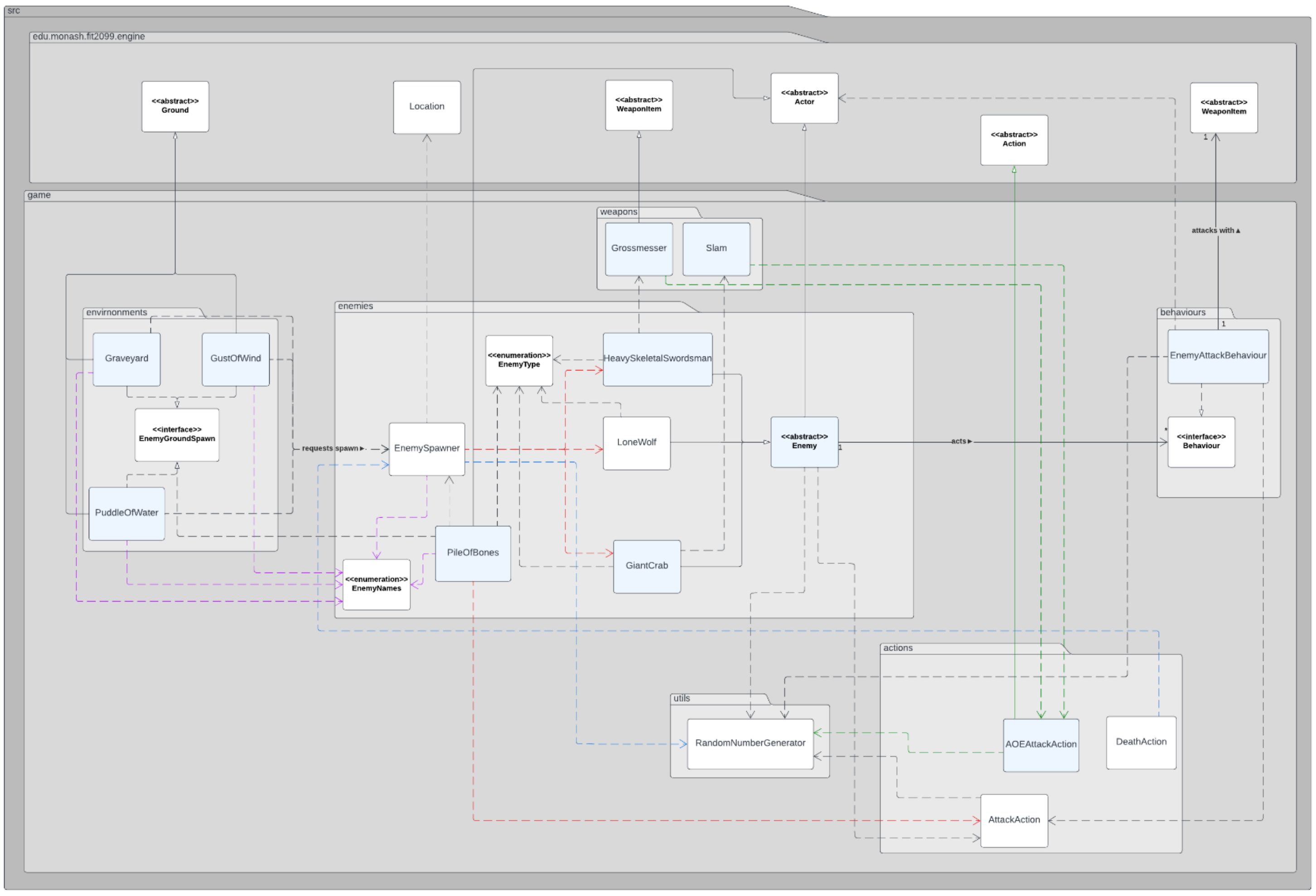
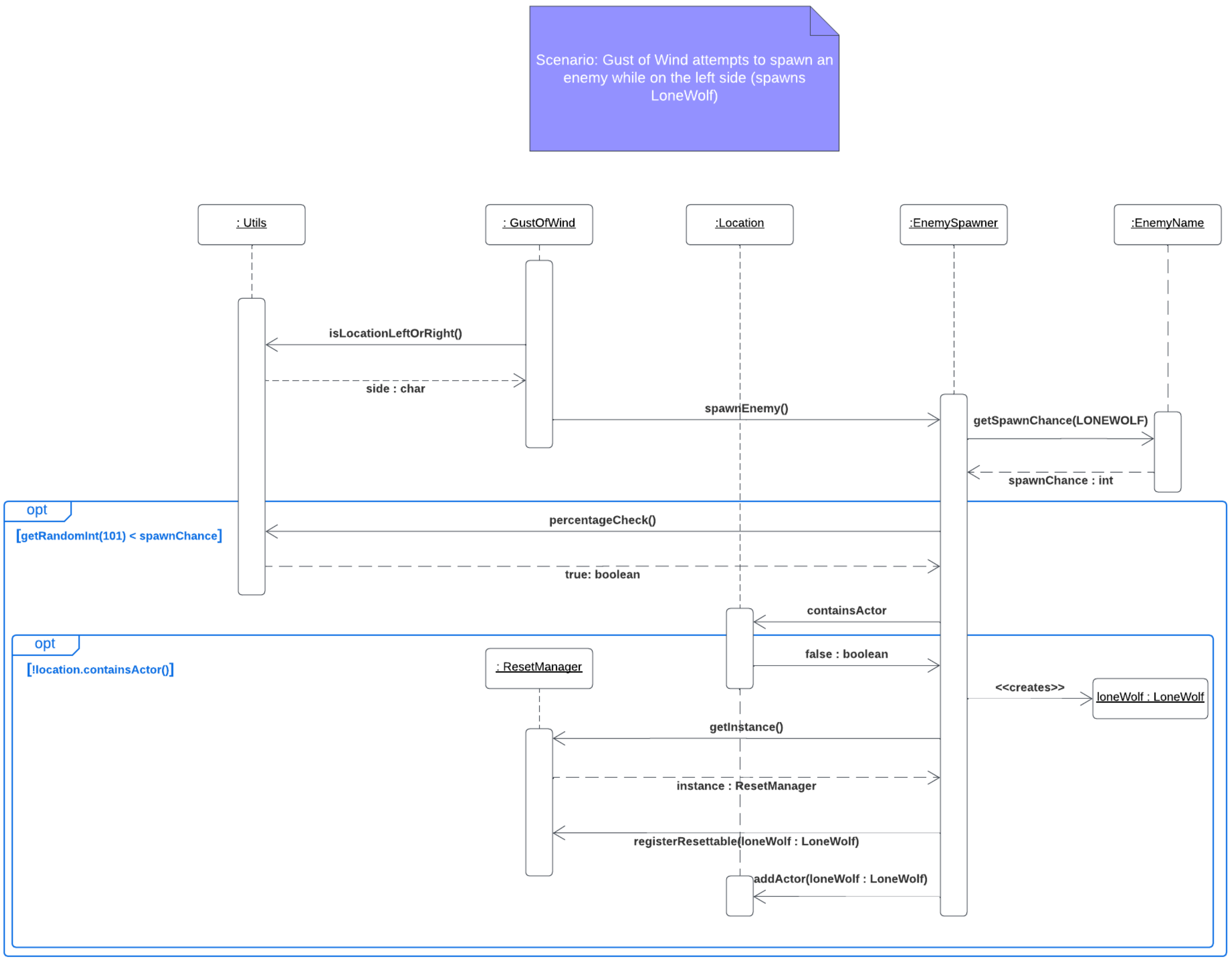
FIT2099 Assignment 2 Master Document

