ASSIGNMENT 3

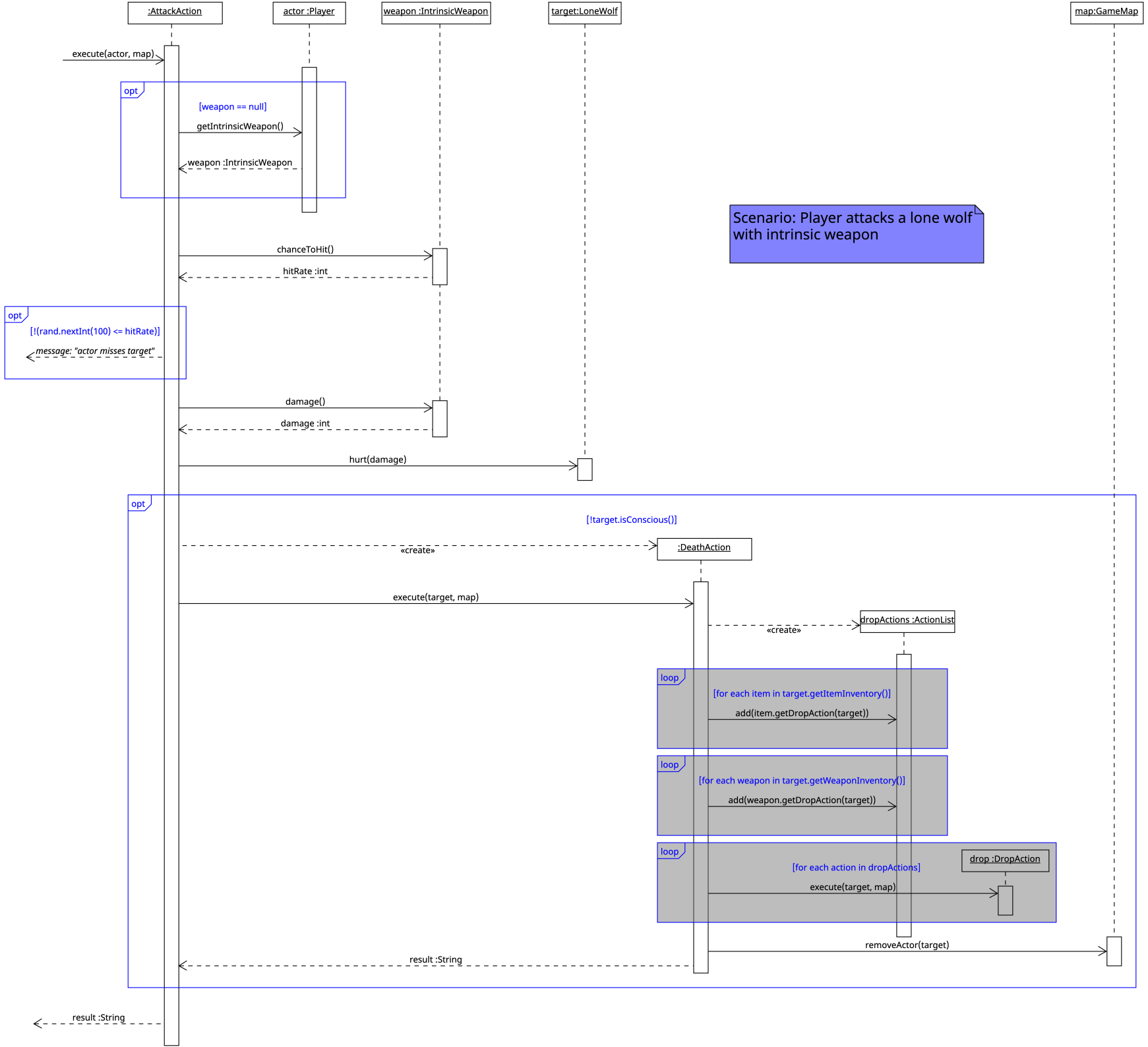
By Lab11Team1

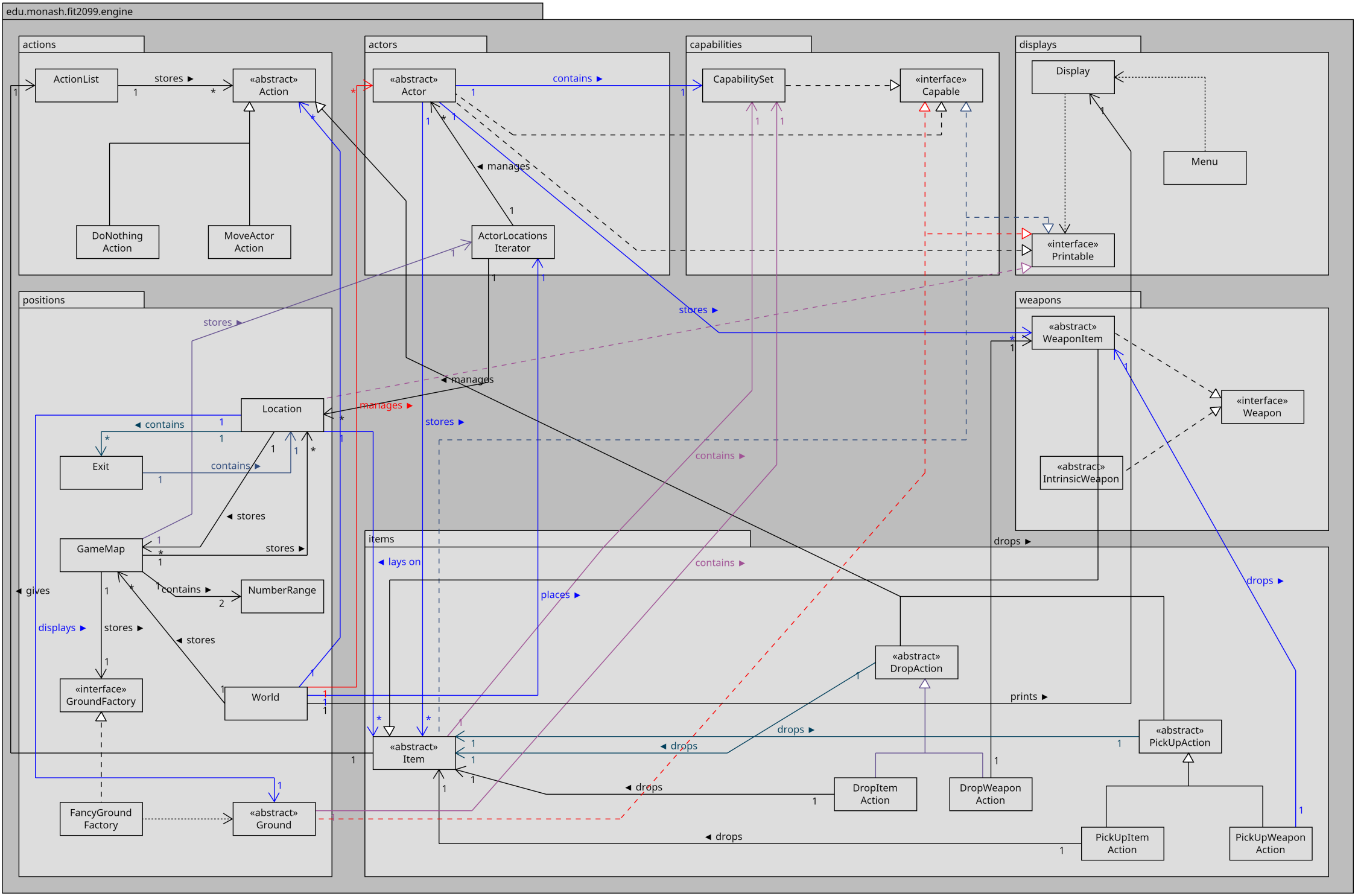
Harshath Murugananthamm, Ziheng Liao, Ho Seng

