ASSIGNMENT 2

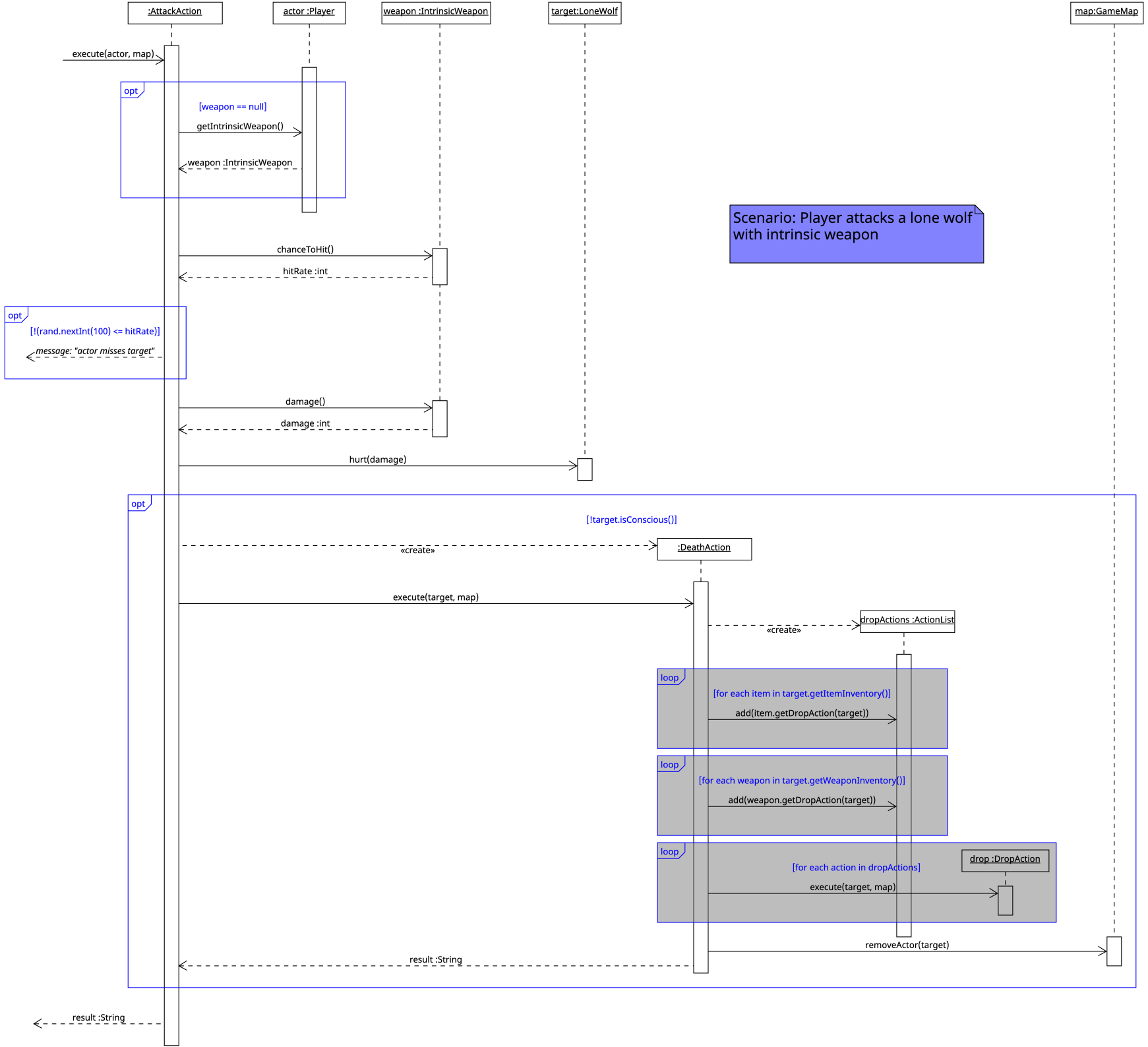
By Lab11Team1

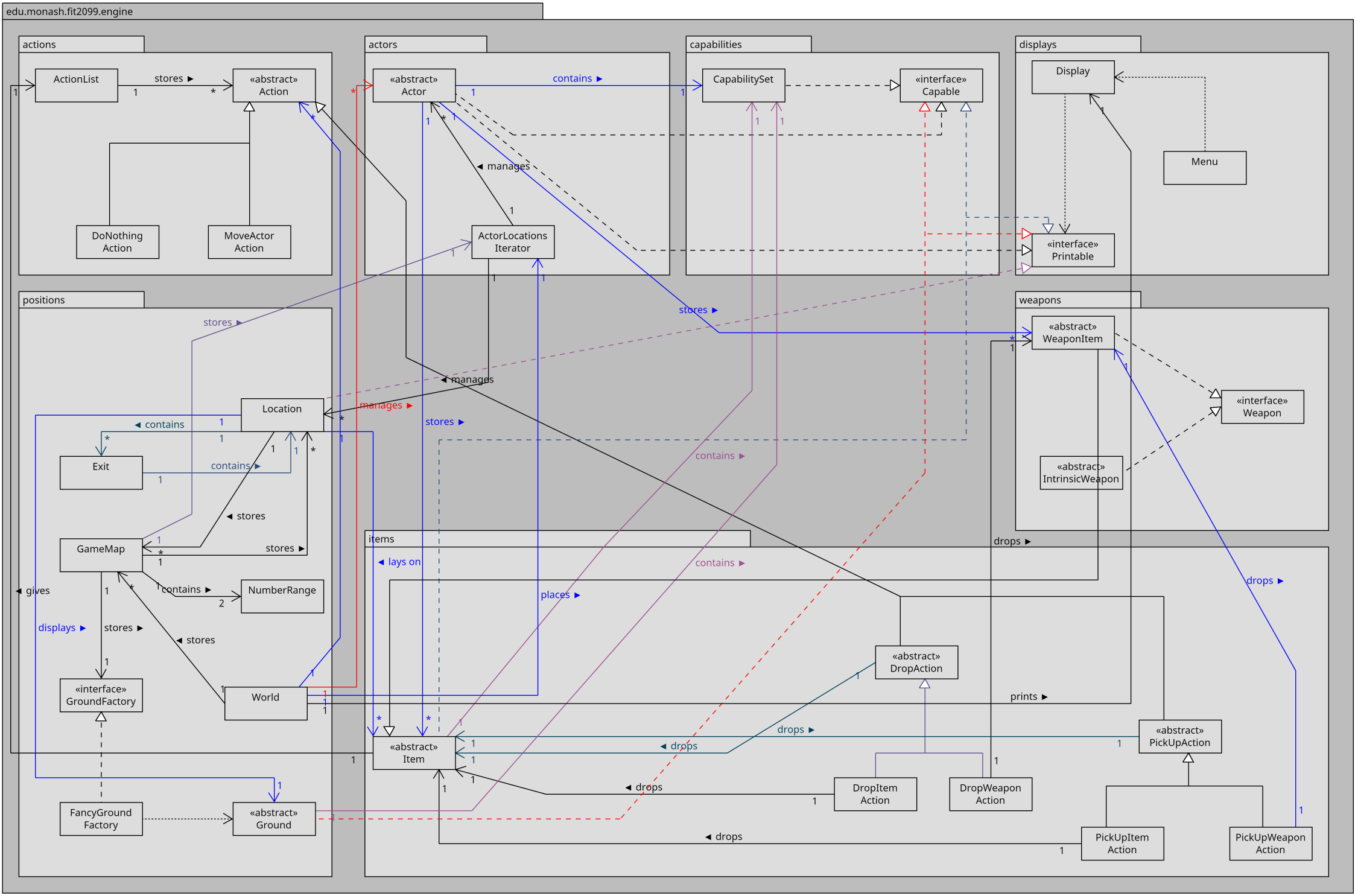
Harshath Murugananthamm, Ziheng Liao, Ho Seng