Applied Session 3, Group 8

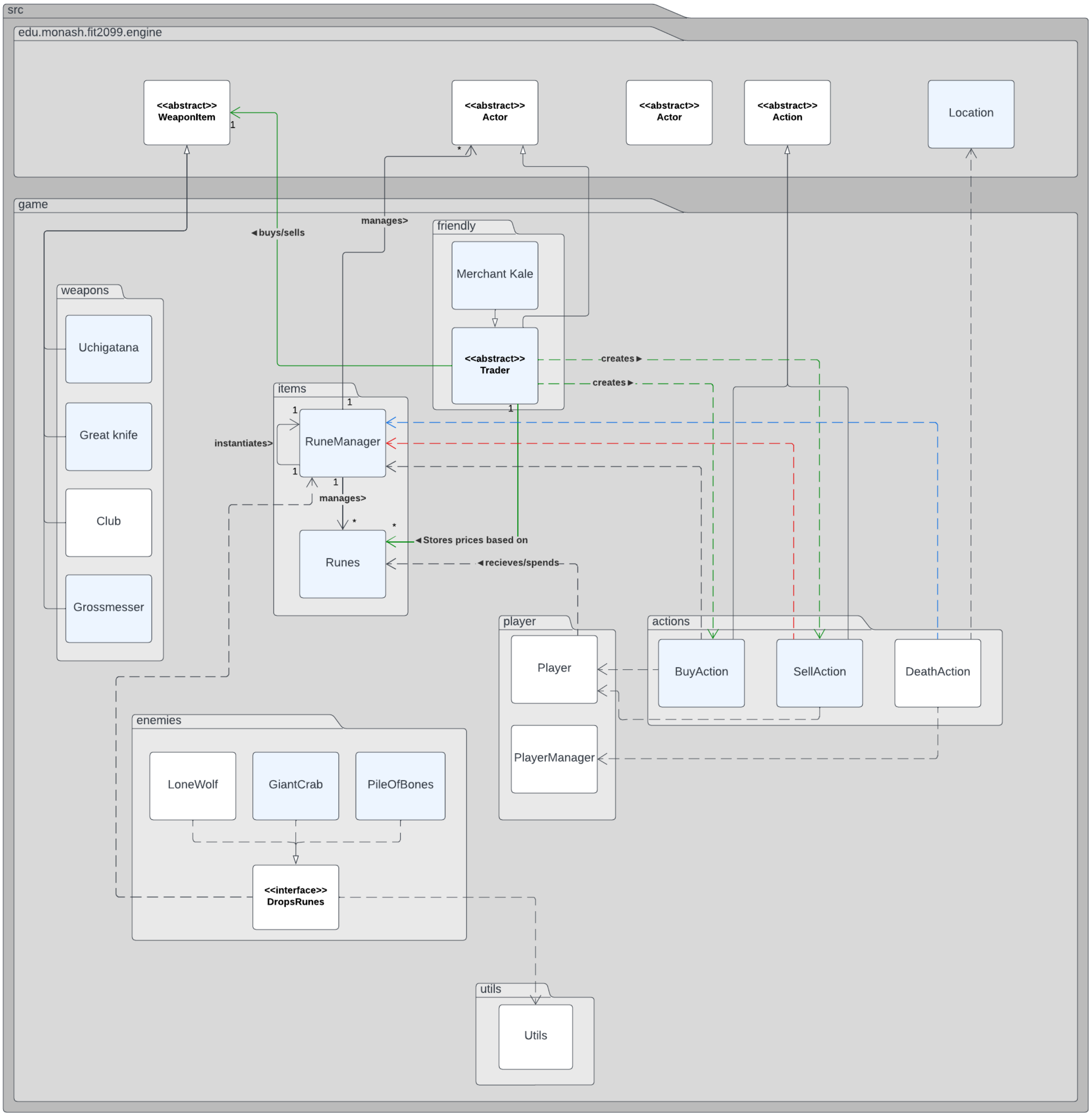
Req 1 UML Diagram:



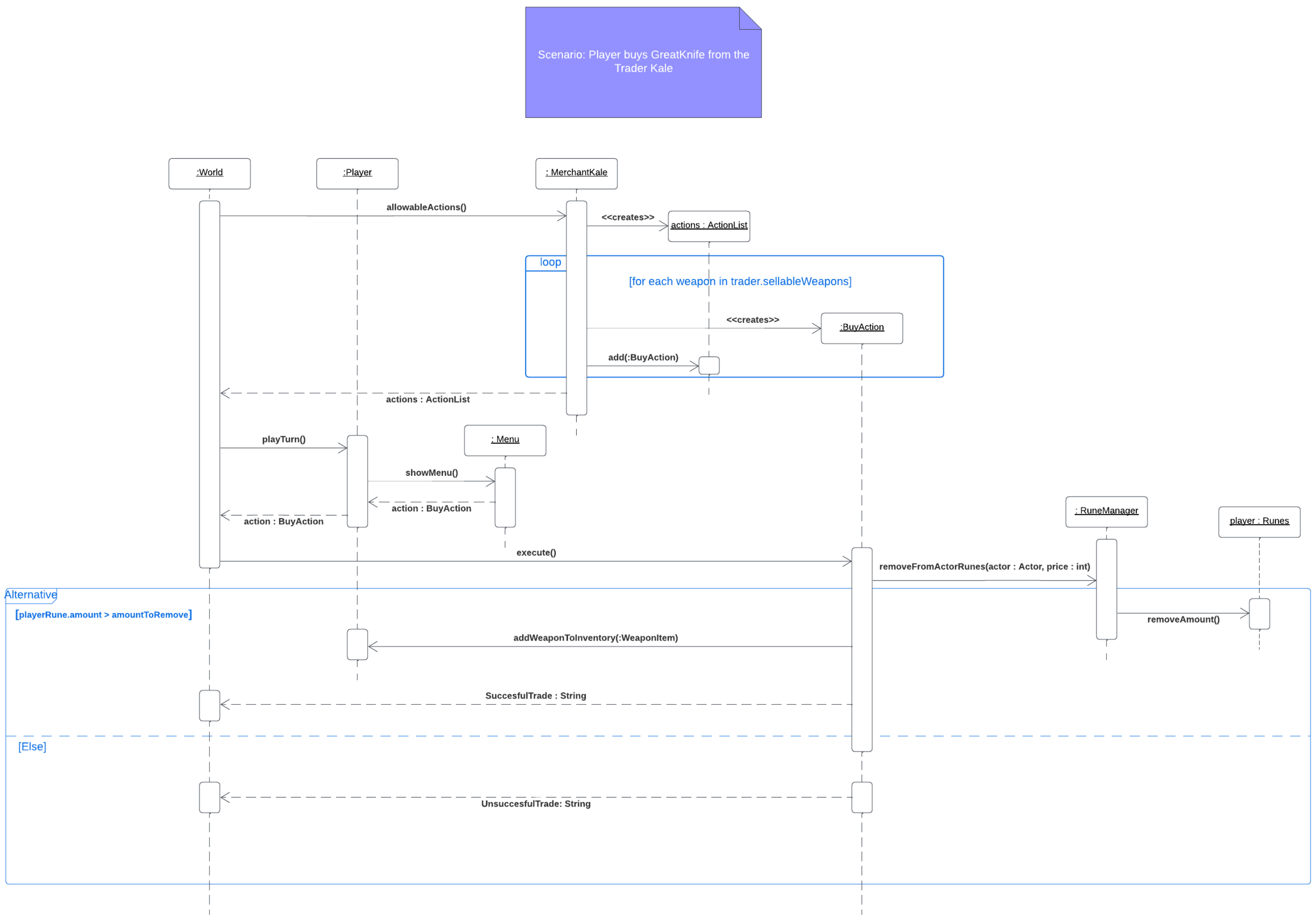
Req 1 UML Sequence Diagram:



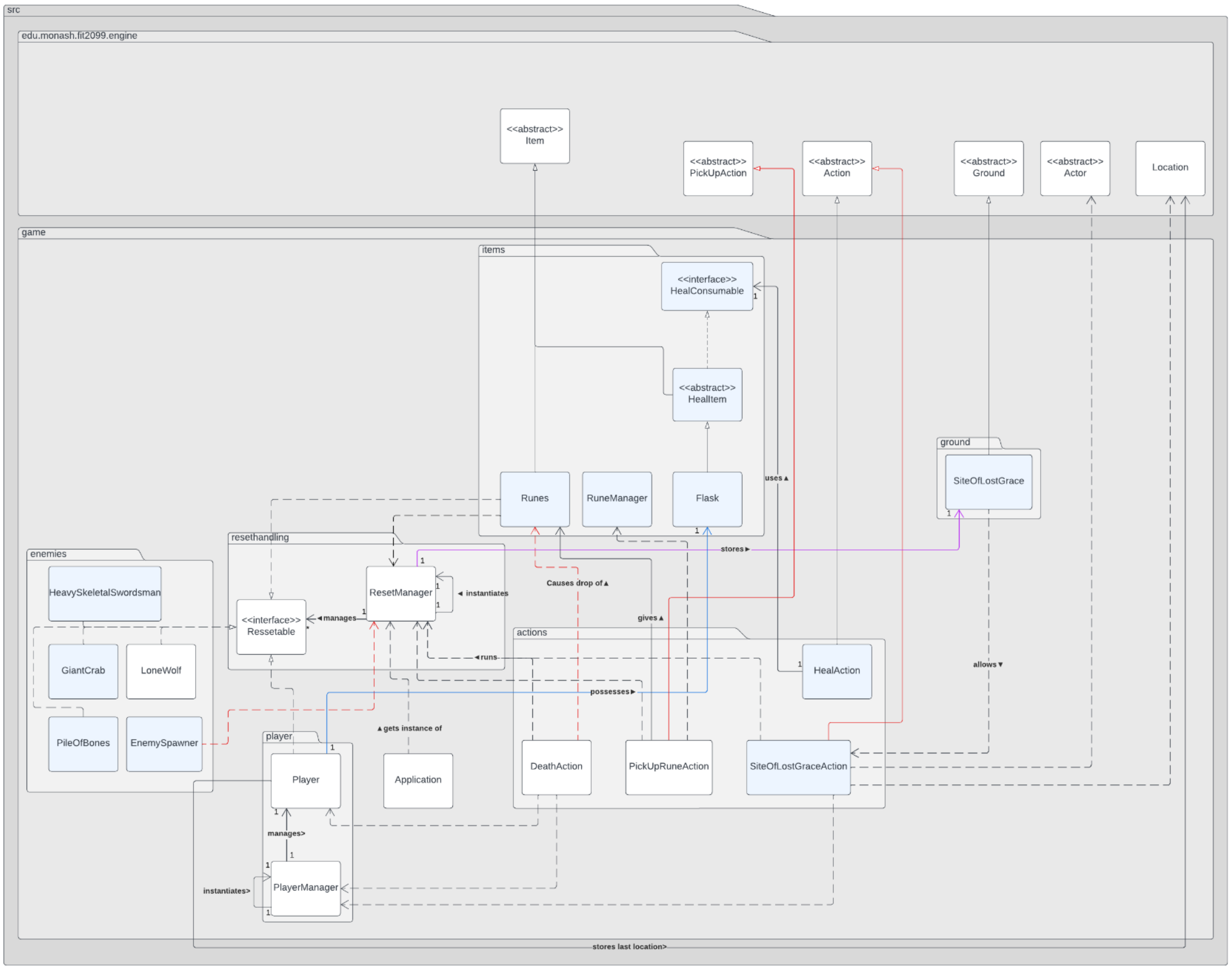
Req 2 UML Diagram:



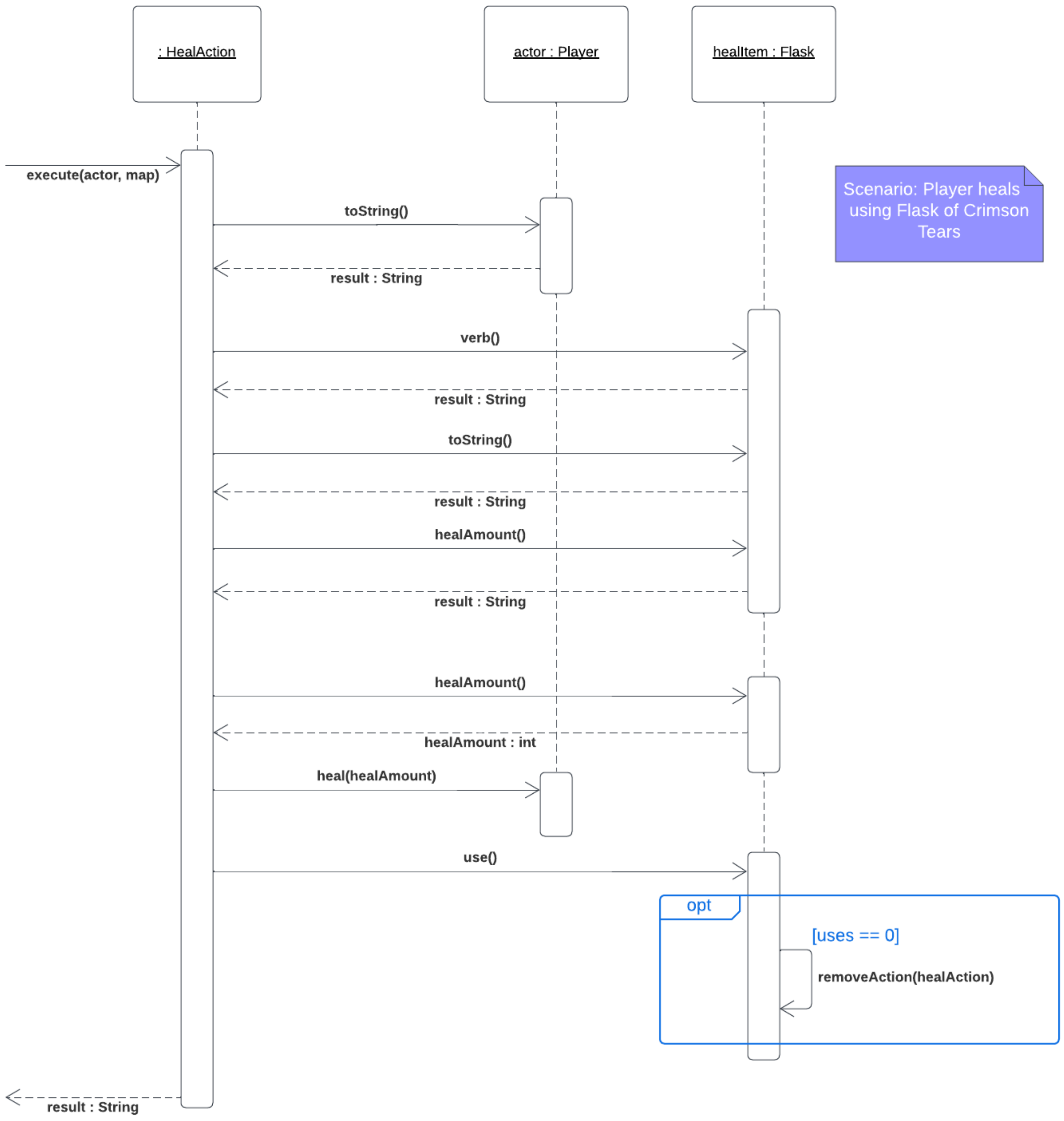
Req 2 UML Sequence Diagram:



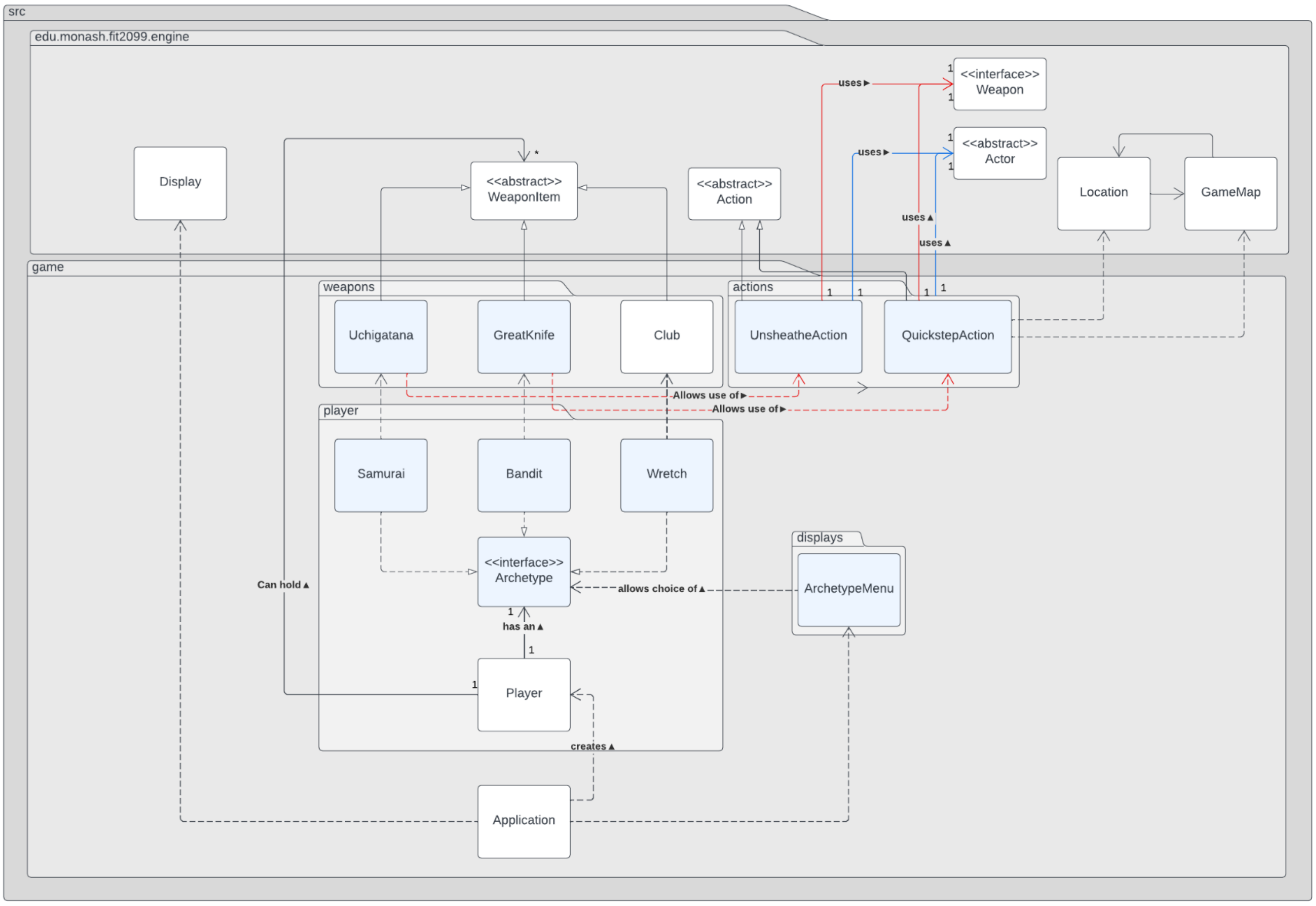
Req 3 UML Diagram:



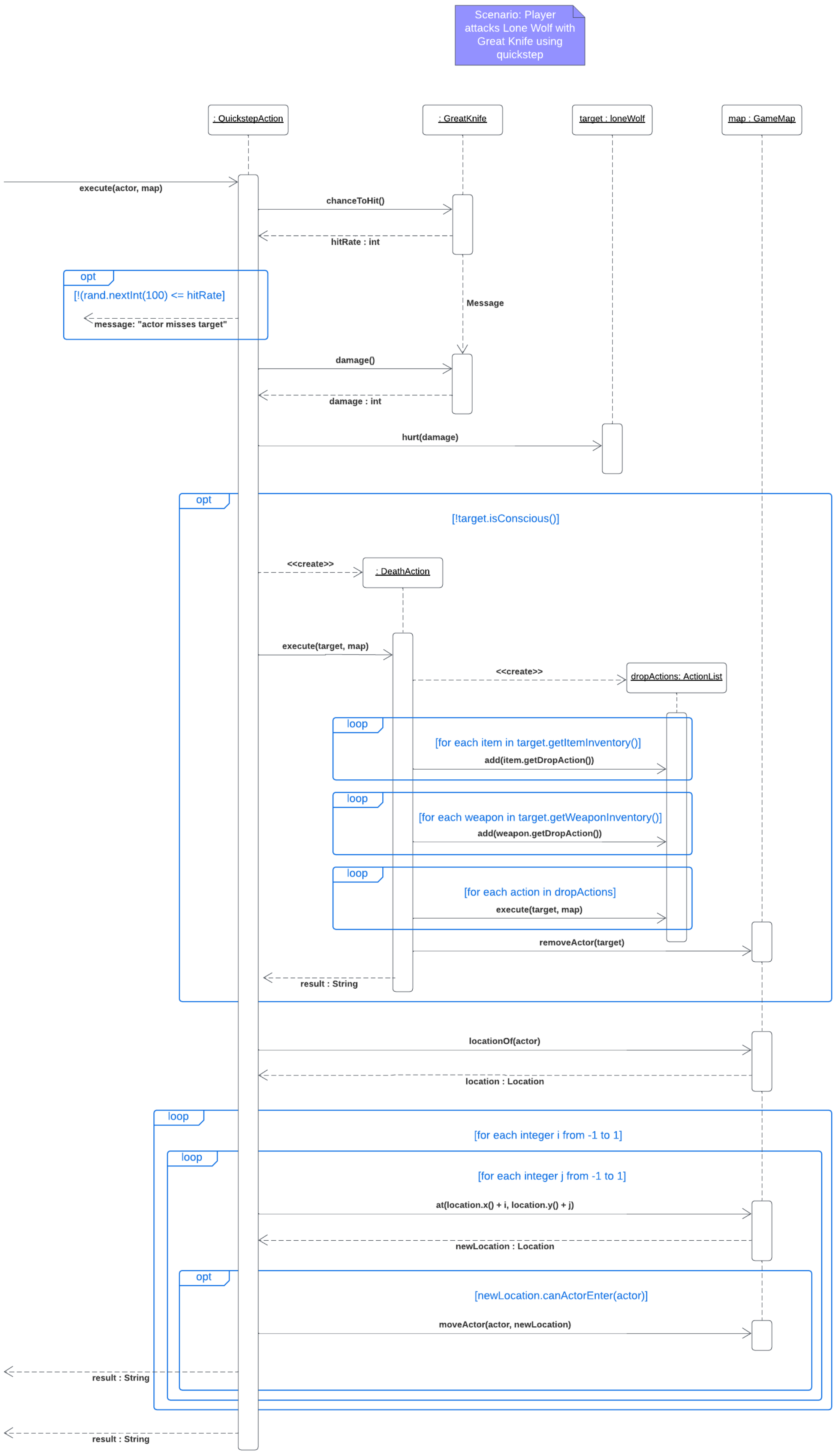
Req 3 UML Sequence Diagram:



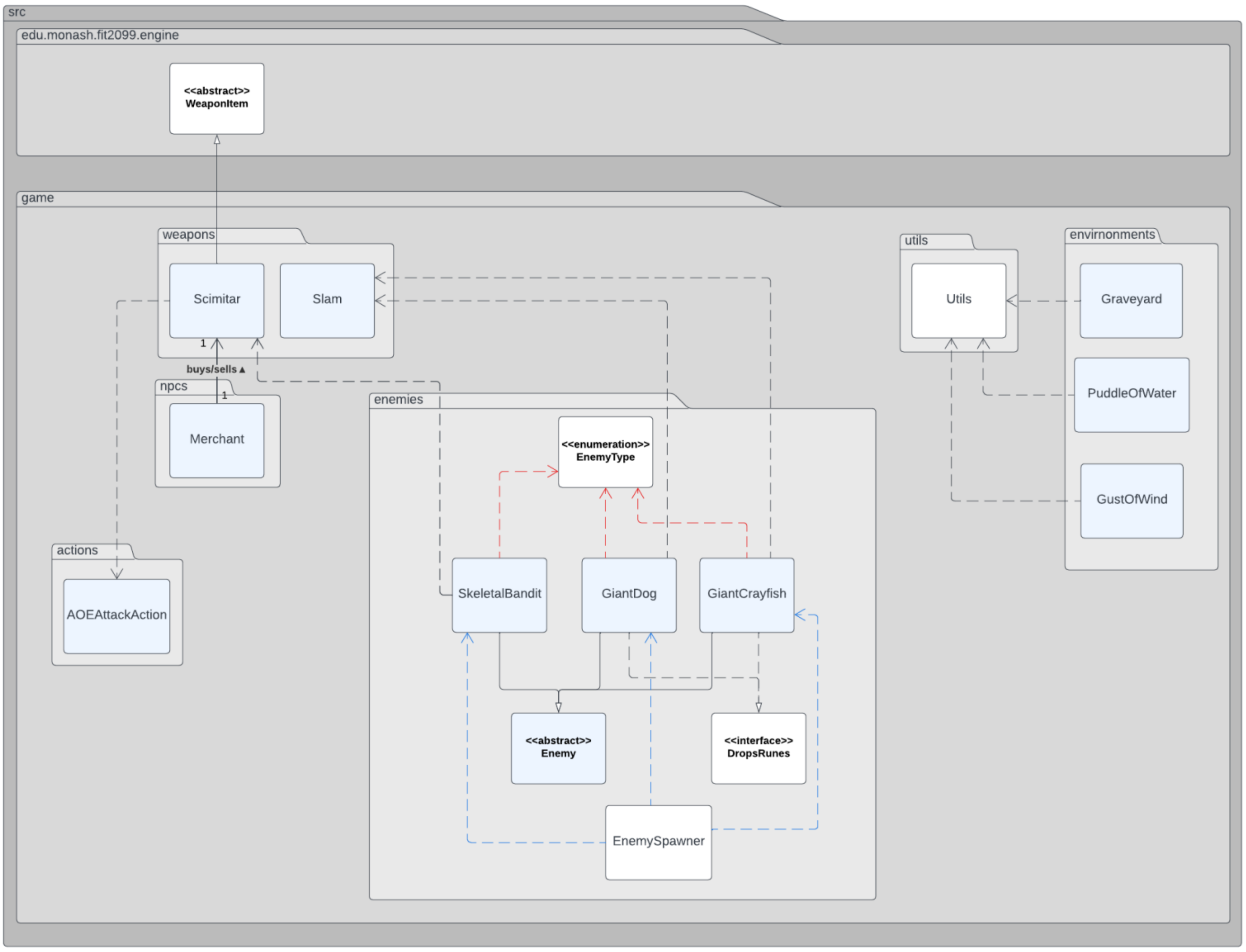
Req 4 UML Diagram:



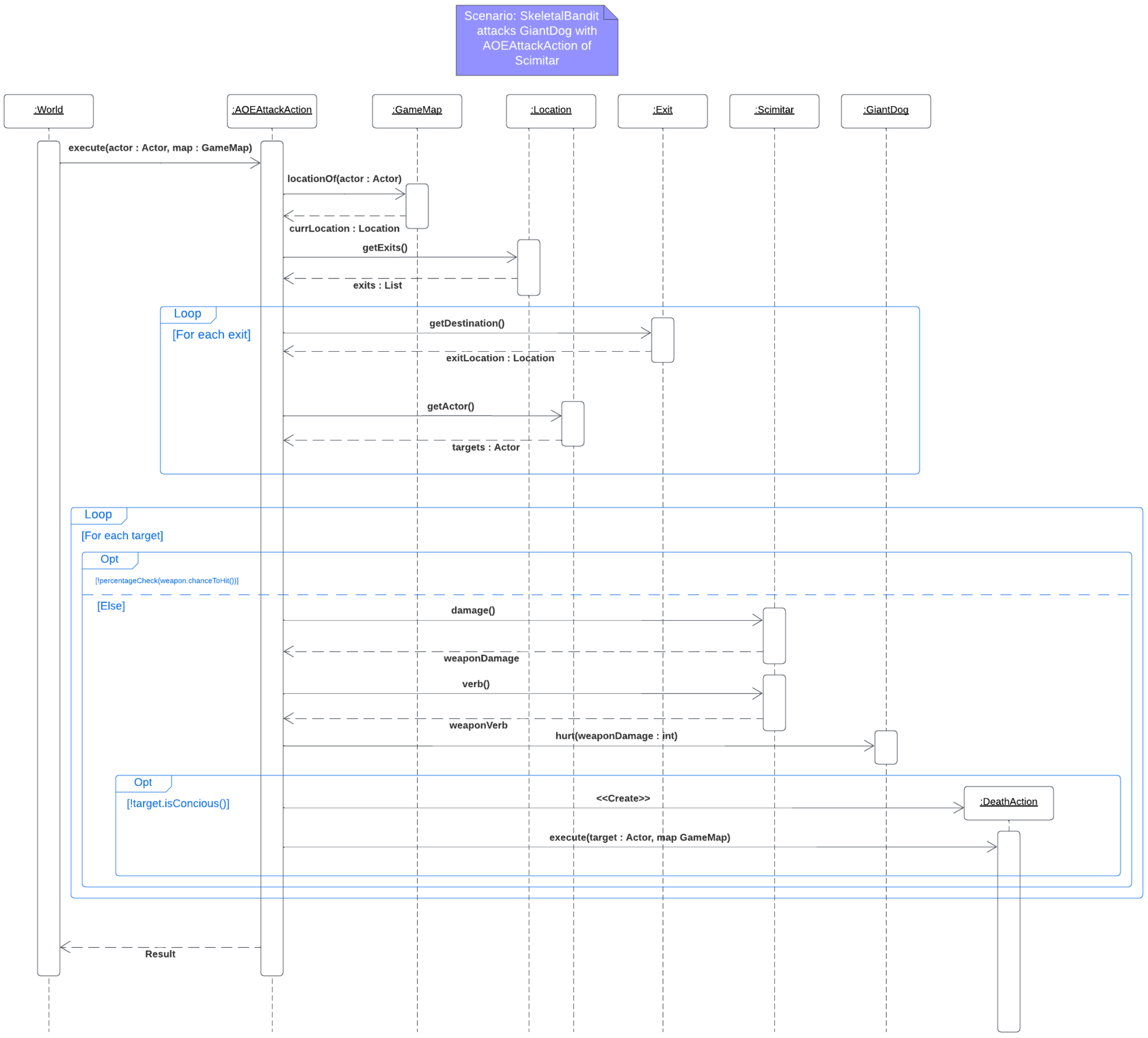
Req 4 UML Sequence Diagram:


Scenario: Player attacks Lone Wolf with Great Knife using quickstep
: QuickstepAction
: GreatKnife
target : loneWolf
map : GameMap
execute(actor, map)
chanceToHit()
hitRate : int
opt
[!(rand.nextInt(100) <= hitRate]
message: "actor misses target"
Message
damage()
damage : int
hurt(damage)
opt
[!target.isConscious()]
<<create>>
: DeathAction
execute(target, map)
<<create>>
dropActions: ActionList
loop
[for each item in target.getItemInventory()]
add(item.getDropAction())
loop
[for each weapon in target.getWeaponInventory()]
add(weapon.getDropAction())
loop
[for each action in dropActions]
execute(target, map)
removeActor(target)
result : String
locationOf(actor)
location : Location
loop
[for each integer i from -1 to 1]
loop
[for each integer j from -1 to 1]
at(location.x() + i, location.y() + j)
newLocation : Location
opt
[newLocation.canActorEnter(actor)]
moveActor(actor, newLocation)
result : String
result : String

Req 5 UML Diagram:



Req 5 UML Sequence Diagram:



Design Rationale and Class Descriptions

Req 1: Environments

- The environments that are being created are the:
    - Puddle of Water
    - Gust of Wind
    - Graveyard
- Each environment inherits the ground abstract class as to follow DRY methodology
    - This is because all environment classes share common attributes and methods
- Moreover, the ground subclasses that can spawn enemies implement the EnemyGroundSpawn interface
    - This allows for greater understanding of the code and maintainability, as spawn logic can be encapsulated and identified easier
- The EnemySpawner class allows for the greatest level of spawn logic encapsulation, as classes intending to spawn enemies must pass in an enum with the available enemies to spawn.
    - Adheres to the Single responsibility principle, but does not follow the Open Closed Principle. This is a limitation of the Factory pattern like design chosen
        - This was acceptable as the level of modification required to extend functionality is very minimal
    - To further improve encapsulation, the spawn chances are retained within the enum of each enemy
- Determining location of object was given to the Utils class, as it seemed the most optimal for reducing code repetition whilst encapsulating the logic behind determining parity
    - Adding a greater amount of directions was not seen as a necessity and so the capacity for extending is reduced (returns only char at the moment). Could be improved

Changes: The Enemy Spawner is a new addition along with the EnemyName enum. Original method to determine map parity was far too complicated, having Utils store the map length was instead used.

<u>Class Responsibilities:</u>

Puddle of Water: Depending on parity, will spawn attempt to spawn Giant Crab or Giant Crayfish from EnemySpawner

Gust of Wind: Depending on parity, will spawn attempt to spawn Lonewolf or Giant Dog from EnemySpawner

Graveyard: Depending on parity, will spawn attempt to spawn Heavy Skeletal Swordsman or Skeletal Bandit from EnemySpawner

EnemySpawner: Spawns enemies based upon their percentage chance to spawn and whether an actor is already on the location to spawn. Encapsulates spawning logic.

Utils: Possess many useful utility methods, with the pertinent function being the isLocationLeftOrRight() method which encapsulates the logic of map parity

EnemyNames: An enum with the names of the enemies so that strings are not used, helps improve maintainability. Also stores the spawn rates of the enemies to ensure encapsulation

Req 1: Enemies

- Enemies will attack the player and all enemies except those with the same type enum (Skeleton, Canine and Crustacean). (In A1, this was implemented with interfaces)
    - This allows for easy extendibility, as a new type of enemy just requires a new enum to be added, for example, an undead type enemy would just need an UNDEAD enum, thus upholding the open-closed principle.
    - Pros: Holds with the Interface segregation principle, as an extra interface for each type would not be used for any extra functionality by the enemy classes. Avoids multiple level inheritance if abstract classes were used for each type.
    - Cons: May violate DRY in the future if extra functionality is added to a certain type of enemy.
- All enemies will have a 10% chance of despawning if not following the player, which will be done in the playTurn method using the performReset Method, which is implemented in the Enemy abstract class. (In A1, this was implemented with a DespawnAction)
    - This allows easy extendibility as each enemy will only have to extend the Enemy abstract class to receive the despawning chance, thus upholding the open-closed principle.
    - Pros: Follows the DRY principle, as the code relating to respawning the enemy will not be repeated and an extra class would not be created.
    - Cons: Does Not allow for the despawn chance to change across enemies.
- Each enemy class (HeavySkeletalSwordsman, LoneWolf, GiantCrab) will extend the Enemy abstract class, which in turn extends the Actor abstract class.
    - This allows for easy extendibility as enemies only need to extend the Enemy abstract class to receive the majority of their functionality, for example, a new Zombie enemy would just extend the Enemy class to receive the allowable actions and play turn functionalities, thus upholding the open-closed principle.
    - Pros: This upholds the DRY principle, as the repeated code of the actors and enemies will be replaced with abstractions.
    - Cons: There is an increase by one level of inheritance.
- As the heavy skeletal swordsman turns into a pile of bones when it is killed, the DeathAction class will check if the enemy being attacked is a Skeleton type enemy and if so, when the enemies health is 0 or less, it will create a PileOfBones class instance. (In A1 this was done in AttackAction instead of DeathAction)
    - This allows for easy extendibility as any new skeletal enemy, for example a skeleton dog will just have to have the Skeleton type enum for the pile of bones to be spawned at its death, thus upholding the open-closed principle.
    - Pros: easy to implement, upholds the DRY principle as the alternative of creating a separate class for a skeleton death action would include repeated code and would also be overapplying the SRP principle, as the class would not add any new functionality, also doing the spawning in attack action would not work as the enemy may be killed by another action.
    - Cons: May violate the SRP principle as the DeathAction class handles enemy, player and skeleton deaths.

<u>Class Responsibilities:</u>

HeavySkeletalSwordsman: Extends Actor abstract class and implements Enemy and Skeleton interfaces. Represents the Heavy Skeletal Swordsman hostile creature, selects a valid action to perform each turn, sets out allowable actions other actors can do to it.

GiantCrab: Extends enemy abstract class and implements Enemy and Crustacean interfaces. Represents the Giant Crab hostile creature, selects a valid action to perform each turn, sets out allowable actions other actors can do to it.

Enemy: abstract class which extends the Actor abstract class. Implements the playTurn and allowableActions methods, as they are common to all enemies, while initialising the behaviours hashmap. This greatly reduces repeated code in the individual enemy classes.

Canine: Interface representing Canines (LoneWolf and GiantDog)

Skeleton: Interface representing Skeletons (HeavySkeletalSwordsman and SkeletalBandit)

Crustacean: Interface representing Crustaceans (GiantCrab and GiantCrayfish)

EnemyAttackBehaviour: Implements behaviour interface, sets out the behaviour of enemies when within 1 block of another type of enemy or the player.

PileOfBones: Extends Ground abstract class. Is created when a skeleton is killed, spawns a new skeleton after 3 turns or is killed if hit once.

Req 1: Weapons

- The Grossmesser and Slam classes extend the abstract WeaponItem class (In A1, Slam was not a weapon, rather implemented only in AOEAction.)
    - The slam class allows for easy extendibility as any other enemies or actors using a slam weapon with an AOE attack can use it.
    - Pros: This upholds the DRY principle.
    - Cons: N/A
- The spinning attack of the Grossmesser and Slam will be implemented as a special skill using the AOEAttackAction class.
    - This allows for easy extendibility as any weapon with an AOE attack, for example a fireball spell weapon can use the AOEAttackAction class as its skill.
    - Pros: by separating the AOE attack from the normal AttackAction class, we uphold the Single Responsibility Principle, as the AOEAttackAction class only deals with area attacks.
    - Cons: It may violate DRY as there is some repeated code across the two classes.

<u>Class Responsibilities</u>

Grossmesser: Extends the WeaponItem abstract class and sets the super attack damage attribute to be 115 and its accuracy attribute to be 85%. Enables the use of the weapon class' methods and attributes.

Slam: Extends the WeaponItem abstract class, however its damage and accuracy are parameters. Enables the use of the weapon class' methods and attributes.

AOEAttackAction: Special attack ability which deals a set amount of damage to all actors within the surrounding environment of the attacking actor. Extends the Action abstract class.

