather on whether or not to include the UML diagram of requirement 1. Eventually decided to include parts of requirement 1 diagram to show the different enemies that each environment spawns.	
Created Design Rationale for Req 1 following on from examples provided.	Design rationale	13/04/2023	Harshath	DONE	14/04/2023	For this design rationale, I took inspiration from the design rationale's provided in Assignment Spec sheet as well as week 5 class activitities. In this activitiy, I tried to incorporate many SOLID Principles as well as OOP priciples to justify our work.	
Created Design Rationale for Req 2 following on from examples provided.	Design rationale	13/04/2023	Ziheng	DONE	14/04/2023	Completed design rationale for this requirement with the goal of explaining the thought process behind the team's decision. Also tried including how our design follows design principles. Encountered problems with how to structure the explanation. Came to a conclusion as I kept working on it.	
Created Design Rationale for Req 3 following on from examples provided.	Design rationale	13/04/2023	Harshath	DONE	14/04/2023	For this design rationale, I took inspiration from the design rationale's provided in Assignment Spec sheet as well as week 5 class activitities. In this activitiy, I tried to incorporate many SOLID Principles as well as OOP priciples to justify our work.	
Created Design Rationale for Req 4 following on from examples provided.	Design rationale	13/04/2023	Harshath	DONE	14/04/2023	For this design rationale, I took inspiration from the design rationale's provided in Assignment Spec sheet as well as week 5 class activitities. In this activitiy, I tried to incorporate many SOLID Principles as well as OOP priciples to justify our work.	
Created Design Rationale for Req 5 following on from examples provided.	Design rationale	13/04/2023	Harshath	DONE	14/04/2023	For this design rationale, I took inspiration from the design rationale's provided in Assignment Spec sheet as well as week 5 class activitities. In this activitiy, I tried to incorporate many SOLID Principles as well as OOP priciples to justify our work.	
Combining of all Design Rationale parts into singular document	Design rationale	13/04/2023	Harshath	DONE	15/04/2023	Just made a final assignment submission template. Added design rationale, contribution log, and UML diagrams to this document	
Initial Review of Design Rationale	Design rationale	13/04/2023	Ziheng	DONE	15/04/2023	Read through the design rationale with the intent on picking up minor mistakes and also possible changes that could be added to improve the design rationale. Made a couple of edits and proposed an aesthetic change which was ultimately rejected.	
Review/Check Final Assignment 1A	Discussion	15/04/2023	EVERYONE	DONE	15/04/2023	Proof read by everyone. Made sure everybody was satisfied with the work that was about to be submitted. Eventually given the pass by everyone to submit.	