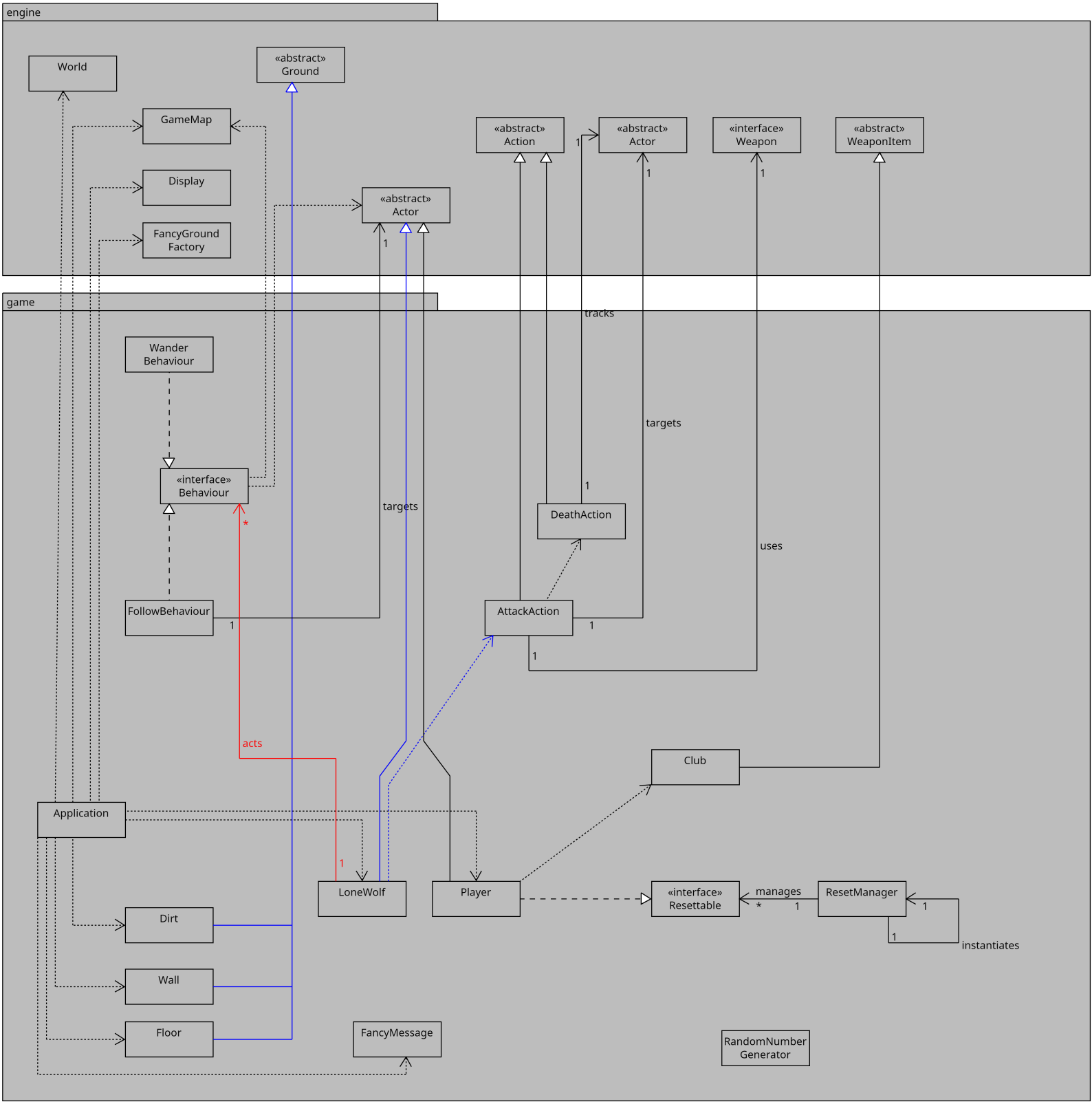
Lab11Team1

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

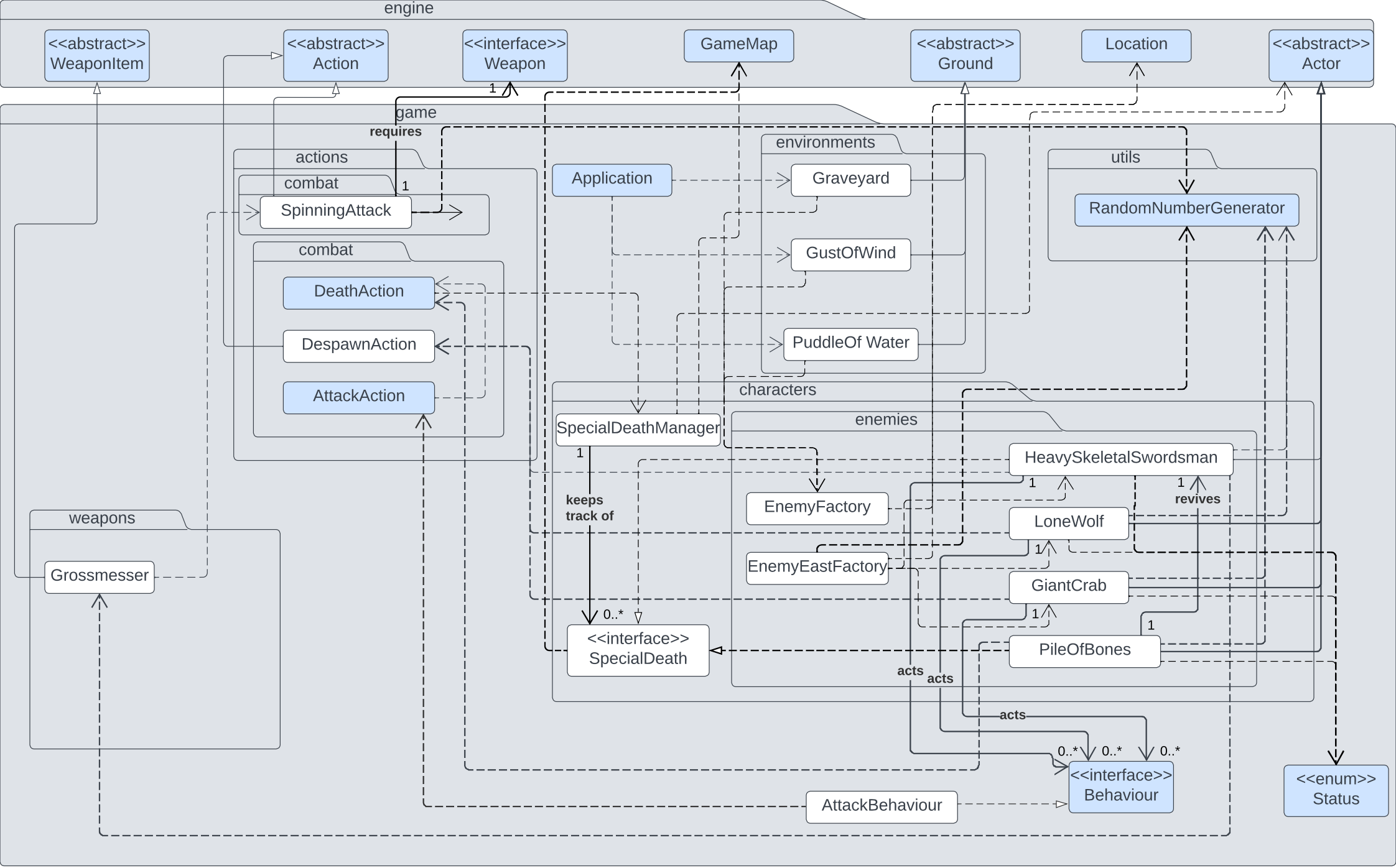

edu.monash.fit2099.engine
actions
ActionList
stores ▶
«abstract»
Action
DoNothing
Action
MoveActor
Action
actors
«abstract»
Actor
contains ▶
◀ manages
ActorLocations
Iterator
capabilities
CapabilitySet
«interface»
Capable
displays
Display
Menu
«interface»
Printable
positions
Location
◀ contains
Exit
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

