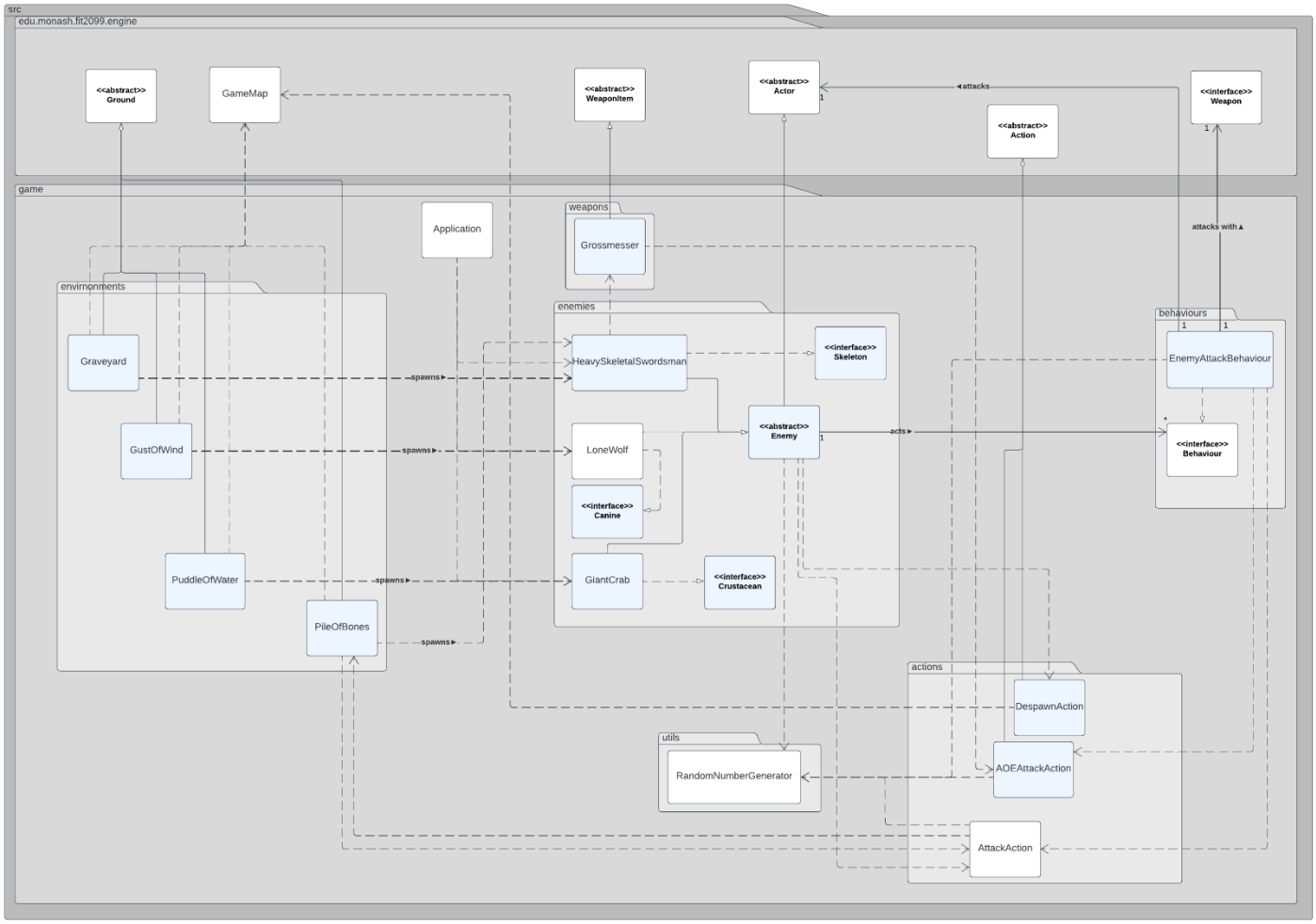