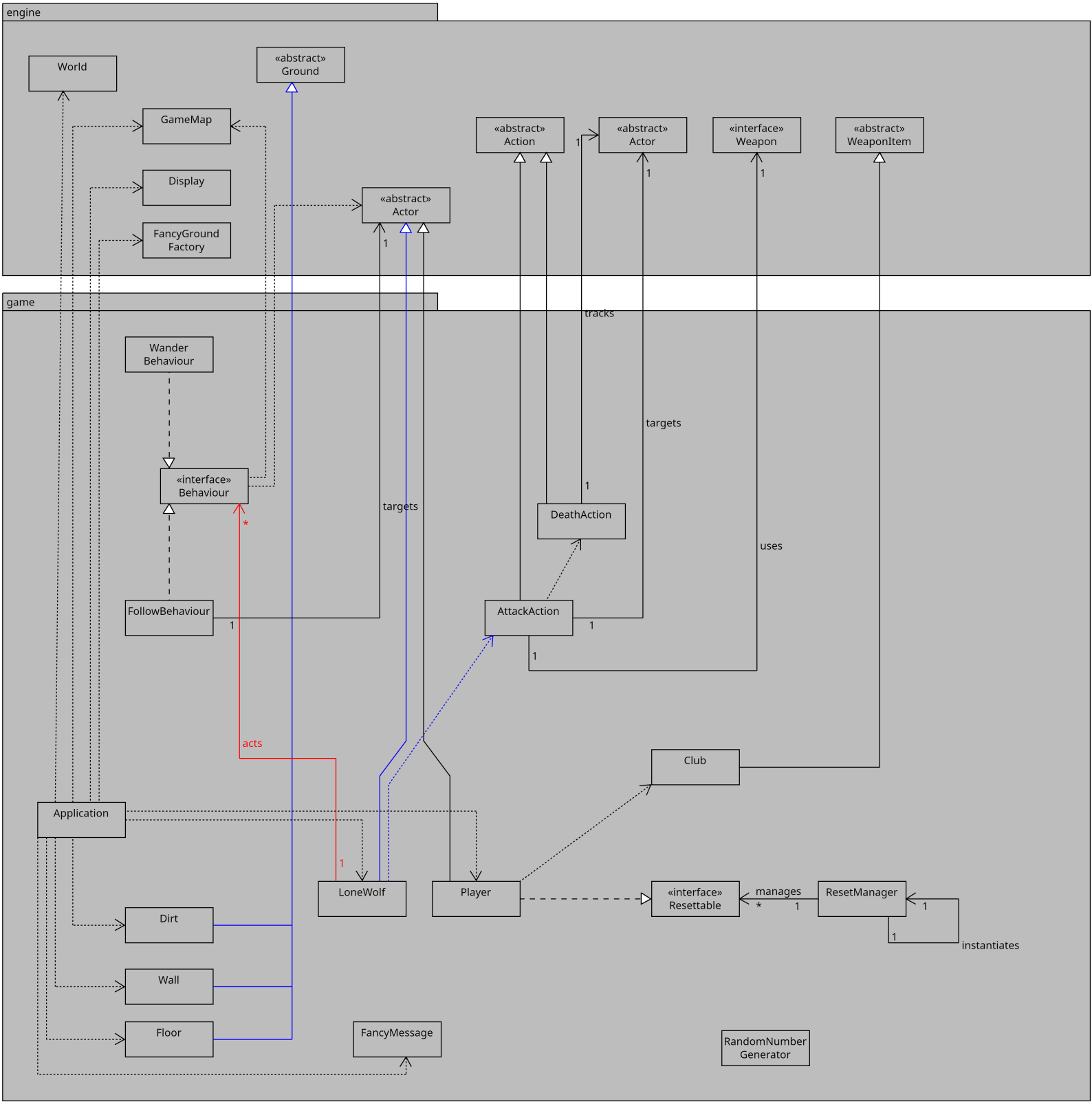
Lab11Team1

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

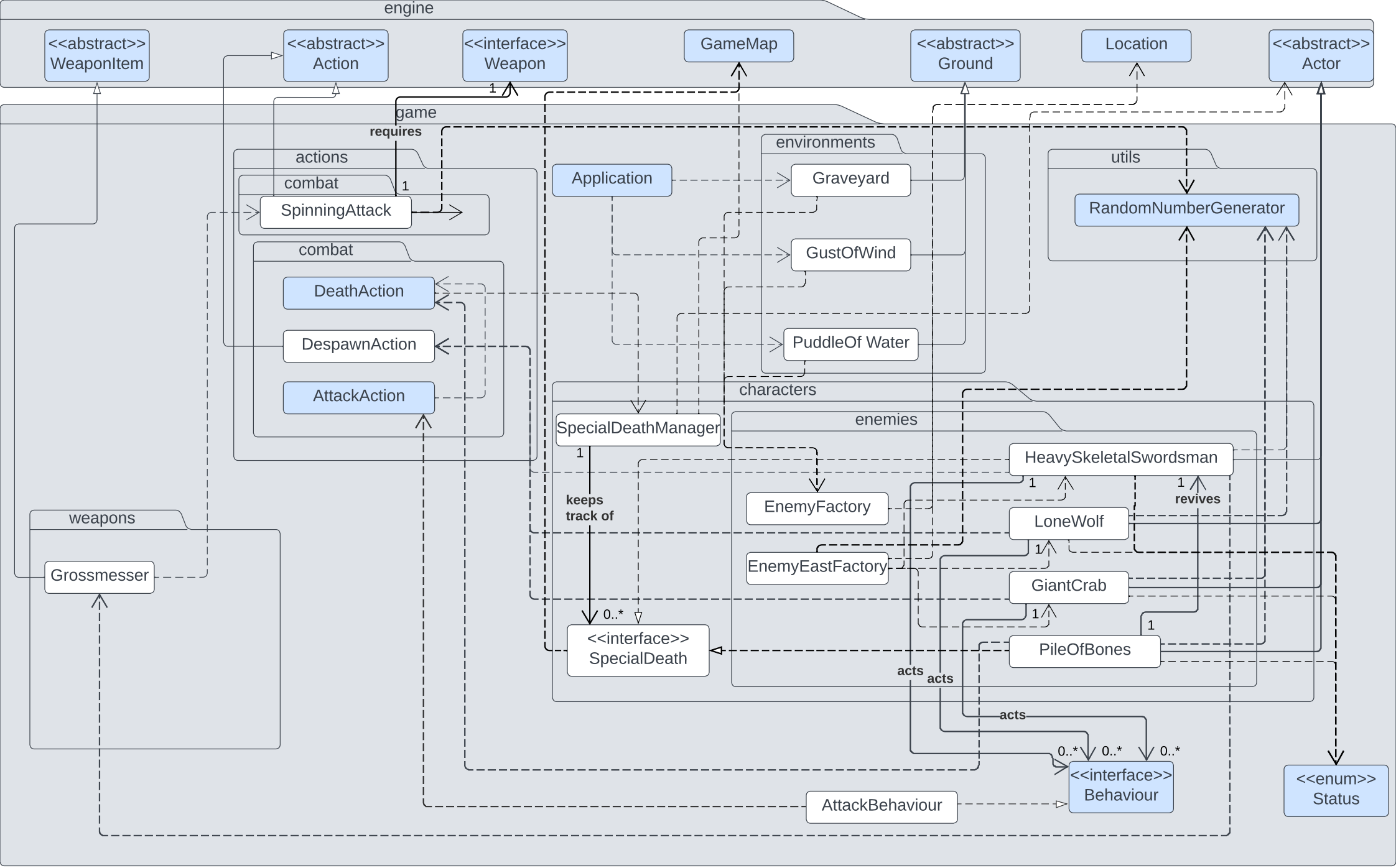

engine
World
«abstract»
Ground
GameMap
Display
FancyGround
Factory
«abstract»
Actor
«abstract»
Action
«abstract»
Actor
«interface»
Weapon
«abstract»
WeaponItem
1
1
1
1
game
tracks
Wander
Behaviour
targets
«interface»
Behaviour
*
targets
1
DeathAction
uses
FollowBehaviour
1
AttackAction
1
1
acts
Club
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 (Assignment 2)

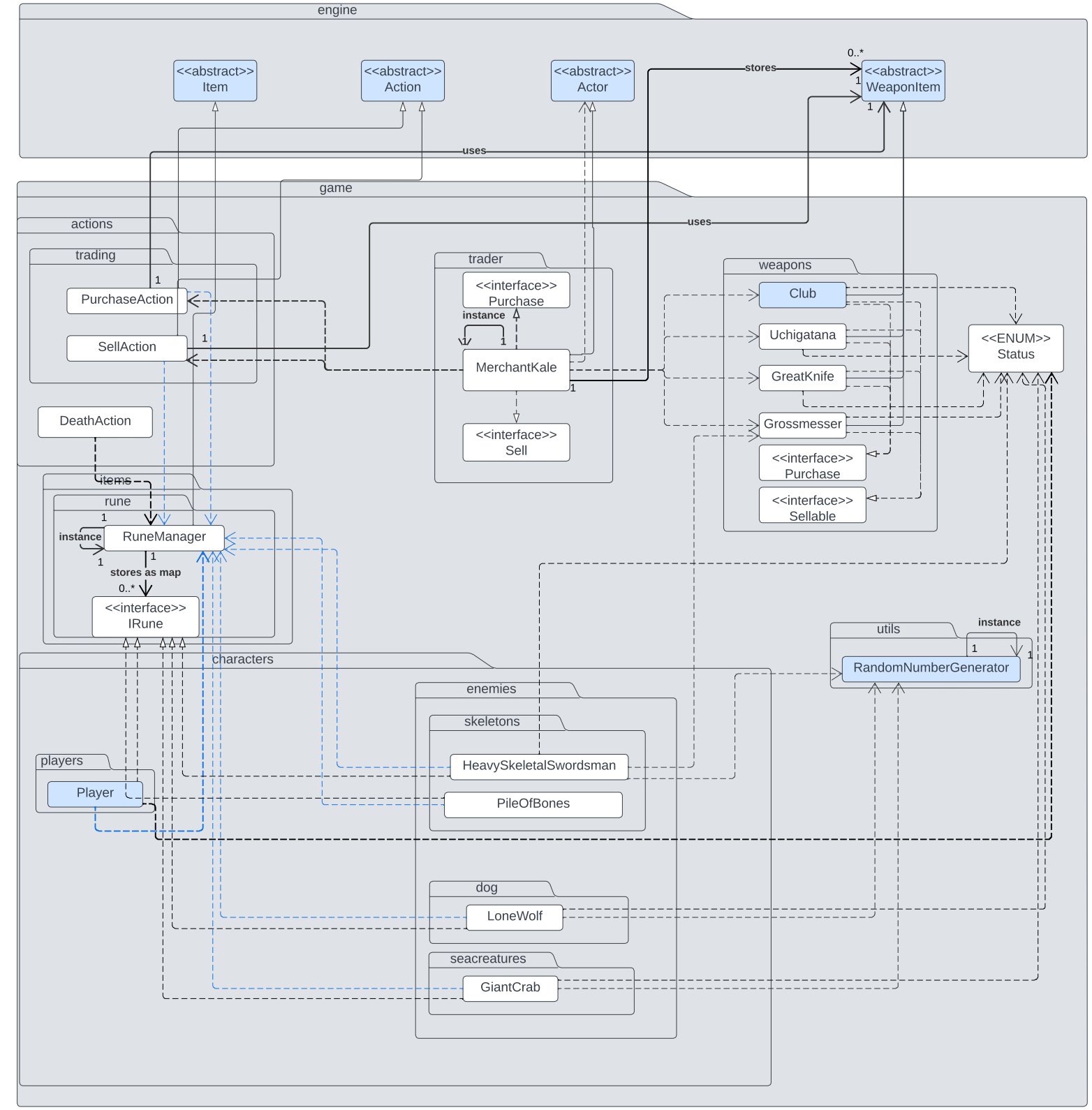
DISCLAIMER: These UMLs have not been changed since our submission to Assignment 2. The new UMLs in Assignment 3 relate back to the classes from Assignment 2 where applicable. View these pages to see how the classes in Assignment 2 (mentioned in Assignment 3) relate to each other.