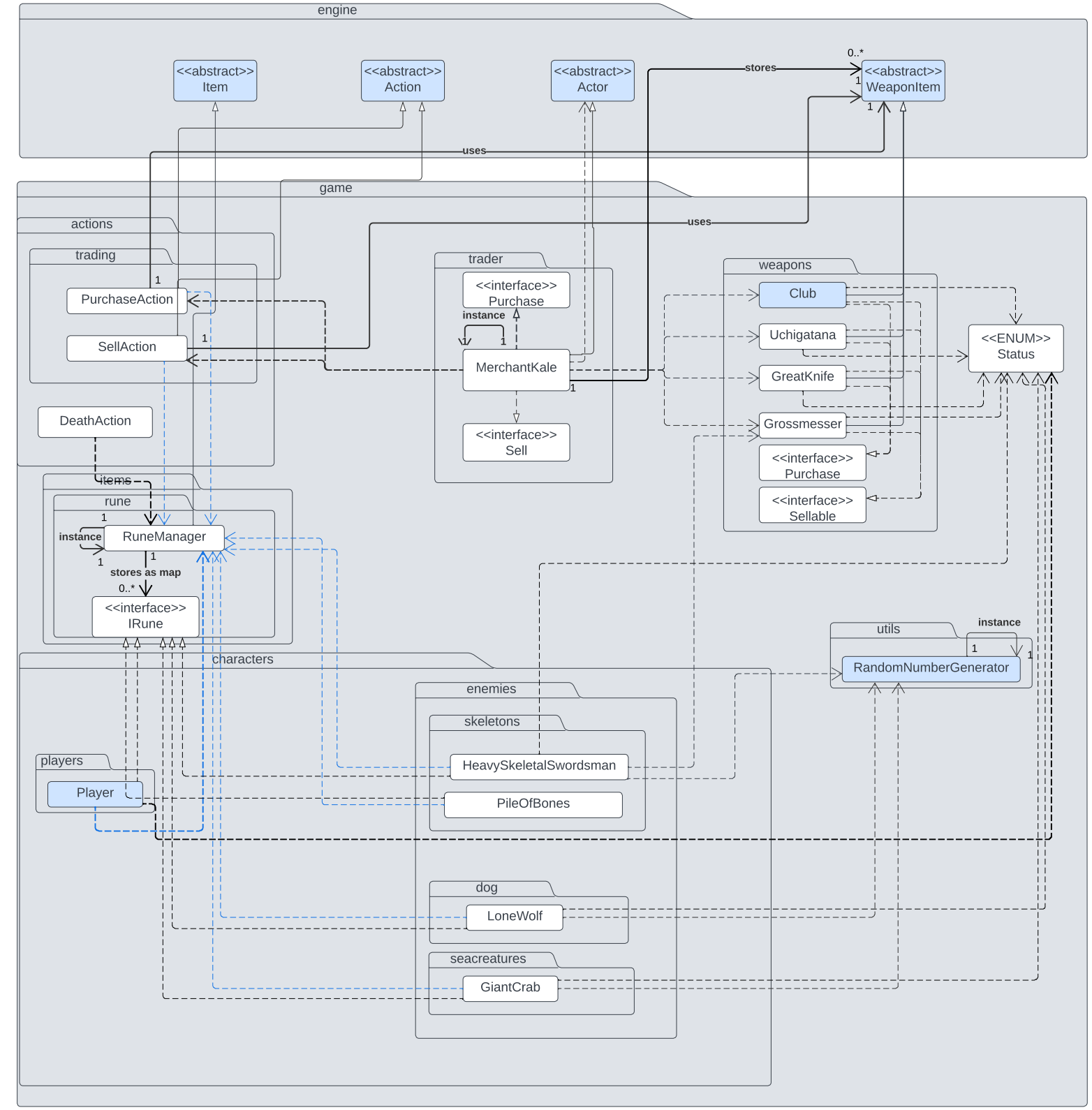
Group's UMLs



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.

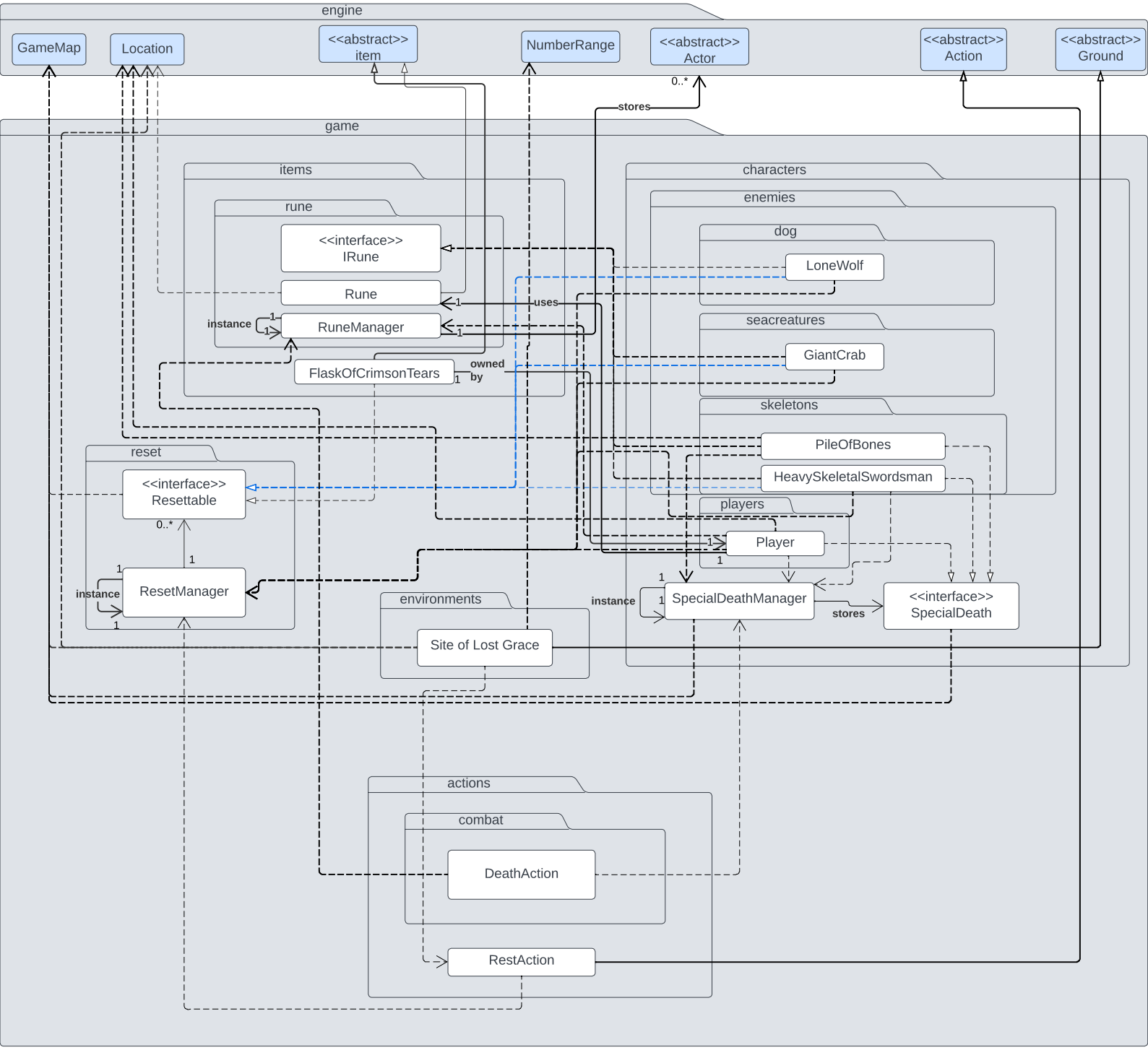
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



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.

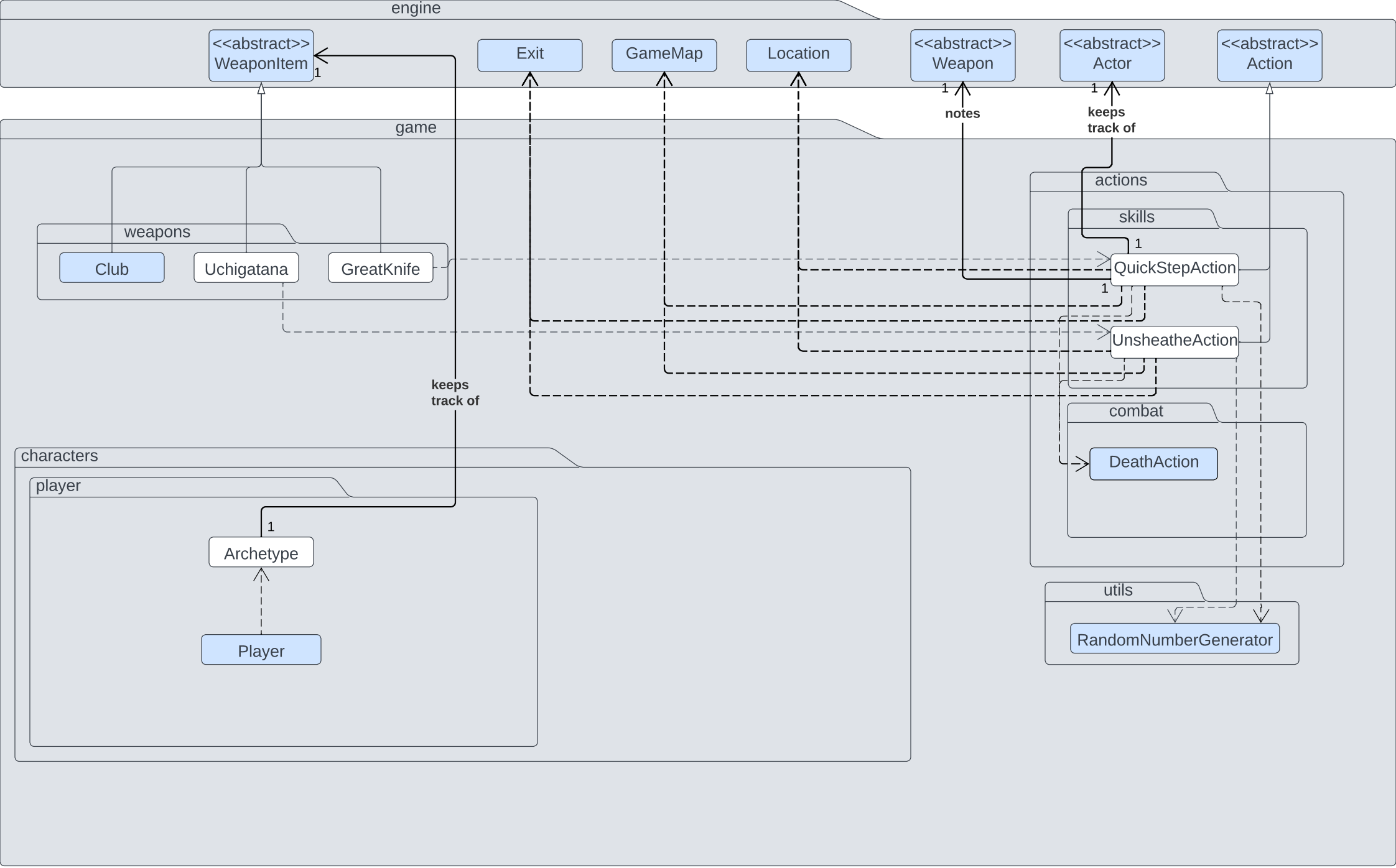
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



