FIT2099 Object-oriented Design and Implementation

Assignment 1 - UML Diagrams & Design rationale

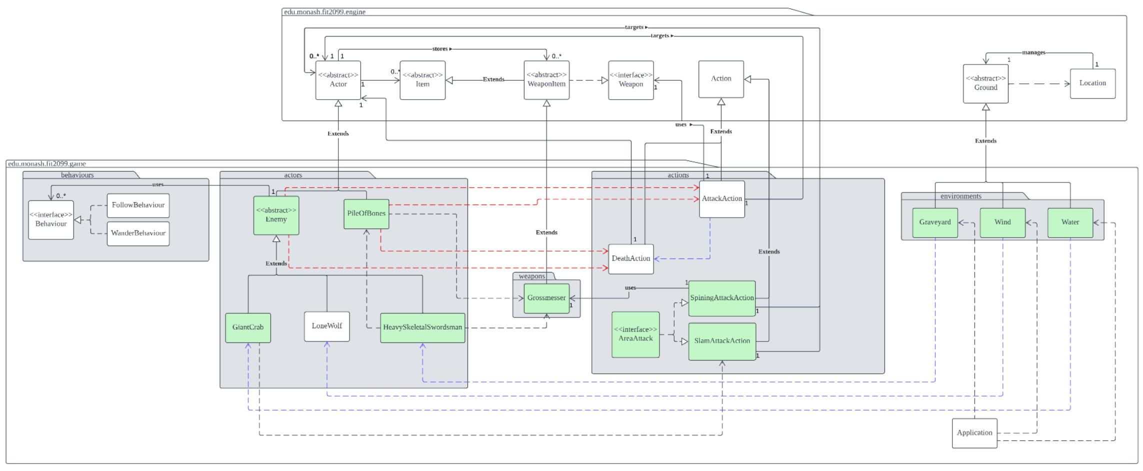
Group members:

1. (1) Muhammad Mustafa Khan 33041202
2. (2) Nurin Damia Ahmad Azhar 32903510
3. (3) Iliyana Samsudin 32738781

Link to Contribution log:
https://docs.google.com/spreadsheets/d/1e7polQC9IYCEiwOh9_jiUQ6UL1qxGAKn3CofYyGJVig/edit#gid=0

Requirement 1 - Environment & Enemies

UML Diagram



Design Rationale

The UML diagram above represents the object-oriented system of the Environment and Enemies that are available in the game.

Graveyard, **Wind**, **Water** are 3 concrete classes that are extended from the abstract class Ground. All of the 3 types of environment are extended from the Ground abstract class because they share common attributes and methods which a type of a Ground should have. However, each environment is able to have unique attributes and methods such as unique characters and different probability to spawn different types of enemies. Hence, creating 3 subclasses allows us to cater to each class uniqueness and reduce code repetition which follows the DRY principle (Don't Repeat Yourself).

Enemy class is an abstract class and is extended from the Actor abstract class which provides all the functionalities required by an Actor to create NPC (non-playable character). Then, **HeavySkeletalSwordsman**, **LoneWolf**, **GiantCrab** are concrete classes that are extended from the Enemy class. Although there is a multiple tier of inheritance, each of the enemies share common attributes and methods such as the WanderBehaviour, FollowBehaviour and despawning from the map which are already available in the Enemy class. By extending from the Enemy class, it allows us to reduce code repetition which follows the DRY principle (Don't Repeat Yourself). Any new additional attributes and behaviours can be implemented in its respective classes to cater each class uniqueness.