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.

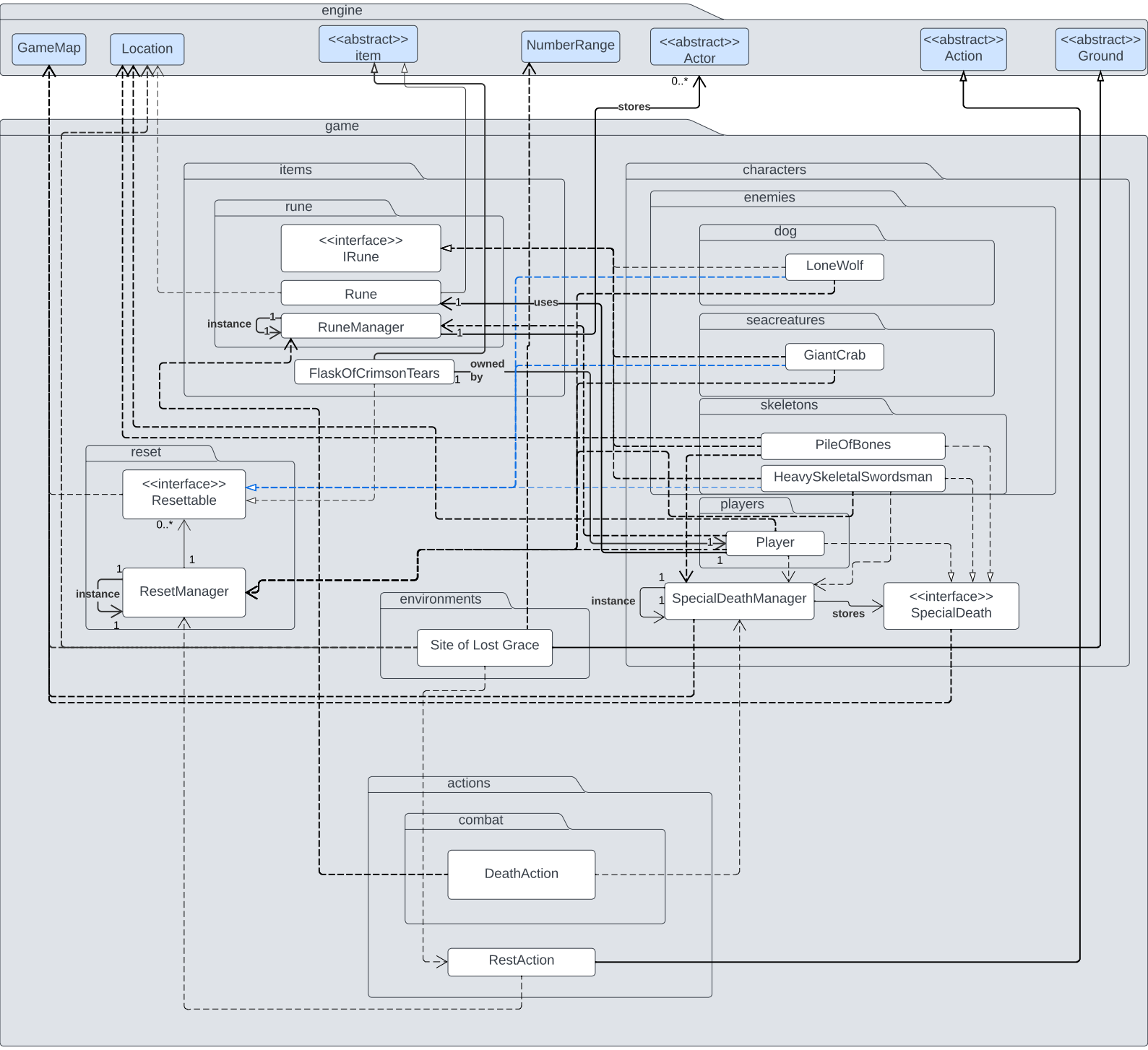
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



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.

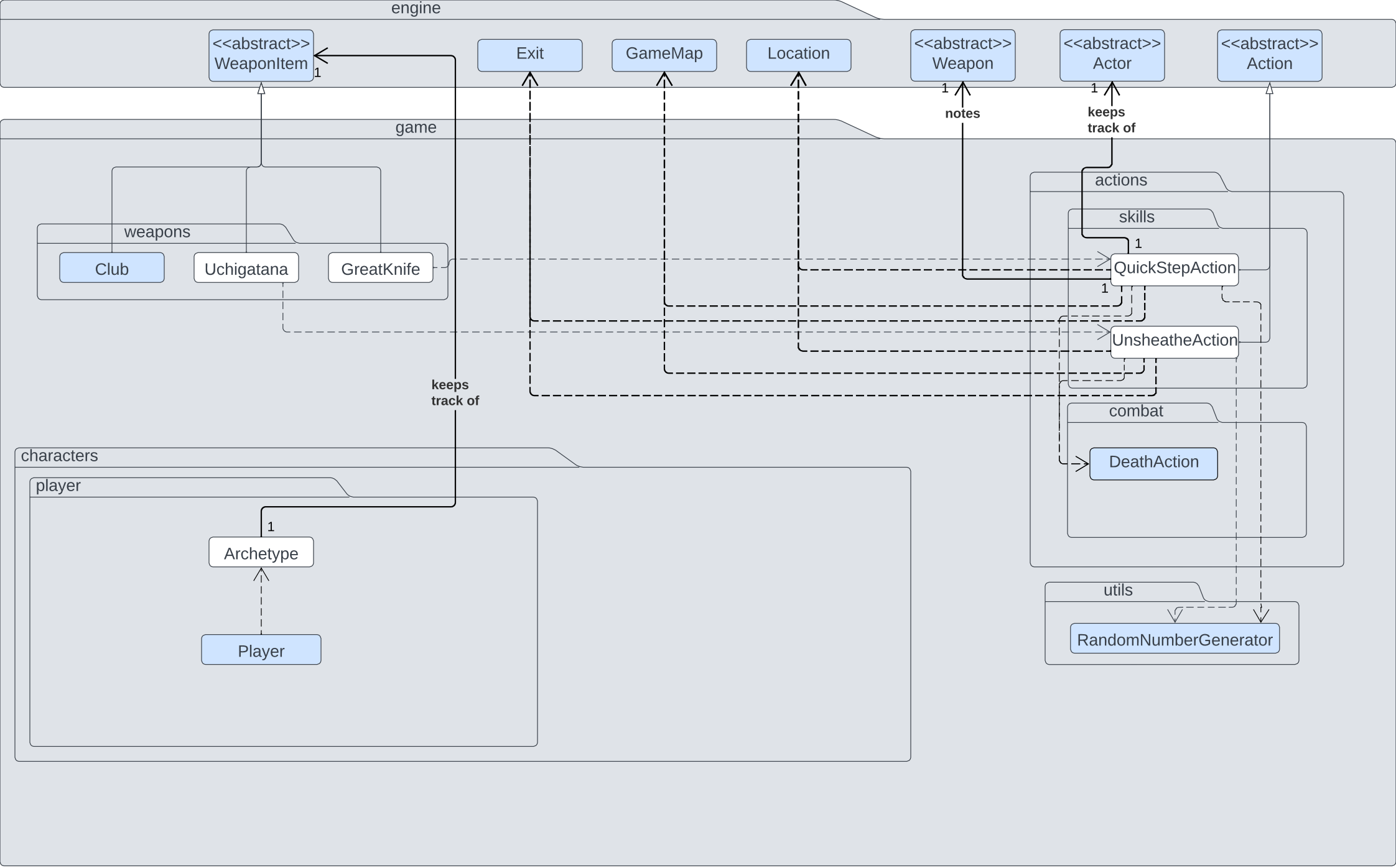
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



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.