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.

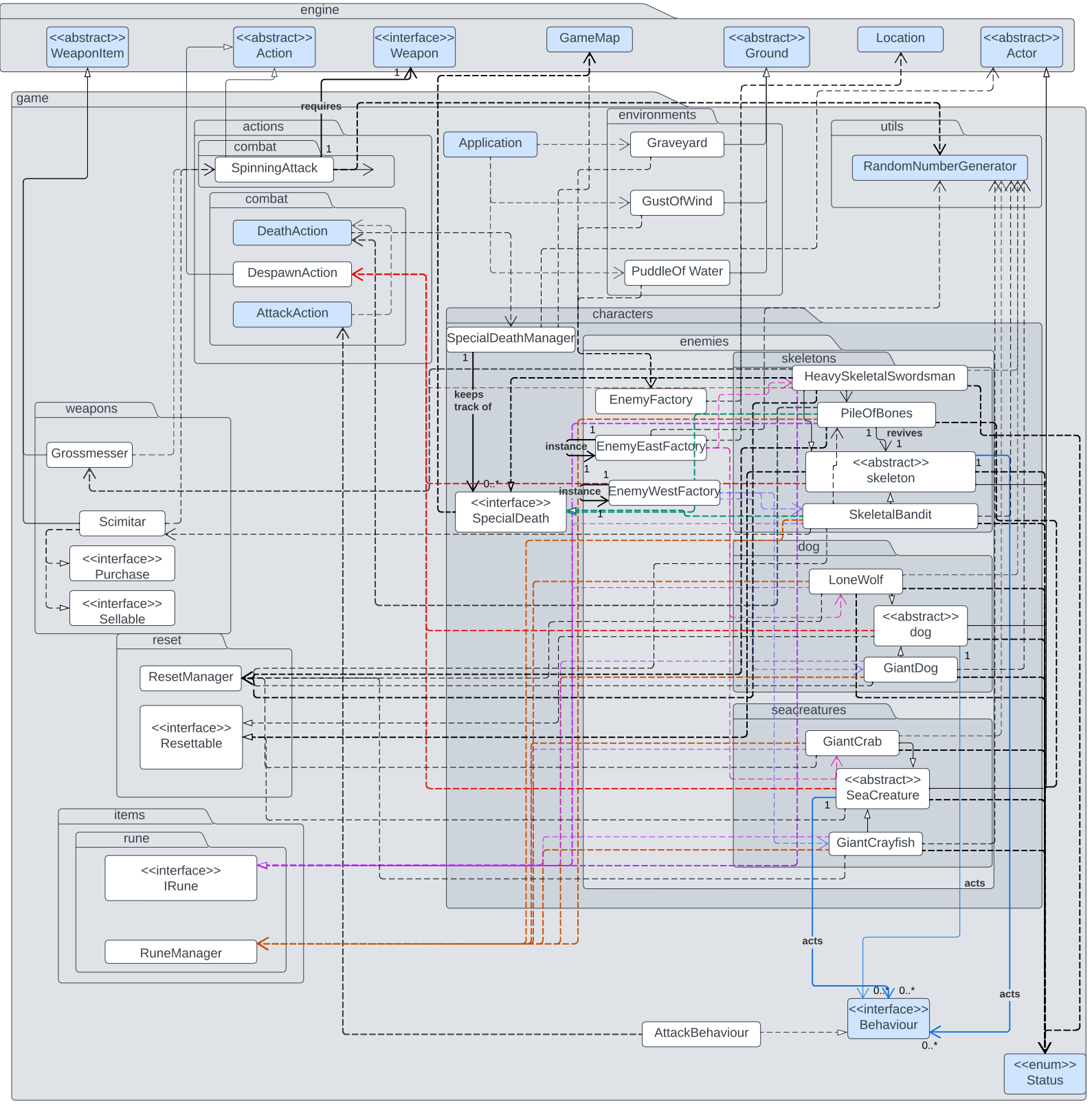
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