Req 2: Enemies

- All actors that drop runes when killed implement the DropsRunes interface, which defines a default method that registers the number of runes to the RuneManager. (In A1, the interface wasnt used)
    - This allows for easy extendibility as any other rune dropping actors only need to implement the DropsRunes interface, thus upholding the open-closed principle.
    - Pros: Follows the Dependency Inversion principle, as enemies extend the DropsRunes interface rather than depending on the RuneManager, and follows the interface segregation principle, as the interface is small.
    - Cons: N/A
- The DeathAction class will use the transferRunes method of the RuneManager to transfer the DropsRunes actors runes to the player if the player killed the actor.
    - Pros: This method does not overapply the SRP principle as would be done if a TransferRuneAction class was used.
    - Cons: This may violate the SRP principle by adding extra functionality to the DeathAction class.

Class Responsibilities:

DropsRunes: Interface which allows actors to transfer runes to the player if the player kills them.

Req 2: Trader and Weapons

- Trader abstract class extends the Actor abstract class as to follow DRY methodology
- In order to greatly extend the selling and buying extendability, individually stored hashmaps are used
    - Each trader subclass can then have their own bought/sold weapons and individual prices associated, initialised in their constructor
    - This method was preferred as it allows for the greatest level of extendibility as storing prices as attributes for each weapon limits the potential prices to only 1 value
        - In future, traders in more dangerous areas may choose to sell and buy more at a more expensive price, or only sell and buy certain weapons, etc
    - The Open/Closed principle is adhered greatly in doing so
    - The only drawback of this design is that the registering different weapons as buyable and sellable may be a bit bloated, however this is necessary for future extendability
- Trader also deals with attempting to buy weapons from the player's inventory as opposed to the player, which may break encapsulation to a degree. However, this was not considered an issue given that all interactions are dependent on the abstract class Actor methods, and so the trader can rely on the engine abstraction which is not liable to change without massive adjustments to the entire system. Thus, a greater level of encapsulation is afforded

- The BuyAction and SellAction were made to be separate actions to adhere to the single responsibility principle such that the logic of each interaction is encapsulated
- To facilitate this, the RuneManager class is used. This class allows for the actor's runes to be altered in allowed methods
    - Encapsulates the Rune logic within Runes and the Rune Manager
    - Follows Single Responsibility principle

Changes: A RuneManager was added to facilitate rune transfer

<u>Class Responsibilities:</u>

Trader: Actor subclass is responsible for buying and selling with the player. They do not move. They have an infinite stock of weapons.

BuyAction: Responsible for allowing the player to buy from the Trader

SellAction: Responsible for allowing the player to sell to the Trader

RuneManager: Responsible for managing actors and their respective runes, and allows specified actions upon those runes

Req 3: Flask of Crimson Tears

- The Flask class implements a HealItem class, which is an abstract class that extends the item class.
    - This is an adjustment from the first design, which simply had the flask extend the item class. This new design improves the extendibility of the code.
- This HealItem class has a HealAction as an allowable action, which extends the Action class and heals the actor using the action inside its execute method.
- The HealItem class also tracks the uses of the HealItem, and allows the reset of its uses using a resetUses method.
    - By implementing the HealItem class in this way, it means that in the future if there are heal items added to the game similar to the Flask of Crimson Tears, they can easily extend the HealItem class. This abides by both DRY and OCP principles.
- The HealItem class also implements a HealConsumable interface, which includes the methods for the healamount, verb, uses and string of the HealItem.
    - In the future, this interface could be used to determine whether an item can be used as a heal consumable.
- The Flask class is added as an attribute of the Player class, and then a reference to that attribute is added to the player's inventory.
    - This allows for the flask to be reset without iterating through every item in the player's inventory and checking if it is an instance of the Flask class inside the player's reset method.
    - One con with this implementation however is that it creates an association between the Player and Flask classes, which would be avoided by only adding the flask to the player's inventory.

<u>Class Responsibilities</u>

Flask: Item which is able to be held by the player and inherits the HealItem abstract class.

HealItem: Abstract class which extends the item class and implements the HealConsumable interface. Allows for the use of a HealAction which heals a specified amount.

HealConsumable: Interface which includes the methods for healamount, verb, uses and string to be used by the HealItem class.

HealAction: Action class which heals the player’s hitpoints a set amount, and deducts 1 charge from HealItem used by the action. Inherits the abstract Action class.

Req 3: Site of Lost Grace

- SiteOfLostGrace class extends the ground class to uphold DRY principles, and passes the ‘U’ character to the super constructor so that it is chosen as the display character.
- The SiteOfLostGrace class overrides the allowableActions method within the Ground abstract class, to provide the player with the ability to rest at the site when next to its location. It also sets the name for a site of lost grace as an attribute.
    - In the future, this will allow for different sites of lost grace to have different names.
- The action stored in the SiteOfLostGrace class allowableActions list is the SiteOfLostGraceAction, as long as the type of actor passed to the allowableActions method is a Player.
- This is checked by adding a capability to the Player class to give that class the status of Player, which can be checked by seeing if the actor has that capability.
    - This avoids an unnecessary dependence on the Player class from the SiteOfLostGraceAction, and also means that the instanceOf method doesn’t need to be used.
- This SiteOfLostGraceAction class execute method gets an instance of the ResetManager class and calls the run method to reset the game. It also updates the last rested location of the player inside the PlayerManager to the location of the site of lost grace.
    - By calling the reset manager directly, it avoids having an unnecessary reset action.
    - By storing the last location of the player in the PlayerManager class, it delegates some of the responsibilities of the Player class helping to avoid it becoming a ‘God’ class, therefore abiding by SRP principles.
- There were no significant changes to this implementation compared to the initial design.

Class Responsibilities

SiteOfLostGrace: Ground class which instructs the visual representation of the SiteOfLostGrace on the map and places it as a location in the game. Inherits the Ground abstract class.

SiteOfLostGraceAction: Action class which resets the game and passes the location of the site to the ResetManager class to be saved for the player’s next respawn. Inherits the Action abstract class.

Req 3: Game Reset

- All objects that are need to be reset will implement the resettable interface so that the code is more maintainable and understandable
- This includes:
    - All enemy actors
    - The player
- The ResetManager is the class that handles the objects that require resetting, following the Single Responsibility Principle
    - By having each class designate their own method of resetting, it allows for ease of extensibility, and so also adheres to the Open/Closed Principle
    - Each actor must have the data required to reset themselves, and so the player stores their last rested SiteOfGrace in their attributes - further encapsulates logic
    - This includes the player, which stores the locations at which they last rested so that they can respawn
- The interface and ResetManager in conjunction allow for a decent degree of encapsulation of reset logic, though the drawback is that the logic does bleed into the playTurn method of actors.
    - The ResetManager also uses the singleton pattern to enforce the design chosen (single instance only)
- The ResetManager clears the list of Resettables after each reset, and so objects must re - register themselves after reset
    - This was preferred over having objects de-register themselves after reset as it is re-registering is the rarer occurrence
- The ResetManager run method will be called after SiteOfGraceAction or if the player dies (checked in DeathAction)
    - Reset process is encapsulated in doing so, as all implementation is hidden
    - This was preferred over having a ResetAction as this was deemed unnecessary given that reset implementation was already encapsulated enough - calling the resetManager is equivalent
- The player will have their health and flasks restored (handled within player own class from reset method)

Changes: ResetManager will now only purely call the reset method of all resettables, with no parameters. Player will handle resetting themselves now, and determining whether they need to respawn. PlayerManager was also created to help facilitate this.