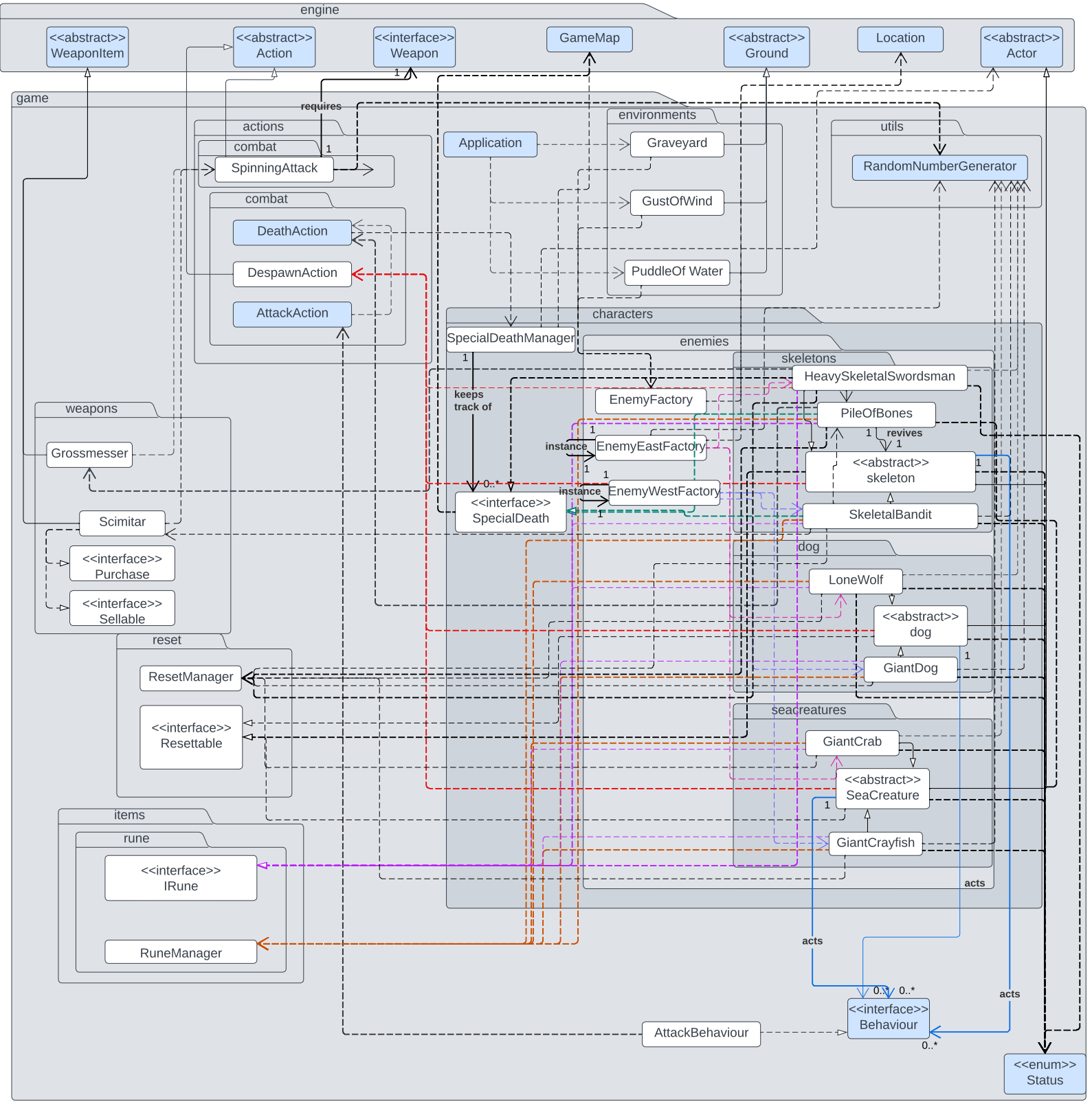
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



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.

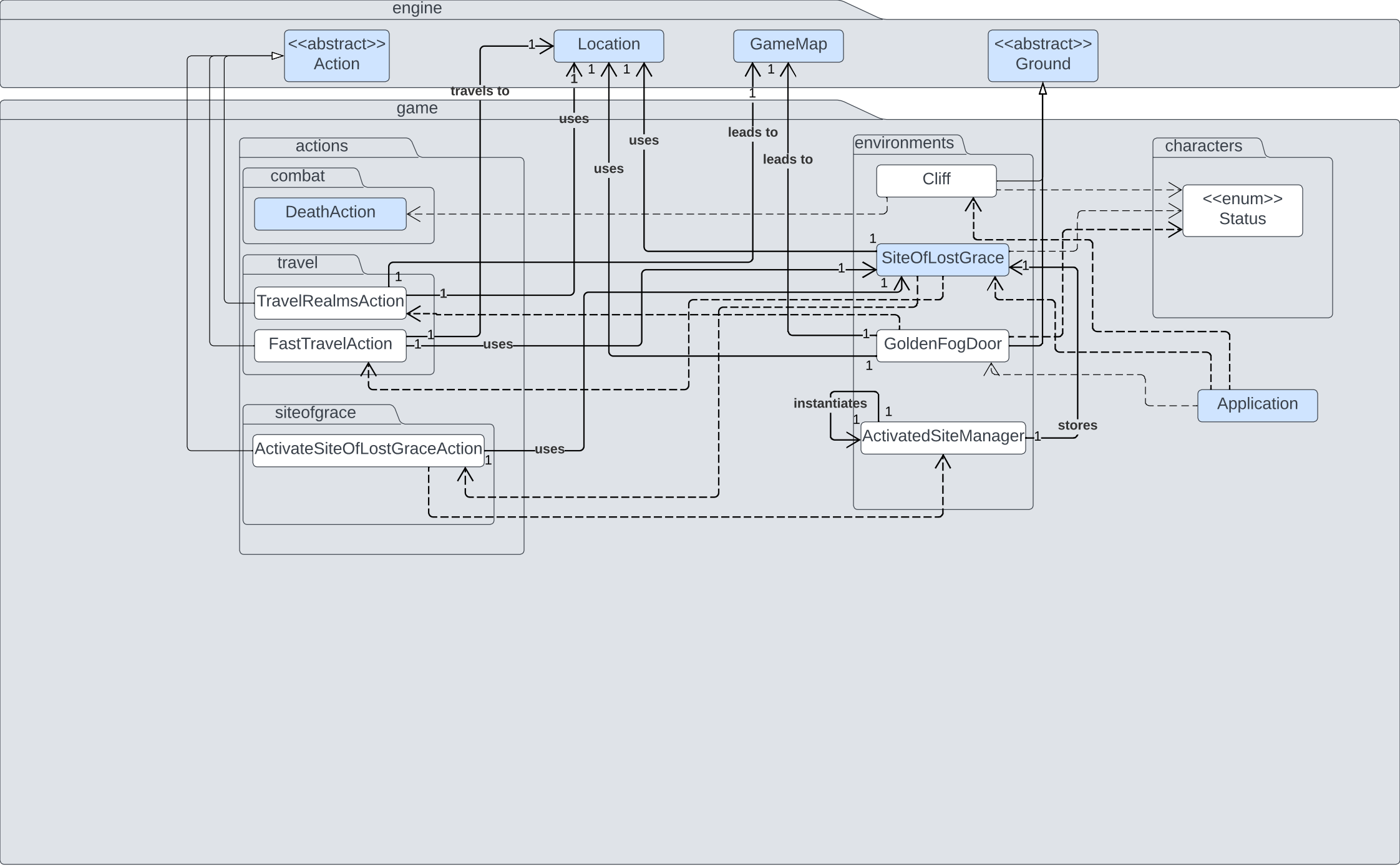
OLD Classes are represented using the colour of this box.

CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng

Lab11Team1

Harshath Muruganantham
Ziheng Liao
Ho Seng

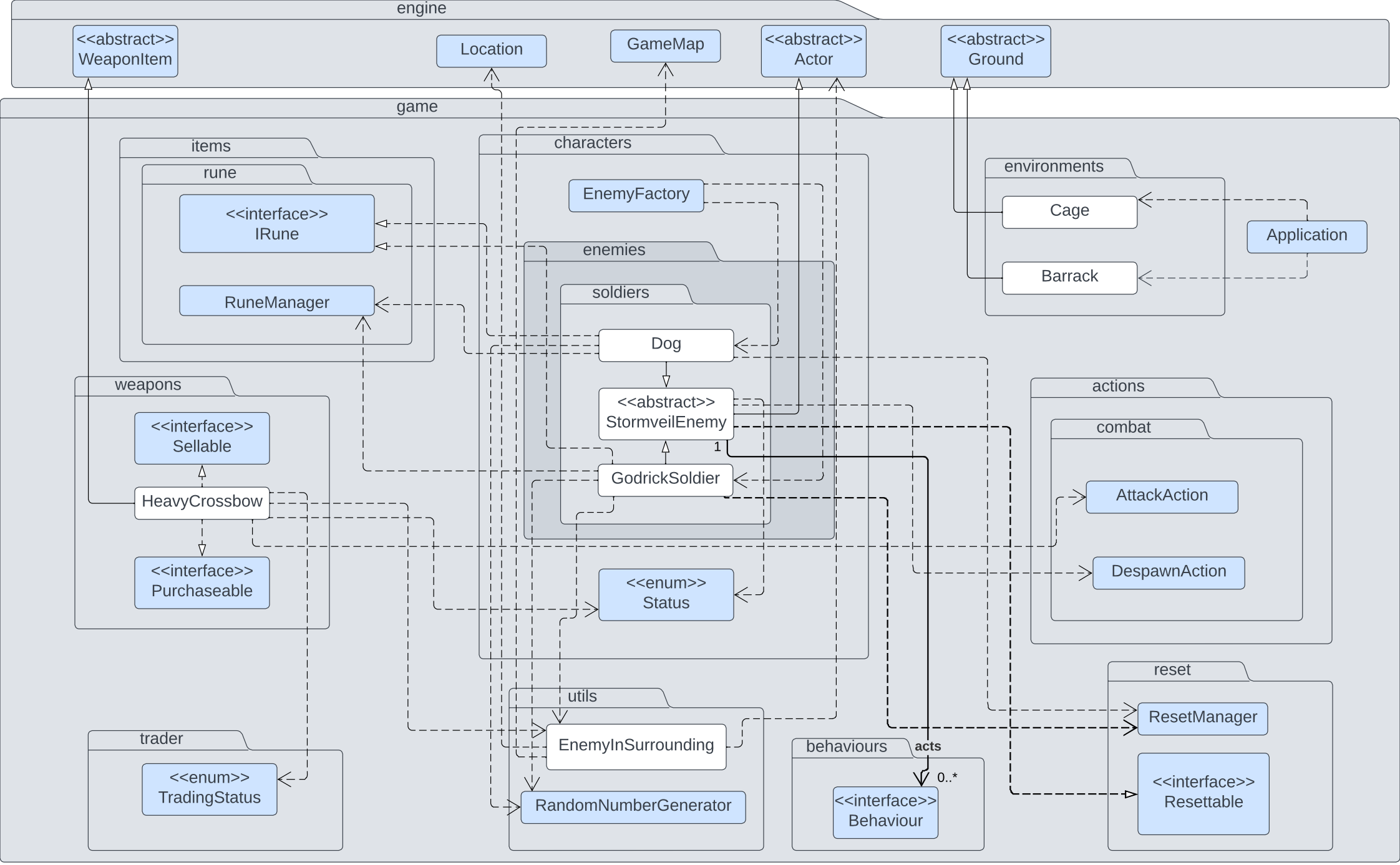
Group's UMLs (Assignment 3)



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.