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

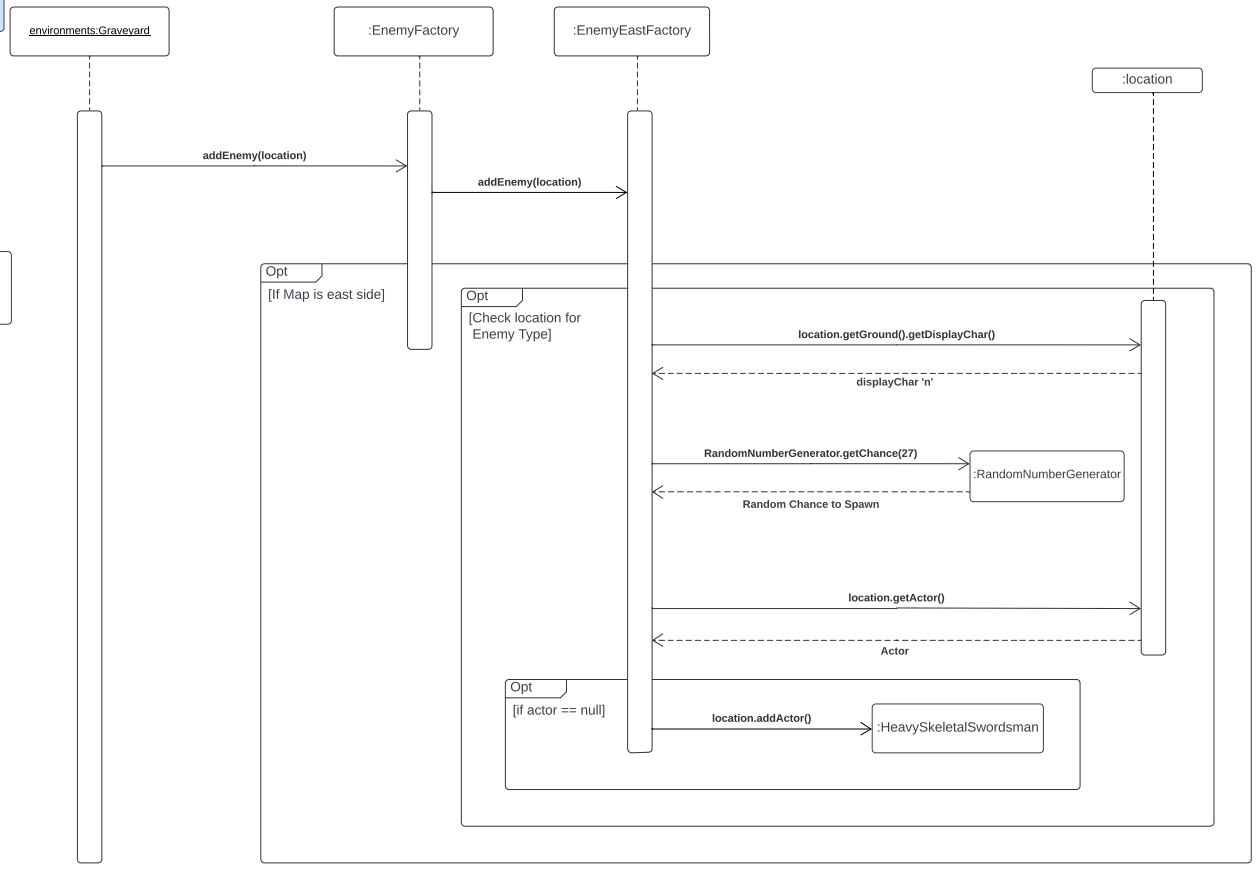
Group's Sequence Diagrams

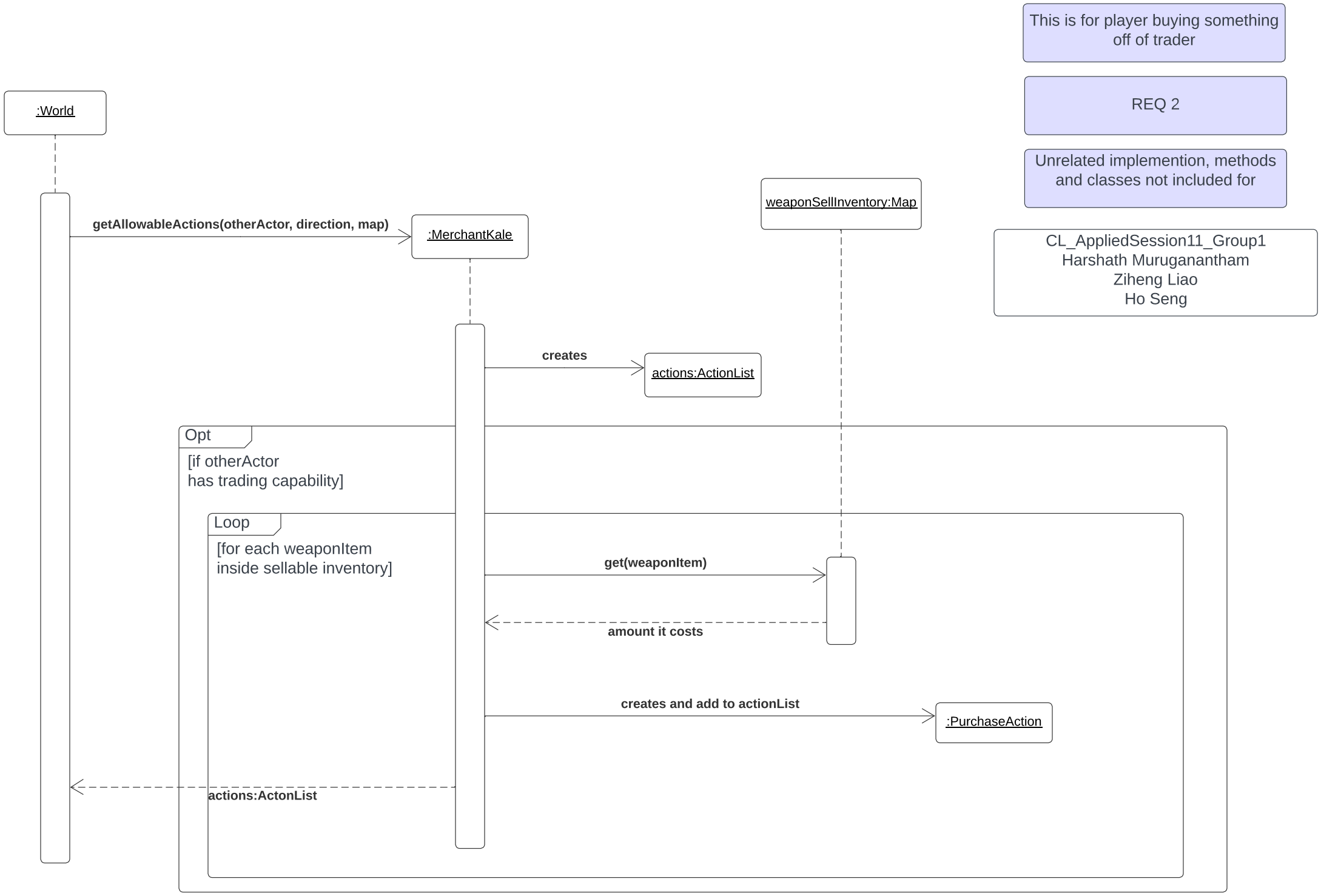
Environments
(Using Graveyard as Example)

Enemies
(Skeletal Swordsman Example)

REQ 1

CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



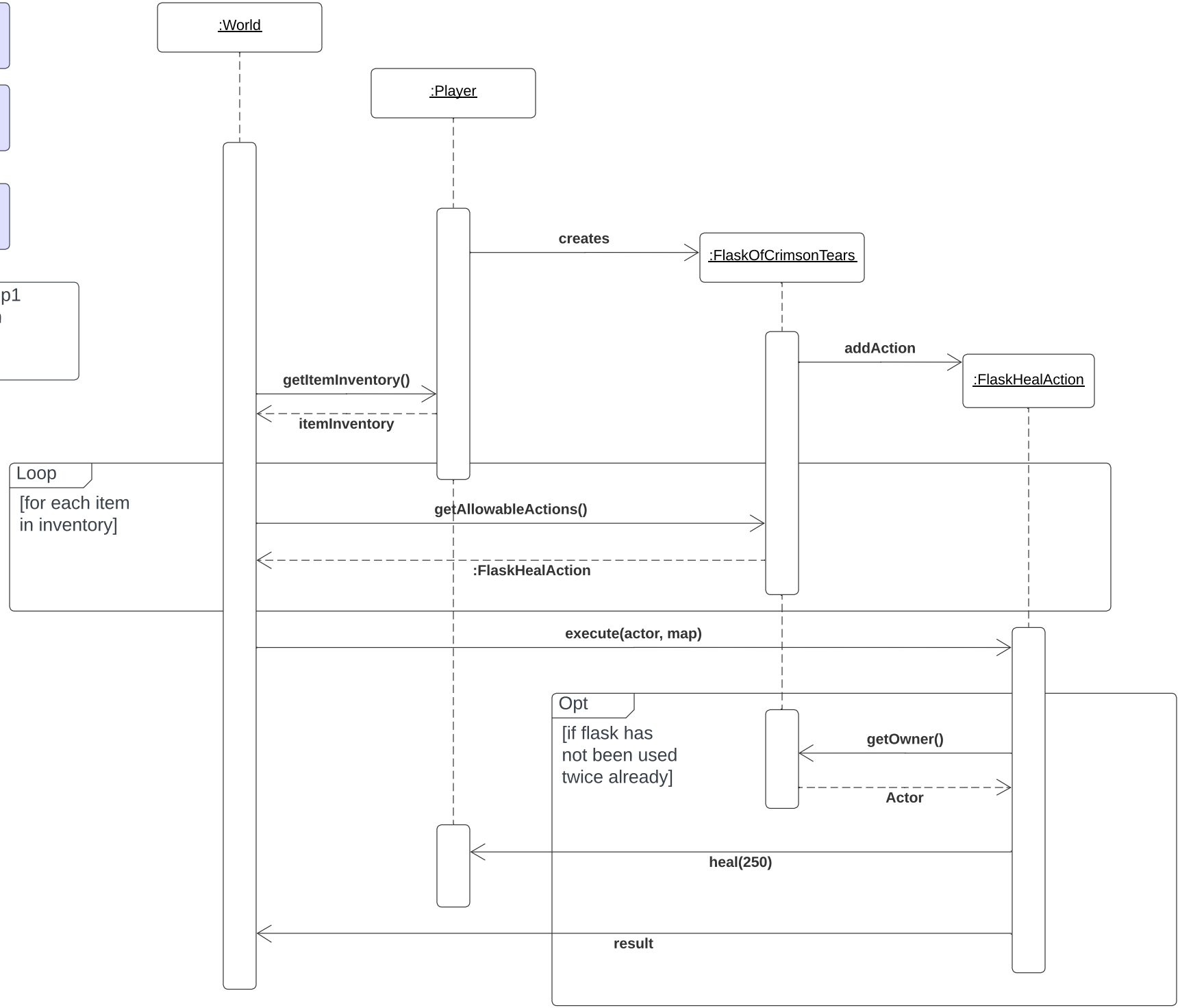

This is for player buying something off of trader
REQ 2
Unrelated implemention, methods and classes not included for
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng
:World
:MerchantKale
weaponSellInventory:Map
getAllowableActions(otherActor, direction, map)
creates
actions:ActionList
Opt
[if otherActor
has trading capability]
Loop
[for each weaponItem
inside sellable inventory]
get(weaponItem)
amount it costs
creates and add to actionList
:PurchaseAction
actions:ActonList