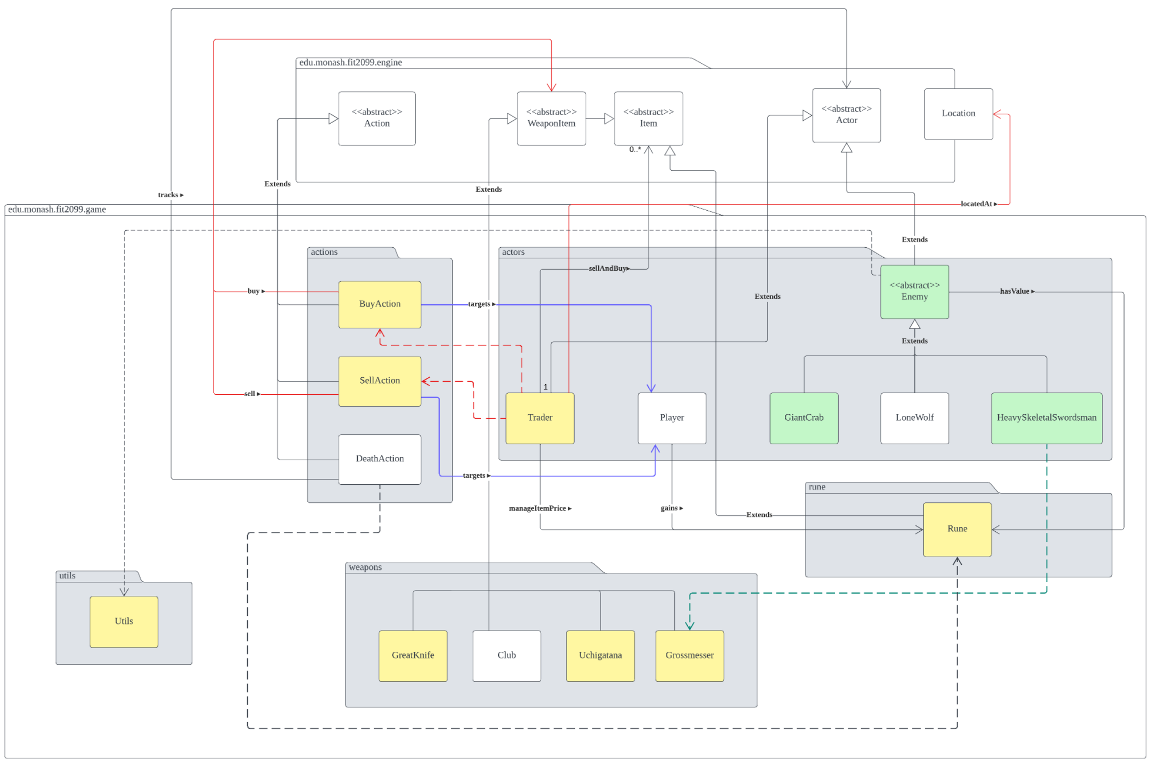
PileOfBones is a class that is extended from the Actor abstract class which provides all the functionalities required by an Actor. This class is responsible for the Heavy Skeletal Swordsman to turn into a pile of bones when it is attacked. The PileOfBones class will be created at the location where the Heavy Skeletal Swordsman was attacked and will be removed from the map if it is destroyed and drop the Grossmesser, else a new Heavy Skeletal Swordsman will spawn after 3 turns. By creating a PileOfBones class, it allows the Heavy Skeletal Swordsman class or other classes that require it by just instantiating it instead of implementing it in each class. Hence, this shows that the PileOfBones class is following the SRP (Single Responsibility Principle) which states that a class should have only one responsibility and it also helps in reducing code repetition. It also prevents the WET (Wasting Everyone's Time) principle.

Furthermore, a new status is created for the **LoneWolf** class to add as a capability in the constructor. This allows the lone wolf to attack other enemies but not its own kind.This also applies to the GiantCrab and HeavySkeletalSwordsman. However, for the **GiantCrab** class, an **SlamAttackAction** class is created and extended from the Action class which provides all the functionalities required by an Action. This class also implements the **AreaAttack** interface which allows the SlamAttackAction to cause damage to all creatures in its surroundings. By creating an AreaAttack interface, it allows different actions such as **SpinningAttackAction** that require the same property to implement the method from the interface which reduces code repetition.

Lastly, **Grossmesser** is a weapon that is used by the Heavy Skeletal Swordsman. It is extended from the WeaponItem class which extends from the Item abstract class and implements the Weapon interface. This allows new Weapon types to be created without having to modify the existing code. By extending the WeaponItem class, Grossmesser shares common attributes and methods that are implemented in the WeaponItem abstract class and the Item abstract class which can be reused. This allows us to follow the DRY (Don't Repeat Yourself) principle.

Requirement 2 - Trader & Runes

UML Diagram



Design Rationale

The UML diagram above represents the object-oriented system of the Trader and Runes classes.

The ***Trader*** class is a concrete class that acts as an NPC (non-playable character) that is only responsible for buying and selling weapons to Players, following the SRP (Single Responsibility Principle) which states that a class should have only one responsibility. The Trader class extends the Actor abstract class which provides all the functionalities required by an Actor, reducing code repetition following DRY principle.