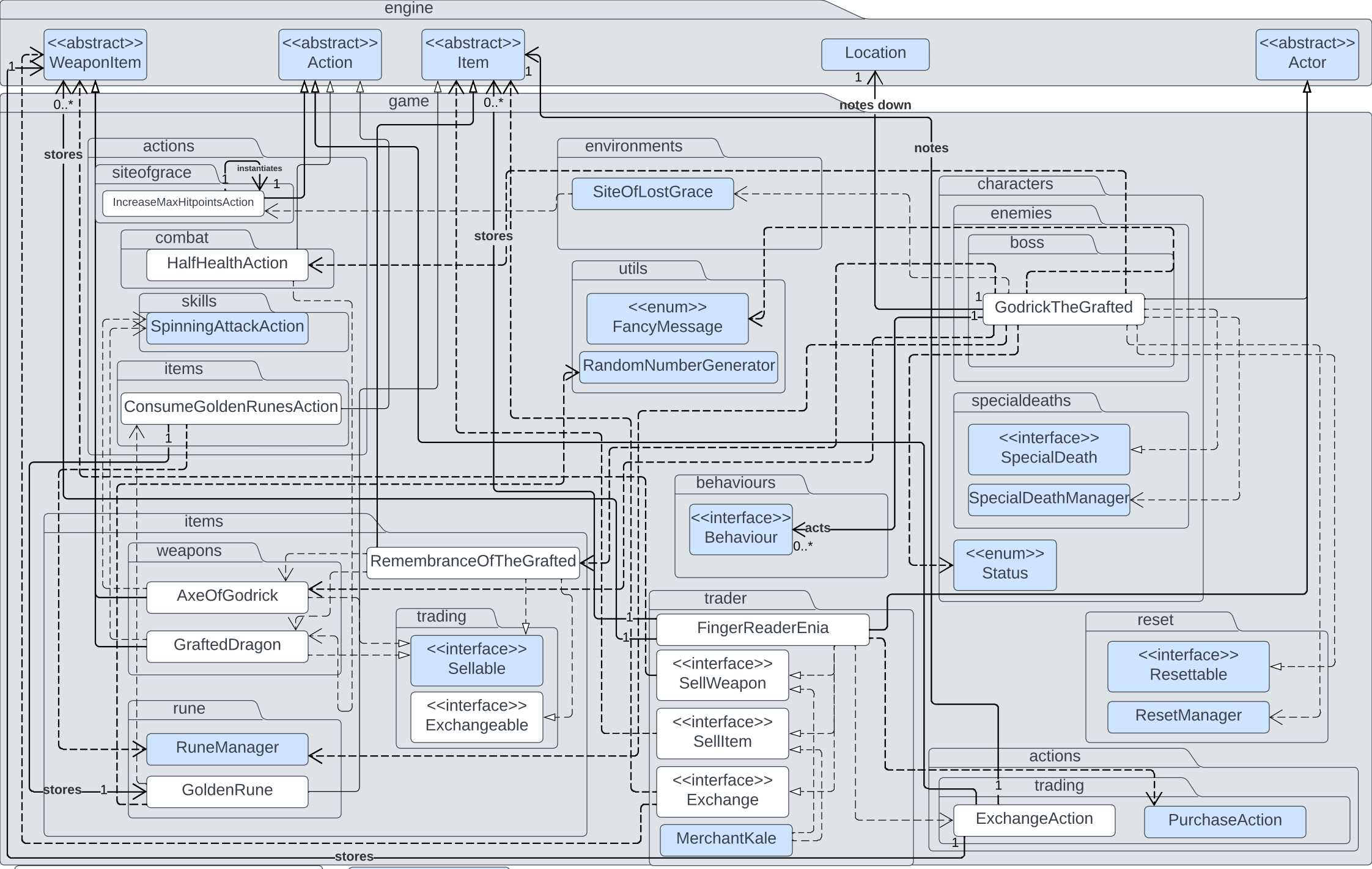
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



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.

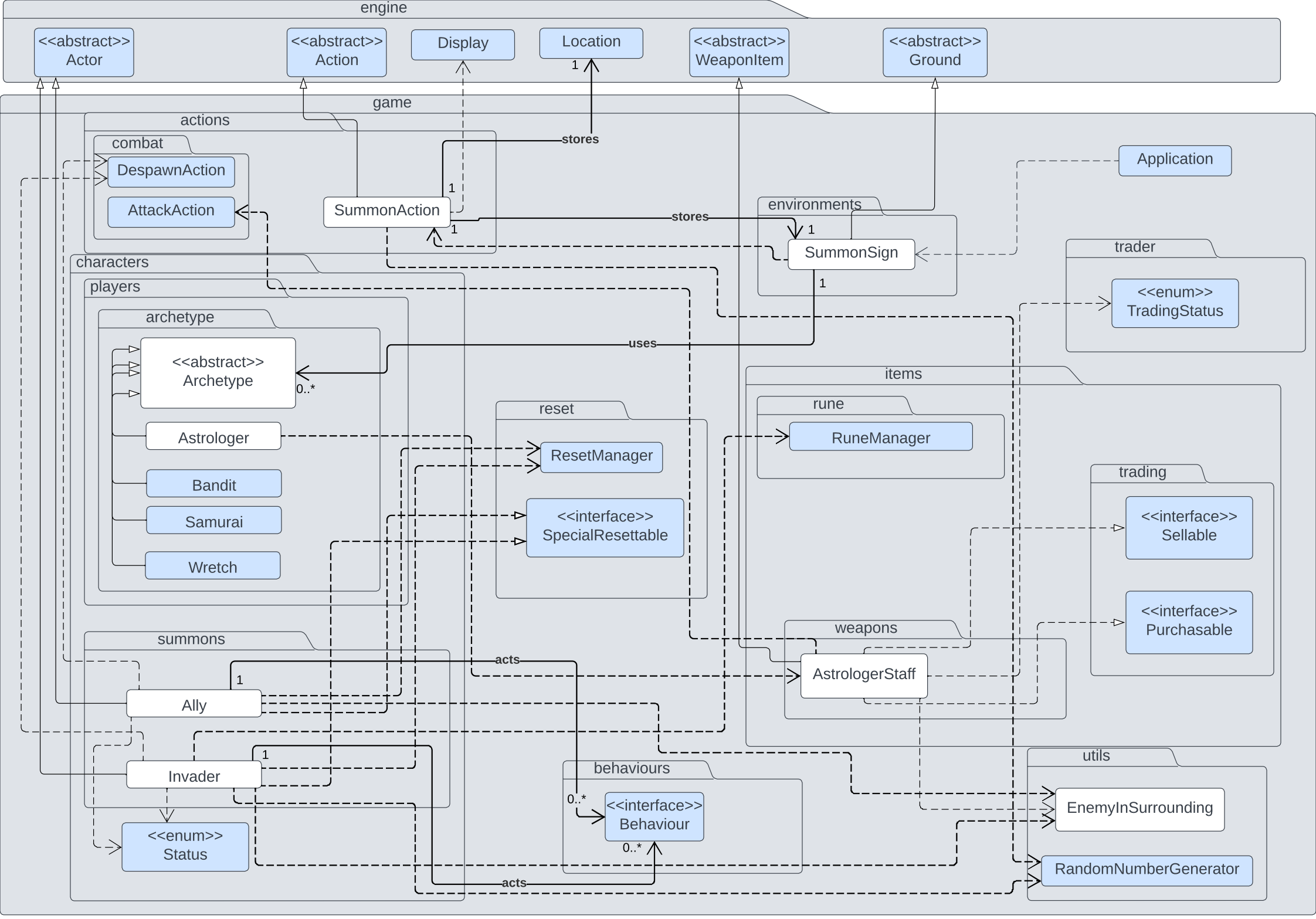
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



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.