This is for using the flask of crimson tears

Unrelated implemention, methods and classes not included for

REQ 3

CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng

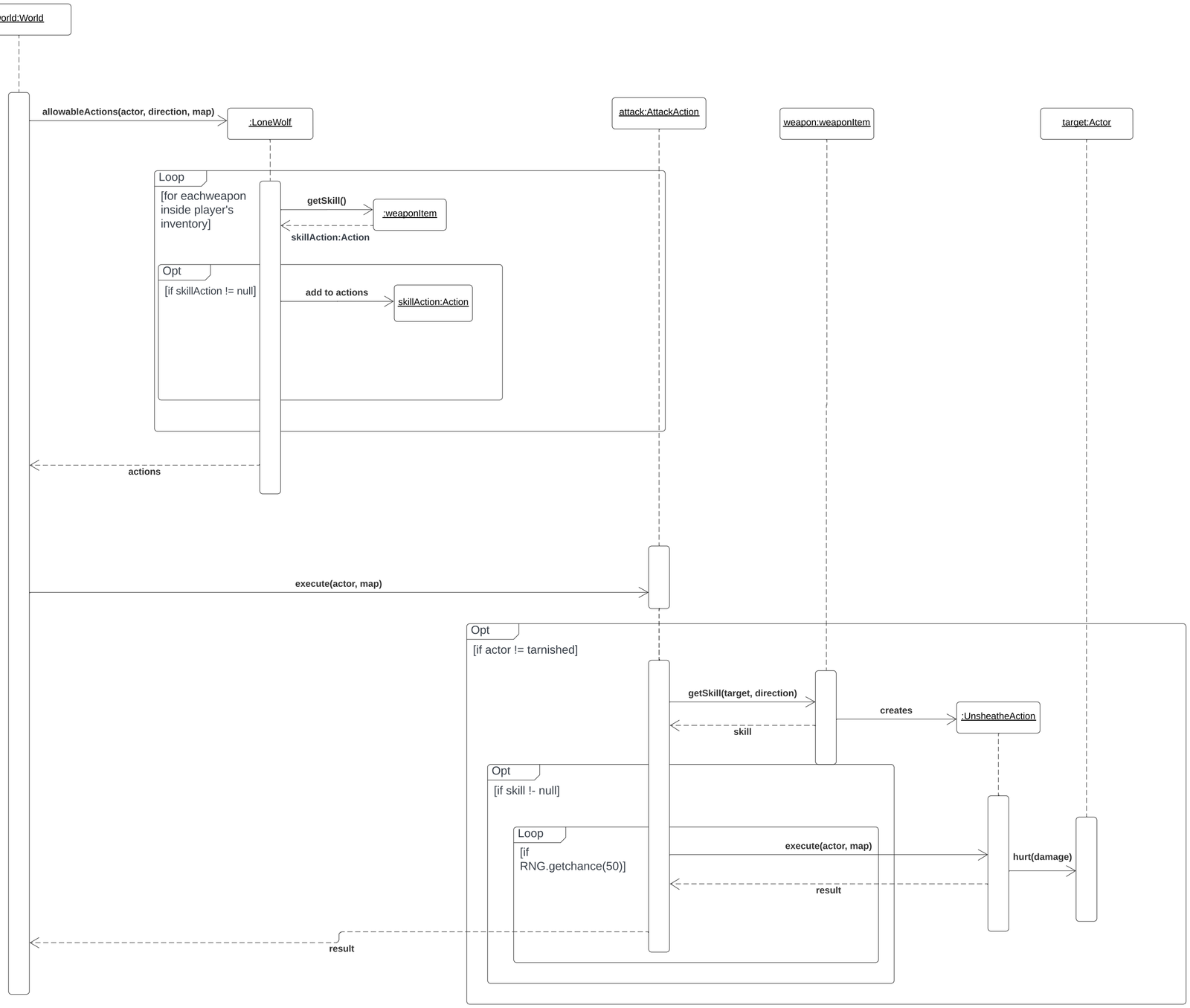


This is for performing a skill under execute

Unrelated implemention, methods and classes not included for

REQ 4

CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng

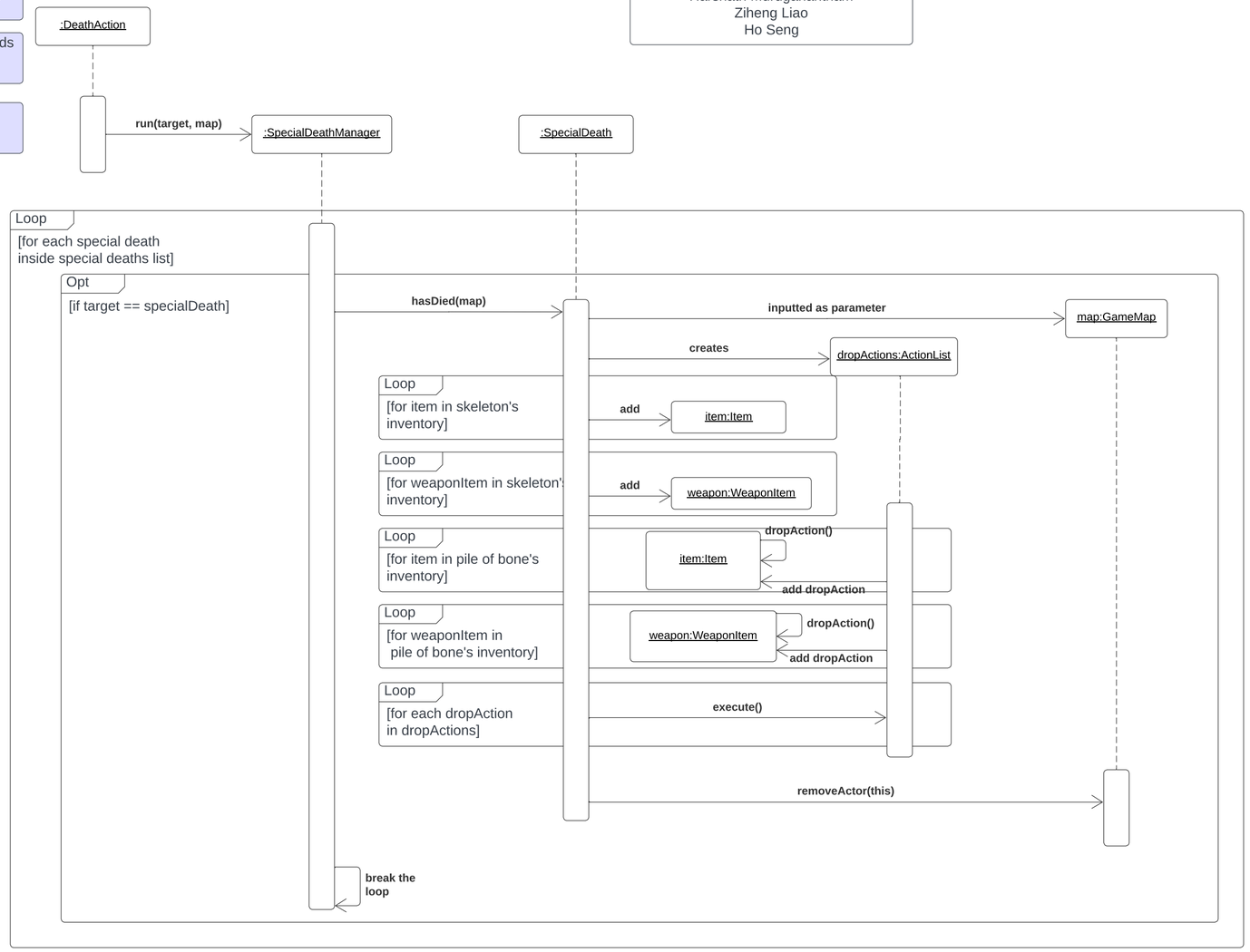


This is for Pile of Bones dying

Unrelated implemention, methods and classes not included for

REQ 5

CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



Lab11Team1 Harshath Muruganantham
Ziheng Liao
Ho Seng

Design Rationale

Req 1

The diagram shows a few new classes being added. Noticeable classes are the new enemy classes, new environment classes and Spinning Attack and Despawn Action, Special Death Manager and Special Death. 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”. A new behaviour was also added called Attack Behaviour, which is implemented inside the playTurn method in all enemy classes. This attack behaviour allow enemies to attack. Inside the playTurn method the enemy has the ability to follow the player if nearby, attack an enemy if nearby and despawn if the chance of 10% is met. Whilst the enemy is deciding to attack, there is a constraint where the enemy cannot attack another character of the same enemy class. To implement this, we have added a capability to the enemy representing its class (ie, Heavy Skeletal Swordsman has the capability cannot attack Skeletons, LoneWolf has the capabilities cannot attack Dogs and GiantCrab has the capability cannot attack SeaCreatures). This was done over checking if the character’s displayCharacter is the same as the enemy’s displayCharacter as we wanted the code to be easily extendable when adding other classes with similar functionality where they cannot attack this class as well (foreshadowing requirement 5). Our method was chosen because all enemy classes already extend from actor class and the actor class has the capability function built in allowing us to reuse code and follow DRY principle. Some of the code in playTurn was included inside playTurn instead of inside behaviour because we felt some of the code such as getting the location of the actor and getting some of the exit points were closely related to the actor’s turn and not the behaviour. Also, a lot of the check constraints that have been done inside playTurn, if abstracted into a behaviour class would be violating many OOPs such as encapsulation.