For the buying and selling of Items/Weapons, we won't be using a list as it is assumed that merchants do not have inventories to store the items. Therefore, we will be using HashMaps, which maps Items to Rune objects, where Rune represents the currency used in the game. The implementation of HashMaps follows the OCP (Open/closed principle) which states that entities should be open for extension but closed for any modification. This is because the Trader class can be extended to include new items/weapons in the future without having to modify the existing code.

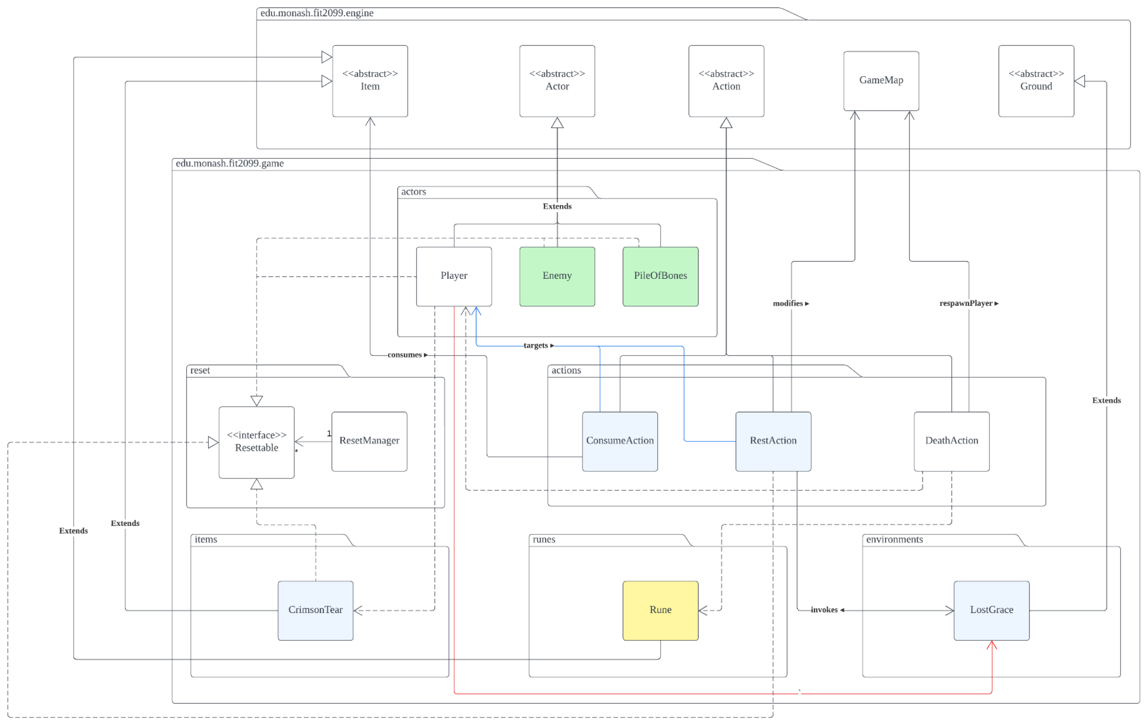
Furthermore, we created a new ***Rune*** class which extends from the Item abstract class which will be representing the currency being used by players and traders in the game of Elden Ring. The Rune class will be responsible for storing the Player's rune value. This follows the SRP by solely having one responsibility, making the class to have a clear purpose which makes it easy to understand and maintain.

Player, Enemy and Trader Classes are all associated with the Rune class because Players can obtain runes from enemy drops or by selling items, Enemies drop runes within a given range when killed, and Traders store the value of Items/Weapons in runes. The Enemy class will have three extra attributes, runeValue, max_rune and min_rune of type int. The max_rune and min_rune is to find the random value of rune between the given ranges of the enemy that will be generated by the random value generator in the Utils class. Then, the action of adding runes to the player's total rune value when an enemy is killed is done in the DeathAction class's execute method. Even so, the Rune class is independent from the other classes that use Runes, following the DIP (Dependency Inversion Principle). This means that the implementation of Rune class will not be affected by the other classes that use it.

Additionally, we have also created two subclasses of the Action abstract class, ***BuyAction*** class and ***SellAction*** class. The purpose of these two classes is for the interactions between traders and players for making transactions. The Player class doesn't have a direct dependency with the Trader class, instead the association between the Trader and Player classes are done indirectly through the BuyAction and SellAction classes. These two subclasses, ***BuyAction*** and ***SellAction*** adhere to the SRP principle because they only have a single responsibility of buying or selling items.