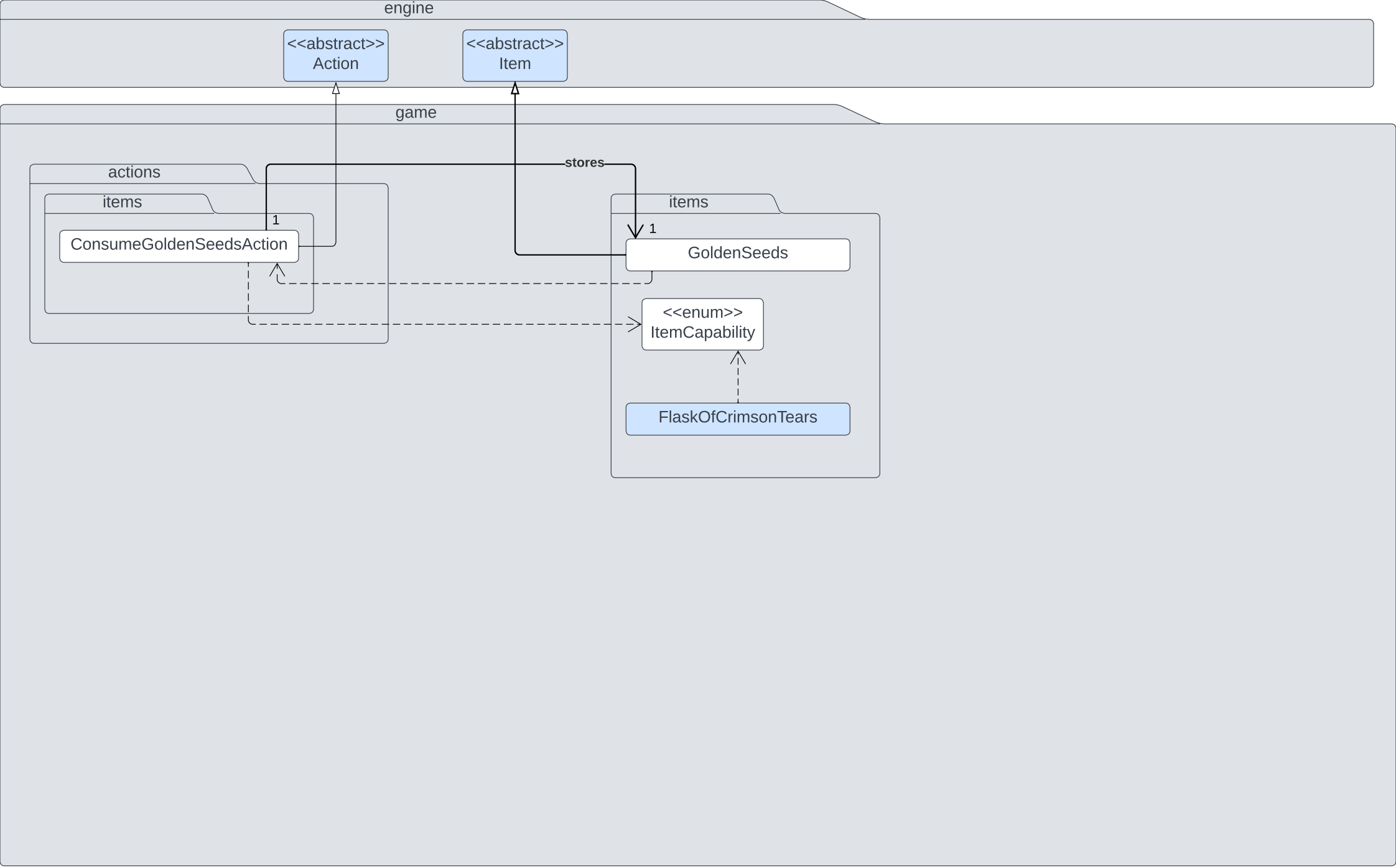
CL_AppliedSession11_Group1
Harshath Muruganantham
Ziheng Liao
Ho Seng



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

DISCLAIMER: Similar to how Assignment 3 builds upon Assignment 2, our Design Rationale for Assignment 3 builds upon our Design Rationale for Assignment 2. The Design Rationale for Assignment 2 can bet found inside docs -> Assignment 2 on the group's git repo.

Req 1

2 new environments are being added. These are the Cliff and the Golden Fog Gate. A new feature is also added called Fast Travel that allows the Player to be able to fast travel to existing Site of Lost Grace that they have already visited.

Design Goals:

1. The new environments added can re-use the old code present and would only require special abilities to be coded in
2. The Fast Travel should be implemented to utilize as much pre-existing code as possible and only new features should be coded in.

The new class "Cliff" extends 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 1. We use a few of the default methods from "Ground" such as canActorEnter and the tick methods so inheriting the class from "Ground" helps avoids repetitions (DRY). We override both methods to implement the special ability of the "Cliff" which is to kill the Player when the Player steps into a "Cliff". To do this, we added a capability called FALLS_OFF_CLIFF to the Player on construction. The "Cliff" checks using the canActorEnter method which checks if "Actor" trying to enter the "Cliff" has the capability. This method was chosen over checking of the player's display char in order to ensure OCP where if in the future, another class needed to be added which also cannot enter the Cliff, it can be easily added without much change to existing code. This also follows OOP principles as the "Cliff" is only checking for objects with the FALLS_OFF_CLIFF capability and is not aware of any particular object in the game when checking for the Actor. For the tick method, we override it by having it return a DeathAction with the Actor that entered the "Cliff". This follows SRP as the player's death is handled by a different class and also follows DRY as we are re-using code already present in the game, the "Cliff" returns an Action which is executed without interruptions from the "Player" class.

The "GoldenFogDoor" extends from the abstract class "Ground" as it shares common attributes and methods, and it is logical to abstract these identities to avoid repetitions (DRY). In particular, we override the canActorEnter and allowableActions methods as they are existing code, and we can re-use them to avoid repetitions (DRY). In order to follow SRP, we override the allowableActions method to return "TravelRealmsAction", that the user can select as part of their turn. "TravelRealmsAction" extends from the abstract class "Action" as we can take advantage of the functionality of the Action class (DRY) to implement the action of the Player travelling from one map to another while allowing for further extension and modification in the future (OCP) by ensuring that if any future class wants to be able to travel between maps, it can use this method without the need for repeated code. The travelling to

another map is implemented by overriding the execute method inherited from the "Action" class. This also follows DRY as we use existing code for any existing functionality, and we only add new code for new functionality.

When the player gets near a Lost Site of Grace, they will be given a choice to activate it by the SiteOfLostGrace through the use of ActivateSiteOfLostGraceAction. This is a newly created class which extends from the “Action” abstract class since they share common attributes and methods, and it is logical to abstract these identities to avoid repetitions (DRY). The use of this Action class helps us follow SRP as we are delegating the functionality of activating the Site to a new class. When a Site is activated, that site is added to a new class called “ActivatedSiteManager”. “ActivatedSiteManager” instantiates itself so that the same object (and by extension the same list of sites) are used on each addition/retrieval. This is needed in order to ensure the new “FastTravel” functionality to travel between activated sites. Another approach for keeping track of activated classes was to have the SiteOfLostGrace instantiate itself and to store all sites there. This method was however quickly omitted as this meant we would not be able to give different Sites different characteristics (such as names) and this approach prevented future extensibility of code where certain sites could not be travelled to. If the player decided to FastTravel to another SiteOfLostGrace, the current SiteOfLostGrace (the one the player is standing on) will call on the newly created class “FastTravelAction”. We decided to extend from this from the “Action” abstract class as we can implement the functionality of displaying the choice to the user in the UI and we can return the FastTravelAction from the SiteOfLostGrace which is in line with DRY as the Ground class has methods that return allowable actions.

The reset method in player was changed to account for the player being able to respawn in their last rested on “Site of Lost Grace”. The reset method in Player now checks if the player is resting near a site of lost grace (which is only possible if they are resting as a site of lost grace is covered by floors which enemies cannot enter and therefore kill the player) and if the player is, then the location of the SiteOfLostGrace is updated in the players’ lastVisitedSiteLocation attribute. An alternate approach to this was to update the location attribute in Player inside RestAction class. However, this required the use of down casting and hence was ultimately omitted as the use of down casting could bring unforseable errors on future extensibility to the code. Our chosen method also follows OOP principles of Encapsulation as the player is responsible for updating its own attribute and “RestAction” is not made aware of the Player attributes.

Harshath Muruganantham
Ziheng Liao
Ho Seng

Req 2

7 new classes were created this requirement. Noticeable classes are Dog, GodrickSoldier, HeavyCrossbow and EnemiesInSurrounding. Some goals aimed to achieve whilst completing the implantation are:

Design Goals:

1. Follow DRY and SOLID principles in implementation.
2. The new enemies added can reuse the old code present and will only require special abilities to be coded in.

The new grounds Cage and Barrack extends from abstract class “Ground” as they share common attributes and methods, and it is logical to abstract these identities to avoid repetitions (DRY). We decided to use the “EnemyFactory” class to spawn the enemies as this method allows the ground classes to follow SRP as the environment ground classes will not be responsible for spawning Actors. Our original idea was to implement spawning inside the tick method inside the environment classes, however this method was quickly omitted in order to better follow encapsulation laws. Our chosen method helps follow encapsulation laws as the environments are not made aware of the actor classes they are creating.

The new enemy classes Dog and GodrickSoldier are inside the “soldiers” package and extend from abstract class StormveilEnemy as they share similar implementation, specifically the fact that they will share similar methods; relating behaviour, the reset method and most importantly, they aren’t allowed to attack each other. This helps us meet design goal 2. An alternative way to implement this which was considered was to have Dog extend from the abstract class “LimgraveDog” like GiantDog and LoneWolf as its name suggest that it’s related to the “LimgraveDog” abstract class. However, in terms of implementation, it would not follow OCP as in the future if we have an additional class that Dog cannot attack, we would have to have another ENUM “Status” capability and more conditionals to check for that capability. So, by creating a StormveilEnemy abstract class, we ensure that StormveilEnemies won’t attack each other in the future and if we were to add more enemies with this functionality, it would be easy to extend. 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, Dogs can’t attack GodrickSoldiers). 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. 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.

The weapon Heavy Crossbow inherits from the abstract class "WeaponItem" 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. 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. This class has an additional capability called range_attack to signify that it can complete a ranged attack and an additional method to this class is the "tick" method to identify the enemies around the surrounding each turn and giving an option to attack for the player as it will add the action AttackAction to the weapon. This could not be done inside the playTurn method inside Player as the playTurn method is used to display the different options that the Player is allowed to take. Furthermore, if we were to have the range_attack implemented inside the Player, we would have to check for each individual weapon that has the capability of a range_attack which is not feasible due to it breaking OCP. The benefits of having it inside the "tick" method of the weapon is that with multiple weapons in the game, each weapon could have a different range, and it will scout for the enemies within that range inside the weapon class as this follows SRP as well and the player will not have to worry about the range_attack. Heavy Crossbow also implements Purchasable and Sellable. 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. The Purchasable and Sellable interface functions as a way to alert the trader that this weapon can be sold to a trader. This is done in order to better fit in with existing code and work without over-modification to existing code, better following OCP and DRY.