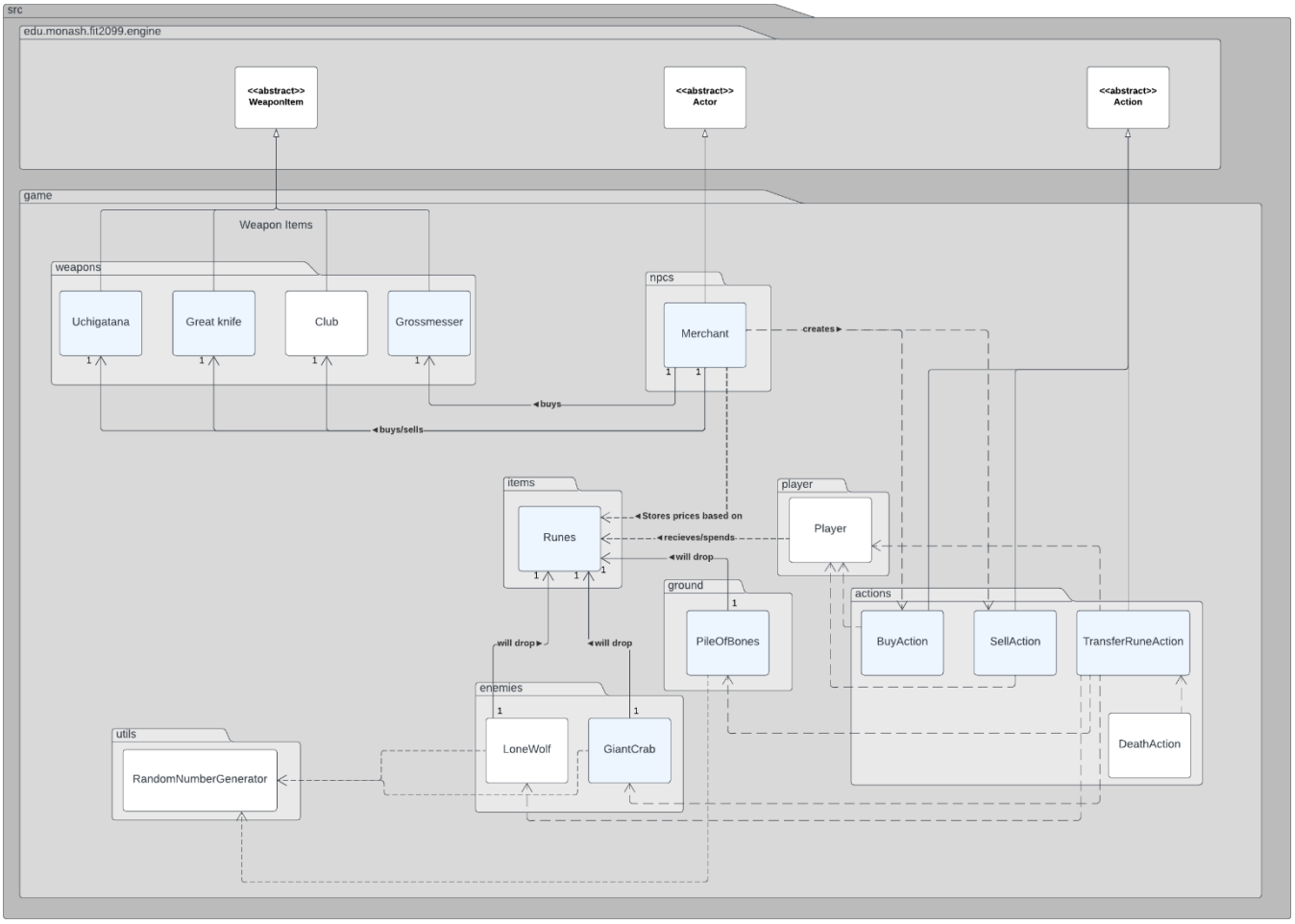
FIT2099 Assignment 1 Master Document