With enemies that have a special intrinsic skill such as “Slam All”, it was done inside the enemy’s playTurn method because SlamAll uses an intrinsicWeapon rather an acutal Weapon like Grossmesser and therefore if we had made Slam a weapon that uses a special Action

called Slam All, we would be repeating code as the Actor's intrinsic weapon would be the same as the actor's actual weapon. In addition, Slam All is an action that fundamentally differs from intrinsic Weapon and does not have the addSkill functionality provided inside the engine, therefore it was decided to place it inside playTurn method so that each time an intrinsic weapon is called, there's a 50% chance that the special skill "Slam All" will be activated. Whereas some enemies have special skills such as "SpinningAttack" which rely on their weapon and this adding the skill to the weapon is possible in this case. We also considered the possibility of extending from IntrinsicWeapon so that each enemy has its own intrinsicWeapon, but this was not possible due to the IntrinsicWeapon class being a final class. Thus it was not possible to add skill to add extra functionality to IntrinsicWeapon.

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 using playTurn/allowableActions to mimic the tick method, 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. We have created a new interface called "SpecialDeath" that handles any death action that is not similar to the normal death action. We have also created a new SpecialDeathManager which contains a list of all the SpecialDeath implemented classes. When HeavySkeletalSwordsman dies, the DeathAction calls SpecialDeathManager instead which then calls the SpecialDeath method inside HeavySkeletalSwordsman. Inside this method is where all the specified requirements for HeavySkeletalSwordsman dying is placed. A similar mechanism occurs with PileOfBones. This method of adding a SpecialDeath interface was chosen over implementing these methods inside DeathAction to avoid typecasting and downcasting from the Actor abstract class to the specific character class that has died. Also, most of these SpecialDeath methods call upon attributes and methods stored within the respective character class therefore it made more sense in terms of encapsulation to let the character deal with its own death.

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). Like mentioned previously, SpinningAttack is created as its own class but called upon in a method called getSkill inside

Grossmesser. This was done because WeaponItem implements the Weapon interface and inside the interface is a method called getSkill which was designed for this very function of getting the skill from a weapon. So rather than creating a new method for returning a SpinningAttackAction, it is more efficient to use already existing code as it was designed for extensibility. When HeavySkeletalSwordsman attacks another enemy, it has a 50% chance of using SpinningAttackAction. This was implemented inside AttackAction rather than inside the Grossmesser class as we are following the SRP where AttackAction should be responsible for determining whether an Attack goes through or not and it isn't the weapon's responsibility to determine the outcome of an attack.

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 a few 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 SpecialDeath method in Player, so choosing the first method would dictate that only code in SpecialDeath method in Player 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.
When Actors other than the Player die, they will not drop Runes but rather transfer the Runes directly to the Player. Therefore it was decided that this implementation is easier to have inside a RuneManager to handle this. More details below.

To manage the currency of the game a new class called RuneManager was created. RuneManager is responsible for identifying how many Runes each Actor is carrying or worth. RuneManager has a static attribute of itself as there should only be one RuneManager inside the game. This is an improvement to our initial idea where each Actor will have a Rune attribute due to the fact that the RuneManager method is much more extensible if there were more Actors to be added into the future (OCP). Whereas it is unreasonable to have each new Actor type to have a Rune attribute and becomes unsustainable in a larger scale.

Initially the reason for "Merchant Kale" to extend from the abstract class "Actor" due to the attributes of itemInventory and weaponInventory inside the Actor class, but the main reason now is to extend the methods playTurn and allowableActions so that the Player is able to interact with the trader. Inside "Merchant Kale", new maps were created to store purchasable weapons and sellable weapons were created called weaponSellInventory and weaponPurchaseInventory. The intended use for this is to display the weapons that are being sold by the trader. Any new weapons that will be sold will need to be added into the inventory.

Lab11Team1 Harshath Muruganantham
Ziheng Liao
Ho Seng

Previously “Merchant Kale” used to interface “Trader” which contained the methods buying and selling. Now “Merchant Kale” extends from “Purchase” and “Sell” due to LSP and SRP where we didn’t want to risk the possibility that in the future a trader could only do one of the following functionalities. New traders created in the future will now have the option to implement one or both interfaces.

Two new classes to handle the actions of buying and selling were created called “PurchaseAction” and “SellAction” and both of these classes extend from Action as it is needed to implement the actions being ran which the abstract Action class does for us. Inside the PurchaseAction is an instance of RuneManager to handle the transactions for us without needing to input it as a parameter(another reason why it is good to have RuneManager have a static attribute). The possibility of implementing these functions as methods inside MerchantKale was quickly scrapped since these are Actions and therefore must extend from the Actions class. Furthermore, in order to follow the SRP, these actions should not be the responsibility of MerchantKale and in the future by having these actions as their own class, we are allowing ease of extensibility as future traders will just need to call such classes without the need to repeat the same code inside a separate trader (DRY). A possibility of combining PurchaseAction and SellAction into one class “TradeAction” was also omitted as to follow SRP and not create a god-class. Our chosen method allows us to segregate functioanlities to different classes and allows for more clear and extensible code.

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. When the user decides to use the "Flask of Crimson Tears" a new class called "FlaskHealAction" is called which allows the user to use the Flask and Heal if they have any usages left. A decision was made to make FlaskHealAction extend from Action as this allowed us to display the FlaskHealAction as an option to the user in the UI whilst running the game without adding any extra code and using code already present (DRY). This decision was in contrast to implementing the Heal action inside the FlaskOfCrimsonTears as we would have to add code in to display the user options.

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. One of the many problems we faced was relating to finding the last visited site of lost grace to spawn the player at. This was mitigated by adding the location of the last visited Site of Lost Grace as an attribute inside the player class that can be accessed by the reset method within the player class whilst the reset manager is looping through the classes. This approach helps us follow encapsulation laws as the player is only accessing the attribute within the player whilst resetting.

Players implements SpecialDeath interface as it undergoes a different DeathAction as opposed to all other characters in the game. The SpecialDeath interface includes a method called specialDeath which is implemented inside player. When the player dies inside DeathAction, SpecialDeathManager is called if we find out that the person dying is the player. Inside SpecialDeathManager there exists a list of all the SpecialDeath characters, and this list is looped until we find the player instance. Once this instance is found, we execute the hasDied method inside Player. In the hasDied method inside Player we find out how many Runes the Player has and we add it to the last visited location of the Player (which is stored as an attribute inside the player). 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. This location is also added an attribute inside the Player class. If the Player happens to die again, this location attribute is accessed and the Rune at this location is removed, otherwise the Player can pick it up normally. This method is better as opposed to the method stated in our previous design rationale as we are able to avoid downcasting in this method and we are also able to uphold encapsulation laws by not accessing various protected attributes inside DeathAction when we can access it inside its own class. However, this method also includes a lot of additional code to be added but we determined our approach is better for future extensibility as if there are other characters who have a similar death to Player, they can be added without extreme changes to current code (OCP). By avoiding downcasting we are able to retain as much of the previous code as possible helping us follow the OOP of DRY. 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.

Initially we had planned the classes, "Wretch", "Samurai", and "Bandit" to be extended 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. However upon further discussion we have decided not to add "Wretch", "Samurai", and "Bandit" as their own classes as their only function is to modify the hitpoints of Player and add a weapon to Player. The previous method involved a risk of over-inheritance, therefore we decided to create a class called Archetype which is a class of its own contianing information about the starting hitpoints and the starting weapon of the new characters. This information is then inputted inside the normal Player class and all other details in Player are retained. The advantage this gives us is that we wouldn't have to implement other methods in Players more times inside the new classes, allowing us to follow the DRY principle.