To identify enemies surrounding the Actor holding the weapon, a new class called EnemyInSurrounding was created solely to identify the enemies around the surrounding and returning the Actors in that surrounding as this method helps follow SRP and OCP as opposed to finding the enemies inside the weapon. This class was also used in all future uses of range_attack helping us follow DRY principle. Another benefit with creating this as its own class is different weapons have different ranges, so the number of locations to identify whether there is an enemy there will vary and rather than having the same code with repeatedly slightly different parameters in different weapon classes, we are following the DRY principle by having it as its own class where it considers the range. The approach we used to find the nearby enemies in range was to iterate the "x" range values and the "y" range values. Another approach we originally thought was to use the exit method already given in the code. This method helps us follow DRY principles as we are using code already present. However this method was ultimately omitted as when using this method we would have had to recursive for loops for each range (ie, 2 recursive for loops for the range of 2, 3 recursive for loops for the range of 3, etc). This would work for our current implementation but for future

extensibility of the code it would be problematic as we would have to hard code recursive for loops as the range increases. Therefore, this method was omitted as it does not help follow OCP.

Looking more specifically into GodrickSoldier, it uses the weapon HeavyCrossbow which uses range attack. Inside GodrickSoldier, we have a conditional to check whether it is carrying a weapon that has the capability of a range_attack. If it does, it will extend its exits to the range of the weapon the enemy is carrying. A different implementation was considered where adding behaviours would be done inside the weapon class, however this was omitted due to the fact that down casting would have to be made as the behaviour attribute in enemies cannot be accessed through the Actor class. Furthermore, down casting would not always work as well if the system were to be extended without conditionals.

Harshath Muruganantham
Ziheng Liao
Ho Seng

Req 3

The diagram shows a few new classes being added. Noticeable classes include Godrick the Grafted, new weapon classes, Golden Runes and Finger Reader Enia. We have also implemented the optional "purchase vigor" feqature in site of lost grace. Some goals aimed to achieve whilst completing the implementation are:

Design Goals:

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

The new enemy class, "Godrick the Grafted" sits within the 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 allows this class to attack. This class has two phases with the first phase being when its hit-points are above, or greater than 50% of the max points. This is the normal phase Godrick is in when he is created. At the start of the playTurn method, Gorick's hit points are checked, and if the hit points are less than half of his max hitpoints, then we return to a new action we have created called half health action. Inside this half health action, is that we remove the Axe of Godrick weapon from Godrick's inventory and swap it with the grafted dragon weapon instead. We have chosen to do this inside HalfHealthAction instead of doing it inside the playTurn method to better adhere to SRP as, it was better to segregate the functions of the class in order to avoid GodrickTheGrafted becoming a god class. Using HalfHealthAction also allows, in the future, to add more capabilities to Godrick when half health is reached, where in addition of getting a new weapon, Godrick could also receive special health boost adhering to OCP. HalfHealthAction extends from Action as it allows the "world" class to run this action when added without any extra addition of code helping us better follow DRY and OCP.

Godrick The Grafted has access to two specific weapons in its "weapons list" attribute (across its lifetime), an attribute that all Actors have (OCP). These weapons are new classes created called "Axe of Godrick" and "Grafted Dragon". "Axe of Godrick" and "Grafted Dragon" both inherit 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). "Axe of Godrick" and "Grafted Dragon" 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). Since the description of this

“Spinning Attack” is very similar to the “Spinning Attack” already implemented in the game, we use that same class for these new weapons classes as well to better adhere to OCP and DRY. Through creating “Axe of Godrick” and “Grafted Dragon” as its own class and not creating it within the constructor of a Godrick The Grafted, we ensure that Godrick The Grafted can use these weapons without knowing what it is exactly (following Object-oriented principles) and so that “Axe of Godrick” and “Grafted Dragon” 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 “Axe of Godrick” and “Grafted Dragon”. 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 “Godrick the Grafted” attacks another enemy, it has a 50% chance of using SpinningAttackAction. This uses code already implemented inside AttackAction rather than having a new separate implementation inside the weapons classes in order to better adhere to OCP and DRY. These new weapons can be sold by the player to a trader if they have it in their inventory. As the “Sellable” interface already exists, which functions as a way to alert the trader that this weapon can be sold to a trader, the weapons classes implement “Sellable” interface in order to better fit in with existing code and work without over-modification to existing code, better following OCP and DRY.

Godrick The Grafted also implements the “resettable” interface as this character can be reset to their original position with full health is they have not been defeated. This reset function only happens when the player rests at a site of lost grace. Godrick the Grafted also implements the SpeacialDeath interface as it does not die like a normal enemy, whereas when it dies, it becomes a Lost Site of Grace. This method of adding a SpecialDeath interface was chosen over implementing these methods inside DeathAction to avoid typecasting and down casting 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. A choice was made to implement both these interfaces instead of implementing just the resettable interface and having if-conditions in resettable to account for death, as implementing both these interfaces helps us better adhere to ISP and creates easy to read code. This also helps ius follow design goal 1.

When “Godrick the Grafted” is defeated, a “Remembrance of the Grafted” is dropped. “Remembrance of the Grafted” extends from the “Item” class as it requires a few of the methods already implemented in Item abstract class such as being able to be picked up, so it is logical to abstract these identities to avoid repetitions (DRY). “Remembrance of the Grafted” can also be sold to a trader for 20000 runes. To implement this feature, we have decided to have “Remembrance of the Grafted” implement the “Sellable” interface. This is because the “Sellable” interface already works with all current traders in providing a sell price for the item. By implementing this specific interface, and not having the sell price hard coded into the Item’s constructor, we were able to add this new item class to the current game without high levels of modification to existing code helping us follow OCP principles and DRY.

Another item added to this game in this requirement are the Golden Runes. “Golden Runes” also extend from the “Item” abstract class as it requires a few of the methods already implemented in Item abstract class such as being able to be picked up (LSP), so it is logical to abstract these identities to avoid repetitions (DRY). This also helps us follow design goal 2. When the Golden runes are picked up, they can be consumed by the actor. To implement this, we had to create a new Action class called “ConsumeGoldenRuneAction”. We chose to extend this class from the Action abstract class as opposed to creating a new class in order to use some of the inbuilt methods within the “Item” class that “Golden Runes” can use. This includes the “addAction()” method that the Item abstract class has. By using this method, we can avoid code repetition and easily jump to the consume action method using this method in the “Golden Rune” class when the item is picked up. The addition of the runes to the player’s purse only happens when they consume the runes. This is taken care of in the “consumeGoldenRunes” action in order to better follow SRP and segregate functions between different classes to avoid creating a god class.

Initially the reason for “Finger Reader Enia” 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 “Finger Reader Enia”, 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. The inventory method was used as, if not for the inventory, downcasting would have needed to be used to retrieve the sell costs of the weapons, which may create unforeseen problems in future extensibility of the code. A new interface was created here called the “Exchangeable” interface, which is implemented in an item if that item can be exchanged for any other weapon items. This is implemented inside of the “Remembrance of the Grafted” class as that class can be exchanged for the two weapons Godrick the Grafted carries. We had chosen to implement this as a separate interface instead of a “Tradeable” interface to better follow ISP as some items, like “Remembrance of the Grafted” can be only exchanged and sold and not exchanged. This “Exchangeable” interface contains a method which returns all the weapons this item can be exchanged for. “Finger Reader Enia”, also implements an “Exchange” interface, which ccontains a method which creates an instance of the “ExchangeAction”. This segregation was done in order to follow ISP and OCP, where if in the future another trader needs to be able to exchange items, it can easily be done by implementing the “Exchange” interface without the need for code repetition.

A new class to handle the actions of exchanging were created called “ExchangeAction” which extends from Action as it is needed to implement the actions being ran which the abstract Action class does for us. “ExchangeAction” takes care of removing the required item from the player’s inventory and adding the exchanged item. The possibility of implementing these functions as methods inside “Finger Reader Enia” 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 “Finger Reader Enia” 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).

The last feature implemented in this requirement is the ability to increase maximum hit points of the player by buying a vigor item from the Site of Lost Grace for 200 runes. This is mainly done inside a new class called “IncreaseMaxHitPointsAction”. This new class extends from Action as they share common attributes and methods, and it is logical to abstract these identities to avoid repetitions (DRY). By inheriting from the “Action” abstract class, we can also use this new class inside the Site of Lost Grace without any further addition to the code, as this class already extends from “Ground” which has a default “addAction()” method which can be easily used to add this action to the environment, without the need for additional code. Inside this action, we have set the constructor to private, and added an instance variable, so that the same instance will be returned each time the method is called. This was done in order to increment the price of buying a vigor by 100 each time, regardless of the location that vigor was bought from. This method of creating an instance was chosen over creating a new class which manages the Vigor attribute and keeps track of the price in order to reduce overall code and make it easier to read and understand. This method also helps decrease convoluted code with many dependencies.

Harshath Muruganantham
Ziheng Liao
Ho Seng

Req 4

The diagram shows a few new classes being added. Noticeable classes are AstrologerStaff, SummonSign, SummonAction, Ally and Invader. Some goals aimed to achieve whilst completing the implantation are:

Design Goals:

1. Have new implementations extend from existing systems
2. Follow DRY and SOLID principles in implementation.

The class Archetype has been changed into an abstract class now due to the fact that multiple player classes (Samurai. Astrologer) extend from it where the details of each class are in the player classes themselves. This promotes abstraction and reduces the complexity it requires to read and maintain code.