Requirement 3 - Grace & Game Reset

UML Diagram



Design Rationale

The UML diagram above represents the object-oriented system of the Grace and Game Reset classes.

The ***CrimsonTear*** class inherits from the Item class as it holds common attributes and methods with its own additional features. It also implements the Resettable interface which allows the reset of the number of uses when the game has been reset. Composition is being used to show its dependency from Player to CrimsonTear class, because each player will obtain two CrimsonTear objects at the start of the game. CrimsonTear cannot be dropped, which restricts the behaviour of the original Item parent class which can be dropped. So this implementation will break the LSP principle.

The ***ConsumeAction*** class follows the principle of encapsulation by extending the abstract Action class and encapsulating the logic for consuming consumable items because we assume that not all items will be consumable. Again, composition is being used as Player and Item classes will be an attribute of the ConsumeAction class, which promotes flexibility and modularity. Both CrimsonTear class and ConsumeAction class follows the SRP because they both have a single responsibility, one is used to represent the logic of CrimsonTear's behaviour and one for the logic of consuming consumable items.

Subsequently, the ***LostGrace*** class, which is a unique ground in the game, extends the abstract Ground class. Since only players can enter the LostGrace, we will be overriding the Ground class' canActorEnter method, where it returns true only if the actor is a Player instance. The LostGrace class follows the LSP principle, because it overrides the canActorEnter method with compatible behaviour which still sticks to the contract of the abstract class. Also, when the player dies, they will be respawned in the last sit of lost grace they have visited before, so there will be an association between the DeathAction class and the GameMap class, as well as Player class to LostGrace class to store the last visited LostGrace ground.

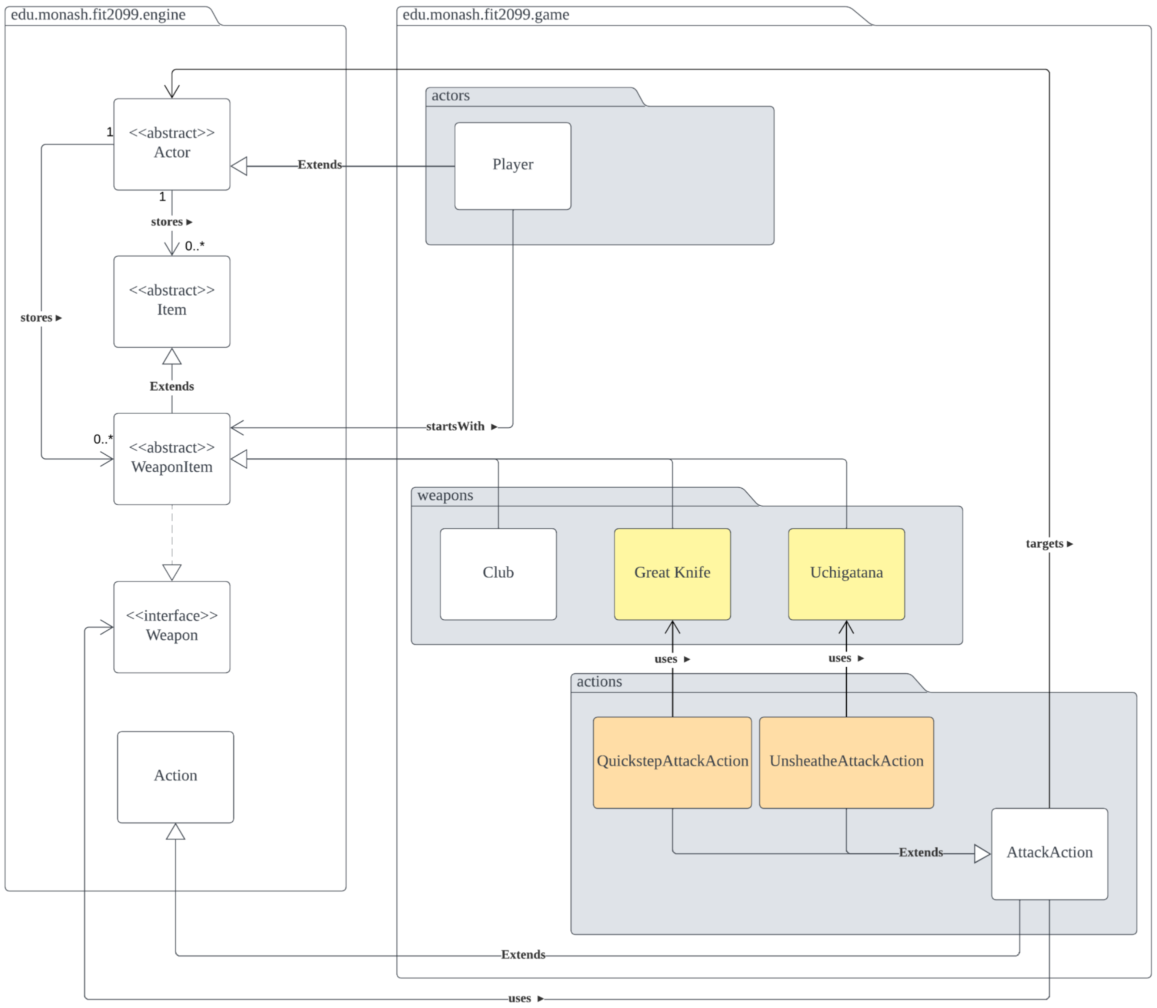
We will also be creating a ***RestAction*** class which extends the Action abstract class and implements the Resettable interface. RestAction can only be invoked when the player is in the LostGrace ground, therefore the RestAction class will have a LostGrace variable. The GameMap class will also be an attribute of the class since the GameMap will be modified during the reset, such as the despawning of enemies and the respawning of the player in the last site of lost grace they visited when the player dies. The RestAction class implements the Resettable interface which contains only one method, reset(), following the ISP principle because it only implements the methods required for its functionality, which is to reset the game.