Each of these new player Archetypes 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 overwrite the getSkill() method to create a new skill and return it. This simplifies implementation as we are able to use existing methods to get the implementation that we need, extending on the code we are given as it was designed for.

The new abilities discussed in the previous paragraph are "Quick Step" and "Unsheathe". "Quick Step" and "Unsheathe" 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 return special abilities (skills) 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, and 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. We considered adding the skills using the method addAction() to the Item class since WeaponItem extends from Item, but an issue with that is the skill will be ran every turn which is not the output desired. Therefore, we just made the weapon’s special skill return in the weapon’s own getSkill() method (already implemented in Engine code). Another consideration for the skills was to have an abstract class for all the skills but the idea was dismissed as there wasn’t sufficient similarity for the skills to inherit from an abstract class. Furthermore, having it extend from Action was sufficient in achieving our goal of performing a skill action and it prevents over inheritance as well. Not only that, we concluded that if there were to be a new skill being added into the game, having an abstract skills class to inherit from will not be useful as the need for writing down the functionality will still be there.

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 the decision to create enemy factories, which clauclates the location of the ground and calls EnemyEastFactory or EnemyWestFactory depending on the location of the ground. This method was chosen over creating a new class called “Find Map Side” in utils. A few implementation details on how “FindMapSide” 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. This method was however ultimately abandoned as this method violated SRP, as the environment ground classes would also be responsible for spawning enemies, wehreas in our chosen method, the EnemyFactory is responsible. The second method also violeates a few Encapsulation laws as Environment characters are made aware of the Actor class they are creating, however, in our chosen method, this is not the case.

Lab11Team1

Harshath Muruganantham
Ziheng Liao
Ho Seng

Contribution Log

Task/Contribution(~30 words)	Contribution type	Planning Date	Contributor	Status	Actual Completion Date	Extra notes	
Move everything into packages	Code review	17/04/2023	Harshath	DONE	26/04/2023		
Draft code for REQ 1	Code review	17/04/2023	Harshath	DONE	26/04/2023	Started early but then needed to edit it to account for req 5 and the feedback that was provided. If we were just soley accounting for REQ 1 and not any changes made due to other requirements, then it was finished at 20/04	
Javadoc for REQ 1	Code comment	18/04/2023	EVERYONE	DONE	03/05/2023	Was attempted as early as possible but only finalised so late due to changes and readjustments to code	
Draft code for REQ 2	Code review	18/04/2023	Ziheng	DONE	02/05/2023	Attempted early but again finsihed late due to feedback being provided. Credit to Ho for helping as well	
Draft code for REQ 3	Code review	20/04/2023	Harshath	DONE	03/05/2023	A lot of changes were applied to this requirement due to new updates through ed and the expected outputs	
Draft code for REQ 4	Code review	24/04/2023	EVERYONE	DONE	03/05/2023	This task was completed half by Ho, the other half completed by Ziheng and then Harshath put the finishing touches	
Draft code for REQ 5	Code review	24/04/2023	Ho	DONE	03/05/2023	Requirement started by Ho, but Harshath made adjustments by putting the implementation in its own class	
Work on Code for REQ 1	Code review		Harshath	DONE	03/05/2023	Finalising Class Structure, Swithcing code to adhere to DRY principles, adding on capabilities of existsing methods to new classes to avaoid repetition of code. Making sure the code works in the expected manner and mainly WORKS.	
Work on Code for REQ 2	Code review		Harshath and Ziheng	DONE	03/05/2023	Finalising Class Structure, Swithcing code to adhere to DRY principles, adding on capabilities of existsing methods to new classes to avaoid repetition of code. Making sure the code works in the expected manner and mainly WORKS.	
Work on Code for REQ 3	Code review		EVERYONE	DONE	03/05/2023	Finalising Class Structure, Swithcing code to adhere to DRY principles, adding on capabilities of existsing methods to new classes to avaoid repetition of code. Making sure the code works in the expected manner and mainly WORKS.	
Work on Code for REQ 4	Code review		EVERYONE	DONE	03/05/2023	Finalising Class Structure, Swithcing code to adhere to DRY principles, adding on capabilities of existsing methods to new classes to avaoid repetition of code. Making sure the code works in the expected manner and mainly WORKS.	
Work on Code for REQ 5	Code review		EVERYONE	DONE	03/05/2023	Finalising Class Structure, Swithcing code to adhere to DRY principles, adding on capabilities of existsing methods to new classes to avaoid repetition of code. Making sure the code works in the expected manner and mainly WORKS.	
Finalising all the code	Code review	29/04/2023	Harshath	DONE	03/05/2023	Just comparing the expected output to our code. Checking over all the implmentation and seeing if any downcasting was made as it was made previously	
Testing the code for expected outputs	Code review	29/04/2023	Harshath and Ziheng	DONE	03/05/2023	Just watching the expected output and playing the game to see if there were any errors that occured. Fixing all errors that occured.	
JavaDoc for REQ 1	Code comment	02/05/2023	Harshath	DONE	03/05/2023	Writing documentation for the code	
JavaDoc for REQ 2	Code comment	02/05/2023	Harshath and Ziheng	DONE	03/05/2023	Writing documentation for the code	
JavaDoc for REQ 3	Code comment	02/05/2023	Harshath	DONE	03/05/2023	Writing documentation for the code	
JavaDoc for REQ 4	Code comment	02/05/2023	Harshath and Ziheng	DONE	03/05/2023	Writing documentation for the code	
JavaDoc for REQ 5	Code comment	02/05/2023	Harshath	DONE	03/05/2023	Writing documentation for the code	

Task/Contribution(~30 words)	Contribution type	Planning Date	Contributor	Status	Actual Completion Date	Extra notes	
UML diagram for REQ 1	UML diagram	30/04/2023	Harshath	DONE	02/05/2023	Redid UML diagram as our initial design was not the best way we could implement it	
UML diagram for REQ 2	UML diagram	30/04/2023	Ziheng	DONE	01/05/2023	Took on feedback provided and made the changes that were encouraged by the tutor in applied class	
UML diagram for REQ 3	UML diagram	30/04/2023	Ziheng	DONE	01/05/2023	Redid UML diagram as our initial design was not the best way we could implement it	
UML diagram for REQ 4	UML diagram	30/04/2023	Harshath	DONE	03/05/2023	Redid UML diagram as our initial design was not the best way to implement	
UML diagram for REQ 5	UML diagram	30/04/2023	Ziheng	DONE	03/05/2023	Redid UML diagram as our initial design was not the best way we could impl	
Sequence diagram REQ 1	UML diagram	30/04/2023	Ho	DONE	02/05/2023	Readjusted the sequence diagram to make it consistent with the other sequence diagrams	
Sequence diagram REQ 2	UML diagram	29/04/2023	Ziheng	DONE	03/05/2023	Had to confirm that certain syntax was allowed which was the reason for the delay in completion	
Sequence diagram REQ 3	UML diagram	29/04/2023	Ziheng	DONE	03/05/2023	Like REQ 2, had to confirm syntax and if using a specifix example was allowed rather than a broad example	
Sequence diagram REQ 4	UML diagram	29/04/2023	Ziheng	DONE	03/05/2023	Similar to other sequence diagram, just had to get confirmation	
Sequence diagram REQ 5	UML diagram	29/04/2023	Ziheng	DONE	03/05/2023	Similar to other sequence diagram, just had to get confirmation	
Design rationale 1	Design rationale	01/05/2023	Harshath	DONE	02/05/2023	Due to changes made with the design of the code, we had to justify why we made such changes	
Design rationale 2	Design rationale	01/05/2023	Ziheng	DONE	03/05/2023	Added extra justificaiton on why current implementation is better than previous implementation	
Design rationale 3	Design rationale	01/05/2023	Harshath	DONE	03/05/2023	Like before, changes were made in this requirement as well so we needed to edit the design rationale	
Design rationale 4	Design rationale	01/05/2023	Ziheng	DONE	03/05/2023	Changes made to the skill added to the weapon specifically and thus needed to change our design rationale behind it	
Design rationale 5	Design rationale	01/05/2023	Harshath	DONE	03/05/2023	Updated req 5 as we found a better way to design our code. But overall, not much of the design was changed. Just additional implementation and justification	
Draft code for REQ 1	Code review	17/04	Harshath	DONE	02/05/2023	Started early but then needed to edit it to account for req 5 and the feedback that was provided. If we were just soley accounting for REQ 1, then it was finished at 20/04	