A new Astrologer class was created and extends from the Archetype since the Archetype abstract class was intended to be more extensible when new Archetype classes are introduced (OCP) whilst also following LSP when summoning an Ally/Invader with a class from Archetype. It is beneficial as inside SummonSign we are not sure of what to summon (Samurai, Wretch, etc), but we know it is an Archetype so we can follow LSP to access the Archetype's attributes. Furthermore, in terms of maintaining the code if there were ever any issues in the future, it would be simple to identify what needs to be addressed and changed since the Astrologer class follows SRP.

Astrologer's Staff, similar to the weapon HeavyCrossbow is a weapon that extends from the abstract class "WeaponItem" as the purpose here is to take advantage of the Open/Close principle implemented from the previous developer of this code and also satisfies our design goal 1. Similarly, since Astrologer's Staff also uses ranged attack, it also calls upon the EnemyInSurrounding to identify the surrounding enemies. This time the range is 3, which is different to the range of HeavyCrossbow which further suggests that the current implantation of EnemyInSurrounding is an acceptable method of implementation as it follows DRY and SRP as well as the fact that it is extensible. This class also has an additional capability called range_attack to signify that it can complete a ranged attack and an additional method to this class is the "tick" method to identify the enemies around the surrounding each turn and giving an option to attack for the player as it will add the action AttackAction to the weapon. This could not be done inside the playTurn method inside Player as the playTurn method is used to display the different options that the Player is allowed to take. Furthermore, if we were to have the range_attack implemented inside the Player, we would have to check for each individual weapon that has the capability of a range_attack which is not feasible due to it breaking OCP. The benefits of having it inside the "tick" method of the weapon is that with multiple weapons in the game, each weapon could have a different range, and it will scout for the enemies within that range inside the weapon class as this follows SRP as well and the player will not have to worry about the range_attack. Astrologer's Staff also implements Purchasable and Sellable. 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. The Purchasable and Sellable interface functions as a way to alert the trader that this weapon

can be sold to a trader. This is done in order to better fit in with existing code and work without over-modification to existing code, better following OCP and DRY.

Summon Sign is a ground with the ability to interact with it as such it extends from Ground to reduce the amount of code that needs to be rewritten and again following the DRY principle and taking advantage of OCP implemented by the previous developer. When interacting with it, it will create a new SummonAction to summon Allies/Invaders as it will follow SRP. SummonAction will be responsible for deciding whether to spawn an Ally or an Invader. An alternative way was suggested as well which was to have the spawning done inside the SummonSign, however this was omitted due to the fact that it breaks SRP since SummonSign should only be responsible for it being on the map and being the class to interact with the Player to allow the Player to decide what to spawn.

With Ally and Invader, they both extend from Actor as they perform an action each turn and have functionalities of an Actor. During design, it was considered that Ally and Invader would extend from an abstract class since Ally and Invader are separate from enemies and different from Player. The biggest difference would be that they can attack all enemies indiscriminately. However, Ally and Invader do not share the same similarities as Ally can attack anything other than the Player and other Allies whereas Invader can attack anything other than other Invaders. Due to this difference, having an abstract class for Ally and Invader was not feasible. In terms of implementation, it's quite similar to other enemies such that they both use behaviours to dictate their actions. Both Ally and Invader 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 Allies and Invaders to attack. Inside the playTurn method the Invader has the ability to follow the player if nearby and attack an Actor if nearby. Whilst the Invader is deciding to attack, there is a constraint where the Invader cannot attack another character of the same class. Likewise with Ally where it is unable to attack other Allies and the Player. To distinguish what they are and aren't allowed to attack, these two classes and the Player class have imported a capability. Inside the playTurn method of both these classes, there will be a condition that checks for the capability status of the Actor it is to attack and will only include the AttackBehaviour if the status of the Actor isn't of its kind.

The two classes Invader and Ally also implement the interface SpecialResettables which is different to Resettables due to fact that they are only removed from the map when the Player dies and aren't removed from the map if the Player decides to rest. Due to the different functionality, a new interface had to be created. We also considered using the previous already existing interface Resettables, but that would be breaking ISP which goes against design goal 2. To actually reset the Ally and Invader, they will be added to the ResetManager. The decision not to create a new ResetManager to handle the SpecialResettables was because it still followed SRP since the ResetManager is still handling resets, only difference now being a slightly different kind of reset. As such, a new method was added inside the ResetManager called specialRun, which when called will despawn all Actors that implement the SpecialResettable interface.

Harshath Muruganantham
Ziheng Liao
Ho Seng

Req 5

The diagram shows a few new classes being added. Noticeable classes include the Golden Seeds Item and the "ConsumeGoldenSeedsAction". Some goals aimed to achieve whilst completing the implementation are:

Design Goals:

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

"GoldenSeeds" are scattered around the 4 maps in Elden Ring. The player has the ability to pick it up when they find it. When the player picks it up, the item is added to their inventory. When it is added to the inventory in the next turn, the player has the ability to consume it. When the player consumes it, the maximum uses of the FlaskOfCrimsonTears increases by 2 and the number of uses is replenished to the maximum number. Once the player uses it, golden seeds is removed from their inventory. Since the player has the ability to pick up the "GoldenSeeds", similar to an item, "GoldenSeeds" extend from the item abstract class in order to avoid code repetitions (DRY). When the "GoldenSeeds" are picked up, they can be cosnsumed by the actor. To implement this, we had to create a new Action class called "ConsumeGoldenSeedAction". We chose to extend this class from the Action abstract class as opposed to creating a new class in order to use some of the inbuilt methods within the "Item" class that "GoldenSeeds" can use. This includes the "addAction()" method that the Item abstract class has. By using this method, we can avoid code repetiotion and easily jump to the consume action method using this method in the "GoldenSeeds" class when the item is picked up. The "ConsumeGoldenSeedAction" is responsible for increasing the number of uses for the "FlaskOfCrimsonTears". By doing this inside a new action, as opposed to have the "Golden Seeds" take care of this, we are adhering to SRP where the "GoldenSeeds" has a singular function of being a consumable item and we are also adhering to OCP where, if in the future, another class needs to be able to increment the uses of "FlaskOfCrimsonTear", they would be able to use the existing method without needing to rewrite the code separately, also helping follow DRY principles. We use capabilities to increment the max uses of the Flask. Inside "ConsumeGoldenSeedAction", INCREASE_MAXIMUM_USES capability is added to the flask. Inside the flask, the default "tick()" method is used to check at each turn wether the flask has this capability and then increment its max uses if it does, removing the capability at the end. This method was chosen specifically as opposed to incrementing the maximum uses inside the action class itself, as doing the second method would require the use of down casting to access attributes inside the flask. Down casting could result in unforeseeable errors in future extensions of the code, so this method was ultimately disregarded.

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 code for REQ 1	Code review	08/05/2023	Harshath	DONE	12/05/2023	Started early. Got the new grounds done. Did the fast travel later down the line	
Draft code for REQ 2	Code review	09/05/2023	Ziheng	DONE	13/05/2023	Added new grounds and enemies but waited for help for the range weapon implementation	
Draft code for REQ 3	Code review	14/05/2023	Harshath	DONE	14/05/2023	Completed the new boss. Decided not to copmlete the dropping fire due to excessive time and debugging needed to perfect it	
Draft code for REQ 4	Code review	14/05/2023	Ho	DONE	14/05/2023	Completed Allies and Invaders as well as the new weapon and archetype. Did not take too long as previous implementation allowed for extensibility	
Submit the google form for creative REQ	Discussion	19/05/2023	Harshath	DONE	19/05/2023	Used Ziheng's email to submit the the form	
Draft code for REQ 5	Code review	18/05/2023	Harshath	DONE	18/05/2023	Made a start to the creative requirement. Decided to complete the Golden Seed creative requirement due to questions and clarifications already in place on the Ed forum	
Work on code for REQ 1	Code review	09/05/2023	Harshath	DONE	12/05/2023	Finalised REQ 1	
Work on code for REQ 2	Code review	10/05	Ziheng	DONE	15/05	Finalised REQ 2	
Work on code for REQ 3	Code review	14/05/2023	Harshath	DONE	14/05/2023	Finalised REQ 3	
Work on code for REQ 4	Code review	14/05/2023	Ziheng	DONE	14/05/2023	Finalised REQ 4	
Work on code for REQ 5	Code review	18/05/2023	Harshath	DONE	19/05/2023	Finalised REQ 5	
Finalising all the code	Code review	20/05/2023	EVERYONE	DONE			
UML for REQ 1	UML diagram	20/05/2023	Harshath	DONE			
UML for REQ 2	UML diagram	18/05/2023	Ziheng	DONE	18/05/2023	Standard UML diagram. Got it doubled checked by Harshath	
UML for REQ 3	UML diagram	20/05/2023	Harshath	DONE	20/05/2023	Initially a bit of miscommunication on who was going to complete it. But eventually completed	
UML for REQ 4	UML diagram	19/05/2023	Ziheng	DONE	19/05/2023	Standard UML diagram. Got it doubled checked by Harshath	
UML for REQ 5	UML diagram	20/05/2023	Harshath	DONE	20/05/2023	Standard UML diagram. Got it doubled checked by Ziheng	
Design rationale 1	Design rationale	20/05/2023	Ho	DONE	19/05/2023	Design rationale completed. Getting Harshath to go over it	
Design rationale 2	Design rationale	17/05/2023	Ziheng	DONE	18/05/2023	Design rationale completed. Getting Harshath to go over it	
Design rationale 3	Design rationale	20/05/2023	Harshath	DONE	20/05/2023	Design rationale completed. Getting the team to go over it	
Design rationale 4	Design rationale	17/05/2023	Ziheng	DONE	19/05/2023	Design rationale completed. Getting Harshath to go over it	
Design rationale 5	Design rationale	20/05/2023	Harshath	DONE	20/05/2023	Design rationale completed. Getting the team to go over it	