<u>Class Responsibilities:</u>

ResetManager: Responsible for calling all other resettables' reset methods

Req 3: Runes

- Runes implements the Item abstract class, and will pass the '$' character to the super constructor, upholding DRY principles. It will also implement the resettables interface
- The chosen implementation involves having a singular rune class that has an internal "amount" as this allowed for improved design flow for many aspects of rune trading/acquiring
    - The drawback to this choice is that the method to pick up runes violates the Liskov Substitution Principle, as the runes have a dedicated method to be introduced to the actor's inventory not possible via the superclass PickUpAction

    - This was seen as acceptable, as runes being picked up should only ever occur via the player interacting with runes, and so redesigning the system would be unnecessary to improve this single area
        - The only future functionality would involve picking up runes not dropped upon death, which can be implemented with the current system
    - The current system is also believed to be easier to extend to further features should they be necessary
        - Less complex transaction system can allow for use of items that provide runes to a far easier degree without producing excess objects
- Reset logic for the runes dropped by the player are encapsulated in the runeManager, namely in the setDroppedFromPlayerDeathRunes method, which while long in name, serves to inform maintainers that this method should only ever be called upon death

Changes: Runes now require PickUpRuneAction to be picked up. Reset logic has now changed, with runeManager and PickUpRuneAction working in conjunction to reset the player's dropped runes only upon death.

Class Responsibilities

Runes: Runes class which extends the Item class and contains a value for a quantity of runes as an attribute.

RuneManager: A singleton class that maps actors to their runes, allowing for access their runes through defined methods

Req 4: Classes/Combat Archetypes

- For the implementation of the combat archetypes, each combat archetype is given its own class which extends the Archetype interface.
    - This differs from the first design, which used an Enum that would contain each archetype. It was thought that this previous implementation would cause the need for too many switch cases and clutter the overall implementation, therefore limiting extendibility.
- Each class for the archetypes implements an Archetype interface, which contains methods for returning the hitpoints and starting weapons of the class.
    - Doing this abides by OCP principles, as it allows for any new archetypes to easily be added and implement the methods from the interface.
    - It also means that if new functionality should be added to all archetypes in the future, it can easily be added to the Archetype interface, due the extendability of the code.
    - An interface was used here instead of an abstract class as there are currently no concrete implementations that could be used by the child classes, however this could be an extendability issue in the future if there is some shared functionality that all archetypes should have, where an abstract class would be more appropriate.
- The Player class then holds one of the combat archetypes as an attribute, resulting in a composition relationship.
    - By implementing things this way, it means that each of the archetype classes don't need to be extensions of the Player class, which would make these

    - classes bloated, and instead makes it much easier to alter the functionality of the archetype classes.
- The selection of the starting class is done using an ArchetypeMenu class, which is called by the Application class.
    - This prevents the Application class from becoming a 'God' class, by delegating the task of archetype selection to this other class. This abides by SRP principles.
- The ArchetypeMenu class works similarly to the showMenu method inside the Menu engine class, iterating through free characters and assigning them to each Archetype before reading user input.
    - This implementation was preferred to alternatives such as using a switch case since it means any new classes will only need to be added once, which is to the archetype list passed to the archetypeSelector static method of the ArchetypeMenu class.
    - The archetypeSelector method will require a list of type Archetype, meaning that any new archetypes that implement that Archetype interface can easily be added to the list, abiding by LSP principles and making the code easily extendable.

<u>Class Responsibilities</u>

Archetype: Interface to be implemented by the different combat archetypes, which requires the implementation of the hitpoints and starting weapons methods.

Samurai: Combat archetype class which implements the Archetype interface. Has 455 hitpoints and starts with the Uchigatana.

Bandit: Combat archetype class which implements the Archetype interface. Has 414 hitpoints and starts with the Great Knife.

Wretch: Combat archetype class which implements the Archetype interface. Has 414 hitpoints and starts with the Club.

ArchetypeMenu: Class to allow the user to select an Archetype by iterating through a list containing all archetypes.

Req 4: Weapons

- Each weapon extends the abstract class WeaponItem, allowing for the abstraction of many of the weapon and item capabilities that each weapon inherits.
    - This abides by DRY principles.
- Uchigatana and Great knife each have classes for their special abilities, UnsheatheAction and QuickstepAction.
    - The classes for these special abilities extend the Action class, once again abiding by DRY principles.
- Uchigatana and Great knife have dependencies on these special ability classes as they return them in the getSkill method, which is implemented from the Weapon interface.
    - This implementation allows for the special skills to be easily accessed when iterating through the possible actions an actor can do on any given turn, as it can simply iterate through the weapon inventory of the actor and check the getSkill method of each weapon.

    - This also means that for any new weapons added in the future with special skills, they can easily be created in the same way, resulting in extendable code.
- QuickstepAction has a dependency on the Location class, as it uses the CanActorEnter method to determine whether the wielder can move to a given location.
- It then has a dependency on the GameMap class as it uses the MoveActor method to move the wielder to a chosen surrounding location.
    - One flaw with this implementation is that it means that the QuickstepAction class needs to have dependencies on both the Location and GameMap class.
    - This would be improved if just the getMoveAction method inside the Location class could be used, however this will only return a MoveActorAction instead of moving the actor directly.
    - It would also not be possible to only use the MoveActor method inside the GameMap class, as it would then be possible for an actor to move to an illegal location using the QuickstepAction.
- There were no significant changes to this implementation compared to the initial design.

Class Responsibilities

Uchigatana: Extends the WeaponItem abstract class and sets the super attack damage attribute to be 115 and its accuracy attribute to be 80%. Enables the use of the weapon class' methods and attributes.

Great Knife: Similar to the Uchigatana, except its damage is set to 75 and its accuracy is set to 70%.

UnsheatheAction: Special ability used by the Uchigatana weapon which deals increased damage at a lower hit rate.

QuickstepAction: Special ability used by the Great Knife weapon which deals normal damage and moves the user away from the enemy to a nearby location where the user is able to be placed.

Req 5: Enemies

- Each enemy class (SkeletalBandit, GiantDog, GiantCrayfish) will implement the Enemy abstract class, similar to the enemies in req 1.
    - Pros: This upholds the DRY principle as no code will be repeated.
    - Cons: there is multilevel inheritance.
- The environments and actors that spawn enemies will spawn either west or east enemies depending on the location of the instance using the isLocationLeftOrRight method of Utils
    - This allows for easy extendibility as the isLocationLeftOrRight method can be used by any new spawner classes, thus upholding the open-closed principle
    - Pros: Easy to implement and does not require extra classes.
    - Cons: May violate the DRY principle as some code is repeated.
- Giant Dog and Giant Crayfish use the Slam weapon and AOEAttackAction similar to GiantCrab in req 1.
    - Pros: This upholds the open/closed principle and DRY principle as code isnt repeated for the weapon.
    - Cons: N/A

- The pile of bones for the Skeletal Bandit is implemented in the same way as for the Heavy Skeletal Swordsman in req 1.

<u>Class Responsibilities</u>

SkeletalBandit: Extends Actor abstract class and implements Enemy and Skeleton interface. Represents the Skeletal Bandit hostile creature, selects a valid action to perform each turn, sets out allowable actions other actors can do to it.

GiantDog: Extends enemy abstract class and implements Enemy and Canine interface. Represents the Giant Dog hostile creature, selects a valid action to perform each turn, sets out allowable actions other actors can do to it.

GiantCrayfish: Extends enemy abstract class and implements Crustacean interface. Represents the Giant Crayfish hostile creature, selects a valid action to perform each turn, sets out allowable actions other actors can do to it.

Req 5: Weapons

- The Scimitar class extends the abstract WeaponItem class
  - Pros: This upholds the DRY principle as the WeaponItem code is not repeated.
  - Cons: N/A
- The spinning attack of the Scimitar will be implemented as a skill using the AOEAttackAction class
  - Pros: Reusing the AOEAttackAction class upholds DRY.
  - Cons: N/A

<u>Class Responsibilities</u>

Scimitar: Extends the WeaponItem abstract class and sets the super attack damage attribute to be 118 and its accuracy attribute to be 88%. Enables the use of the weapon class' methods and attributes.