Applied Session 3, Group 8

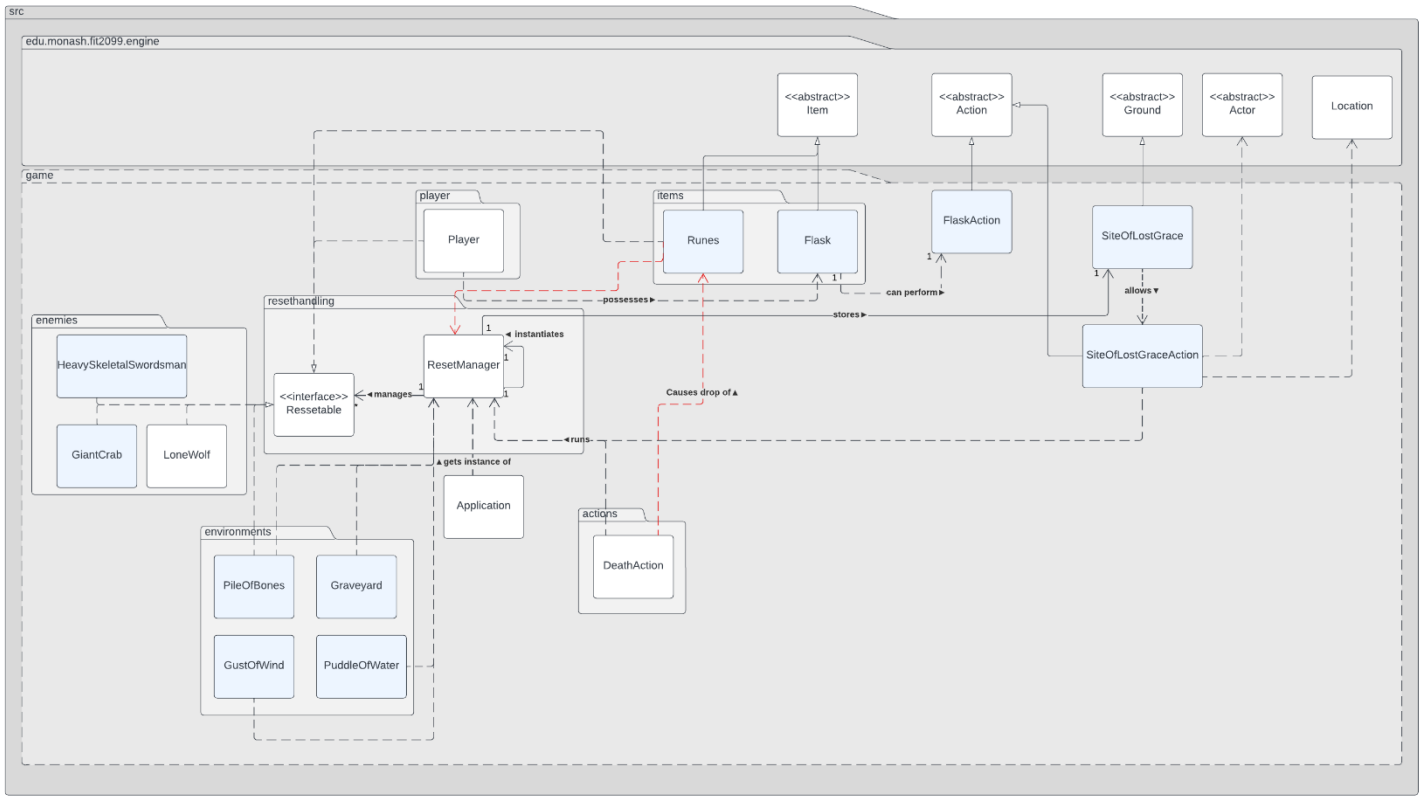
Req 1 UML Diagram:



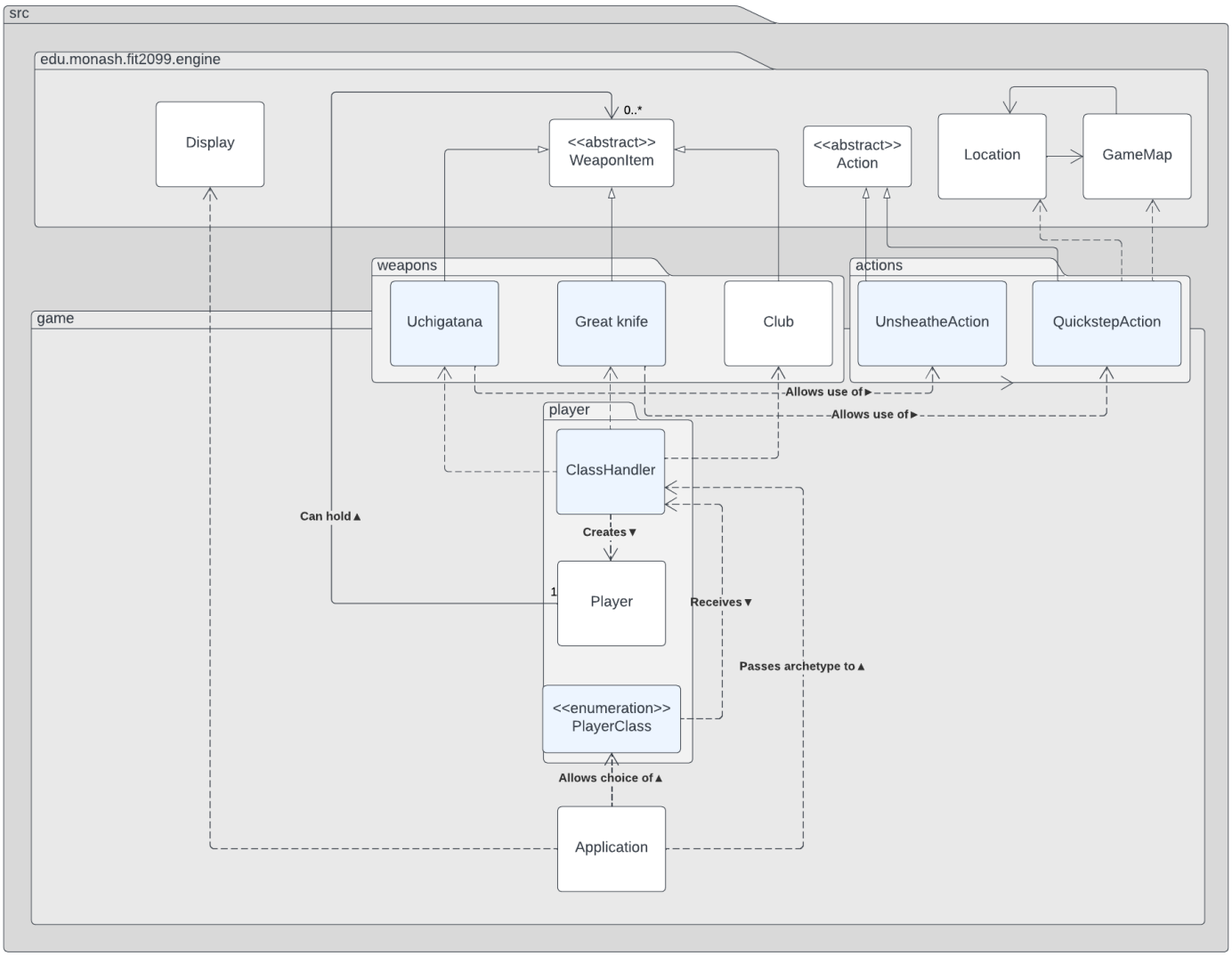
Req 2 UML Diagram:



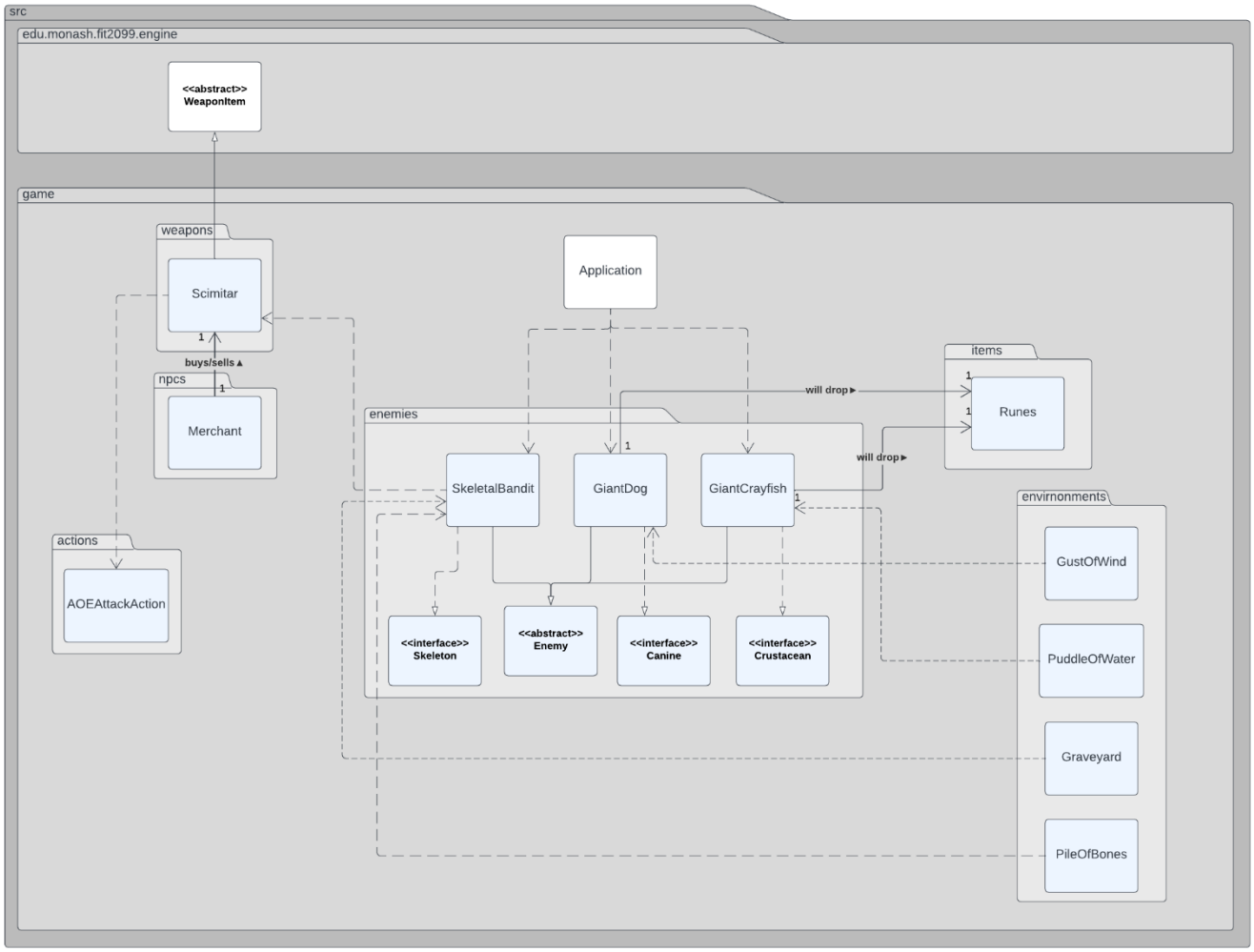
Req 3 UML Diagram:



Req 4 UML Diagram:



Req 5 UML 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 method.
- In the tick method of ground, the class should run the probability of spawning, then given a success, should:
    - Access the map getter for the location (the argument of tick method)
    - Use the GameMap to access the addActor method to add actor as required.
    - This ensures that the Single Responsibility Principle is maintained, and that code is in line with the engine design.
- This method of spawning has the benefit of remaining unaffected by changes within the game project, as all classes that are interacted with are within the engine, and so ensures a high degree of encapsulation.
    - Also increases the Single Responsibility of the class, as the spawn logic is separated within its own method.

<u>Class Responsibilities:</u>

Puddle of Water: Extension of Ground class. Uses the ASCII character “&”. Spawns Giant Crab at 2% chance each tick .

Gust of Wind: Extension of Ground class. Uses the ASCII character “~”. Spawns Lone Wolf at 33% chance each tick.

Graveyard: Extension of Ground class. Uses the ASCII character “n”. Spawns Heavy Skeletal Swordsman at 27% chance each tick.

Req 1: Enemies

- Enemies will attack the player and all enemies except those with the same type interface (Skeleton, Canine and Crustacean).
- All enemies will have a 10% chance of despawning if not following the player, which will be done in the playTurn method using the DespawnAction class, which is implemented in the Enemy abstract class to uphold the DRY principle.
- Each enemy class (HeavySkeletalSwordsman, LoneWolf, GiantCrab) will extend the Enemy abstract class, which in turn extends the Actor abstract class. This upholds the DRY principle, as the repeated code of the actors will be replaced with abstractions, the reduction in repeated code outweighs the fact that there is a slightly deep class extension.
- As the heavy skeletal swordsman turns into a pile of bones when it is killed, the attack action class will check if the enemy being attacked implements the skeleton interface and if so, when the enemies health is 0 or less, it will create a PileOfBones. This approach was chosen over creating a separate class for a skeleton attack action as that would be overapplying the SRP principle as the skeleton attack action class will be essentially the same, just with a single changed method.