The four classes, Player, Enemy, PileOfBones and CrimsonTear classes will have specified changes when the game is being reset, so they will be implementing the Resettable interface and have the rest() method being overridden to state their logic or behaviour when the game is being reset. This design follows the SRP principle as each of these classes will be responsible for resetting its own state, and this will allow extensibility and maintainability for further game versions in case other classes are resettable.

Regarding the runes, they will be dropped on the location just before the player dies. To do this, we will be storing the previous location of the player in the Player class as an attribute. If the player died again before getting the runes that they dropped previously, the runes will disappear. This will be handled in the DeathAction class by creating a new method for managing the runes on the map. This follows the OCP as the DeathAction class extends the behaviour of the Rune class, allowing extension of functionality without any modifications.

Requirement 4 - Classes (Combat & Archetypes)

UML Diagram



Design Rationale

The UML diagram above represents the object-oriented system of the combat and archetypes of players.

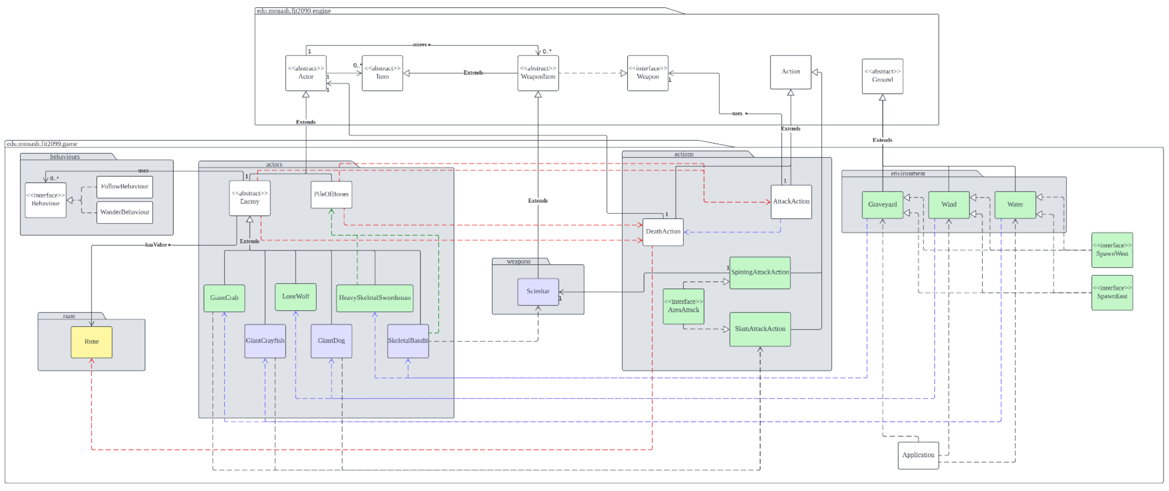
At the start of the game, the players will have to choose between the player classes they want to play, which are Samurai, Bandit and Wretch for the current version. The three classes differ from each other because they will have different hit points and they will also be given different starting weapons at the start of the game. Instead of using Inheritance to represent the different player classes, we will be using Composition to represent the different types of starting weapons for the player classes, so the Player Class will be modified to have a WeaponItem as a variable as the starting weapon. The WeaponItem variable will be initialised in the constructor based on the type of player class the user picks. For example, ***Samurai*** will be starting with ***Uchigatana***, ***Bandit*** with ***Great Knife*** and ***Wretch*** with ***Club****.* By using Composition, our code will be more flexible and extensible because we can just change the weapons or add new player classes with new weapons without having to modify the Player class. On top of that, we will be able to avoid Multiple Inheritance, which can become too complex and prone to getting a lot of errors.

All three starting weapons, ***Uchigatana, Great Knife*** and ***Club*** extend the WeaponItem class which extends the Item abstract class and implements the Weapon interface. This allows the creation of new Weapon types without having to modify the existing code. By extending the WeaponItem class, the three classes, ***Uchigatana***, ***Great Knife*** and ***Club***, the common attributes and methods that are implemented in the WeaponItem abstract class as well as the Item abstract class can be reused. This follows the DRY (Don't Repeat Yourself) principle. Any new additional attributes and behaviours can be implemented in the classes.

There are two unique skills, ***Unsheathe*** skill for ***Uchigatana*** users to deal 2x damage of the weapon with a hitRate of 60% to attack the enemy and ***Quickstep*** skill for ***Great Knife*** users that deals normal damage to the weapon to the enemy, and after attacking, the user moves away from the enemy to evade their attack. These two unique skills are created as two separate concrete subclasses of the AttackAction class, ***UnsheatheAttackAction*** and ***QuickstepAttackAction***. The execute method will be overridden to implement the behaviours of the unique skills which follows Polymorphism. Here, not only Inheritance is being used, but also OCP (Open-Closed Principle) as the AttackAtion class is open for extension and closed for any modification to the code, meaning that the behaviour of the class can be extended through the creation of new concrete subclasses without modifying the existing code. Furthermore, AttackAction is kept as non-abstract because the users that use Uchigatana or Great Knife weapons can still use the parent AttackAction class for a basic attack, and also their respective unique skills that are subclasses of the AttackAction class.

Requirement 5 - More Enemies (HD Requirement)

UML Diagram



Design Rationale

From our design in requirement 1, we are able to create new enemies as we have designed it following the DRY principle by creating an abstract Enemy class. Hence, 3 new concrete classes are extended from the **Enemy class** which are the **Skeletal Bandit**, **GiantDog** and **GiantCrayfish**. The implementation for the 3 new enemies are similar to the enemies that we have created in requirement 1. Therefore, no new classes are required to be created other than the Scimitar class which is the weapon used by the Skeletal Bandit. In addition, the DeathAction will have a dependency with the Rune class as the enemies are able to drop runes within a specific range if it is killed.

The SkeletalBandit class should be implemented similarly to the HeavySkeletalSwordsman as they are the same type. The only difference is that instead of the Grossmesser weapon, the skeletal bandit will have a Scimitar. **Scimitar** class is extended from the WeaponItem abstract class which extends from the Item abstract class and implements the Weapon interface. This implementation allows the Scimitar to have the ability as any weapon including the ability to be bought from and sold to the Trader class which is similarly implemented in requirement 2. This shows that we are following the DRY principle.

Although the GiantDog and LoneWolf are the same type, the implementation of the GiantDog would be slightly different as it has slam attack (single target) as an intrinsic weapon instead of bite and the GiantDog will have a dependency to the **SlamAttackAction** that implements **AreaAttack** interface which allows the GiantDog to cause damage to all creatures in its surroundings as a special skill. This is similar to the implementation of GiantCrayfish which has the same type as GiantCrab.

Lastly, we have created two interfaces which are **SpawnWest** and **SpawnEast** for the **Graveyard**, **Wind** and **Water** classes to implement. These interfaces allow the different Ground subclasses to spawn different enemies based on their location (East or West). Hence, by creating the two interfaces we have followed the ISP principle (Interface Segregation Principle) where the subclasses are not forced to depend on interfaces that they do not use and the DRY principle as it is needed for different types of ground. Since the Graveyard, Wind and Water classes are able to spawn different enemies for east and west, any new type of ground will have the flexibility to implement any of the interfaces or not at all.