- The slam attack of the GiantCrab is implemented in the AOEAttackAction class, which will apply the intrinsic weapon of the GiantCrab to all actors within the area. This upholds the single responsibility principle by separating the AOE attack from the normal AttackAction class.

Class Responsibilities:

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.

DespawnAction: Extends action abstract class. Removes actors by using GameMap removeActor method.

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.

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 class extends the abstract WeaponItem class, upholding the DRY principles.
- The spinning attack of the grossmesser will be implemented as a skill using the AOEAttackAction class.

Class Responsibilities

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.

Req 2: Enemies

- All enemies except skeletal enemies will have a Runes attribute added to their class where the number stored as the quantity of these Runes is generated using the RandomNumberGenerator.
- This item will not be dropped at the death of enemies
- We considered not having the enemies hold the Runes attribute and instead generate the Runes to be dropped upon death, however we felt it would be easier as the runes had a different range for each enemy.
- PileOfBones will also have a runes attribute.
- The DeathAction class will use the TransferRuneAction class to transfer the enemies'/pile of bones' runes to the player if the player killed the enemy/pile of bones.

Class Responsibilities

TransferRuneAction: Extends Action abstract class, transfers runes of killed enemy to player.

Req 2: Trader and Weapons

- Trader extends the Actor abstract class as to follow DRY methodology
    - This trader will possess a CanSell hashmap attribute containing weapons (keys) that can be sold and associated costs (items) of runes
    - They will also possess a CanBuy hashmap attribute containing weapons (keys) that can be bought and associated costs (items) of runes
    - This method allows for easy extension (adding new items to buy and sell) as per the Open - Closed principle
- BuyAction and SellAction will extend Action abstract class as to follow DRY methodology
- When an actor is in front of the trader, they will initiate all possible BuyAction and SellAction if the actor is the player
    - To find all valid SellActions, Trader will use the getItemInventory method of the player and compare what items they have vs what items can be bought
    - This ensures the Open - Closed Principle is maintained as it can easily be extended with no complications
- Thus, the trader must have an association with the weapons they can sell (Uchigatana, GreatKnife, Club), and can buy (same as sell weapons but GrossMesser is included)
- The trader will also have a dependency on the BuyAction and SellAction, as they construct the object but do not store it.
    - While this may go against dependency injection, the
- The BuyAction and SellAction will also have a dependency on the player as it must access and alter the amount of runes in the player's inventory and their weapon inventory
- A benefit to separating the buying and selling into 2 different classes is that it will allow flexibility and clarity, as buying or selling may follow a different process in the future independent of each other, such as bartering of some kind exclusively for buying. This also allows for more clarity in determining parts of the code that will represent selling and buying specifically
    - This follows the Single Responsibility Principle

Class Responsibilities:

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

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%.

Req 3: Flask of Crimson Tears

- Flask is an item added to players' inventory on the start of the game from within the player constructor. By implementing it this way, it ensures that only the Player class is dependent on the Flask class.
- Flask extends the item class.
- Flask has a FlaskAction class which extends the Action class.
- By using the extension of the abstract item and action classes, we uphold DRY principles.
- The flask action is added to the flask item's allowableActions ActionList.
- This allows for the FlaskAction to be added to an actor's allowable actions on each turn of the game, when the Flask item is iterated over.
- The Flask item has an integer attribute representing the amount of uses left.
- Each time the flask action is called from the flask item class, the integer loses 1 point.
- Once the integer is equal to 0, the flask action is removed from the flask item's allowable actions ActionList.
- This implementation was preferred over removing the item from the Players' inventory, as it removes the necessity to add the flask item back to the Players' inventory upon the game's reset.
- Since the flask item class inherits the Item class, we can then easily set the portable attribute set to false, so that it is not able to be dropped.

Class Responsibilities

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

FlaskAction: Action class which heals the player's hitpoints a set amount, and deducts 1 charge from the Flask item for each use. Inherits the abstract Action class.

Req 3: Site of Lost Grace

- SiteOfLostGrace class will extend the ground class, and pass the 'U' character to the super constructor so that it is chosen as the display character.

- The SiteOfLostGrace class will override 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.
- The action stored in the SiteOfLostGrace class allowableActions list will be the SiteOfLostGraceAction, as long as the type of actor passed to the allowableActions method is a Player.
- This will be 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 will use the ResetAction class to reset the game. It will also pass the location at which the player interacted with the site to the ResetAction class, which will use this Location to choose where to respawn the player.

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,
  This includes:
    - All enemy actors
    - The player
    - The PileOfBones
- All environments that spawn creatures should add said creature to resetables in ResetManager
- ResetManager will have a factory method and will only allow one instance of the class to exist at one time, but can also be accessed from anywhere
    - Allows for better encapsulation, as the means of constructing the class is hidden
- It will store the object for the last rested site of grace so that the player can respawn there
    - This does decrease the cohesiveness of the module, however it is necessary for the player to be returned to the correct spawn location. Storing the last rested Grace on the player (an alternate implementation) would reduce cohesiveness as well whilst also affecting the cohesiveness of the player and its module
- It will iterate over all stored resettable objects and call their reset methods
    - These will most likely result in each resettable creating a DespawnAction by some means
    - This preserves the Encapsulation of the resetting, as each class has their own reset method responsible for whatever action that resetting may do
    - In doing so, this also ensures the Open Closed Principle is maintained as you can easily add new functionality for resetting

- 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
    - However, the run method will require input as to whether the character has died in order to determine whether to delete the runes
    - This will reduce the cohesiveness, but is required to ensure correct functionality is produced
- The player will have their health and flasks restored (handled within player own class from reset method)

Class Responsibilities:

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 Rune item class will not be portable, until the DeathAction upon which the portability will be toggled, and the item will be dropped. It will then be added to resettable list in the ResetManager
- Inside the Runes class, there will be an attribute describing the quantity of runes the player currently holds.
- Once picked up, the runes will no longer be resettable
- This implementation was preferred over having the Runes class represent a single rune, and having a list of these runes, as we felt it would overcomplicate rune transactions and limit further extendability.

Class Responsibilities

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

Req 4: Classes/Combat Archetypes

- When the Application class is run, the player will be able to select a combat class from the types in the PlayerClass enumeration.
- The use of an enumeration here allows for the elimination of 'magic numbers' throughout the code, and also allows for new classes to be added easily in the future.
- One limitation of using an enumeration to cover all classes instead of making each combat class their own class is that it limits future extendability, as it would be difficult to add extra class based capabilities to the player.
- User selection is made before the world object is run in the application, which allows for the user to only have to select a class once at the start of the game, which will become useful after the game reset function is implemented.
- The type selected is then passed into a static method for creating a player in the ClassHandler class.
- The use of the static method here means that an instance of the ClassHandler class does not need to be created in the Application class.

- This method uses a switch case to create a player with the correct weapon and hitpoints by passing this information to the Player class' constructor.
- The use of the ClassHandler class here means that the Player class does not need to have a dependency of every single weapon individually, and can just be dependent on the abstract weapon class.
- It also means the switch case does not need to be placed in the Application class, preventing the creation of a 'god' class.

Class Responsibilities

PlayerClass: Enumeration to contain types of classes for the players' selection.

ClassHandler: Handler to pass the starting weapon and amount of hitpoints to the Player class' constructor based on the combat class selected.

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.
- Uchigatana and Great knife each have classes for their special abilities, UnsheatheAction and QuickstepAction. These classes extend the Action abstract class.
- Uchigatana and Great knife have dependencies on these special ability classes as they will return them in the getSkill method, which is implemented from the Weapon interface.
- This implementation allows for the Unsheathe and Quicksteps actions to be returned as allowable actions for actors holding the weapon on each turn of the game.
- 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 will need 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.

Class Responsibilities

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. This upholds the DRY principle.
- The Graveyard, GustOfWind and PuddleOfWater will spawn either west or east side enemies depending on the location of the class instance.
- The Application class will spawn either west or east side enemies depending on their location.
- Giant Dog and Giant Crayfish use the AOEAttackAction similar to GiantCrab in req 1.
- The pile of bones for the Skeletal Bandit is implemented in the same way as for the Heavy Skeletal Swordsman in req 1.

Class Responsibilities

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, upholding the DRY principles.
- The spinning attack of the Scimitar will be implemented as a skill using the AOEAttackAction class.

Class Responsibilities

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.