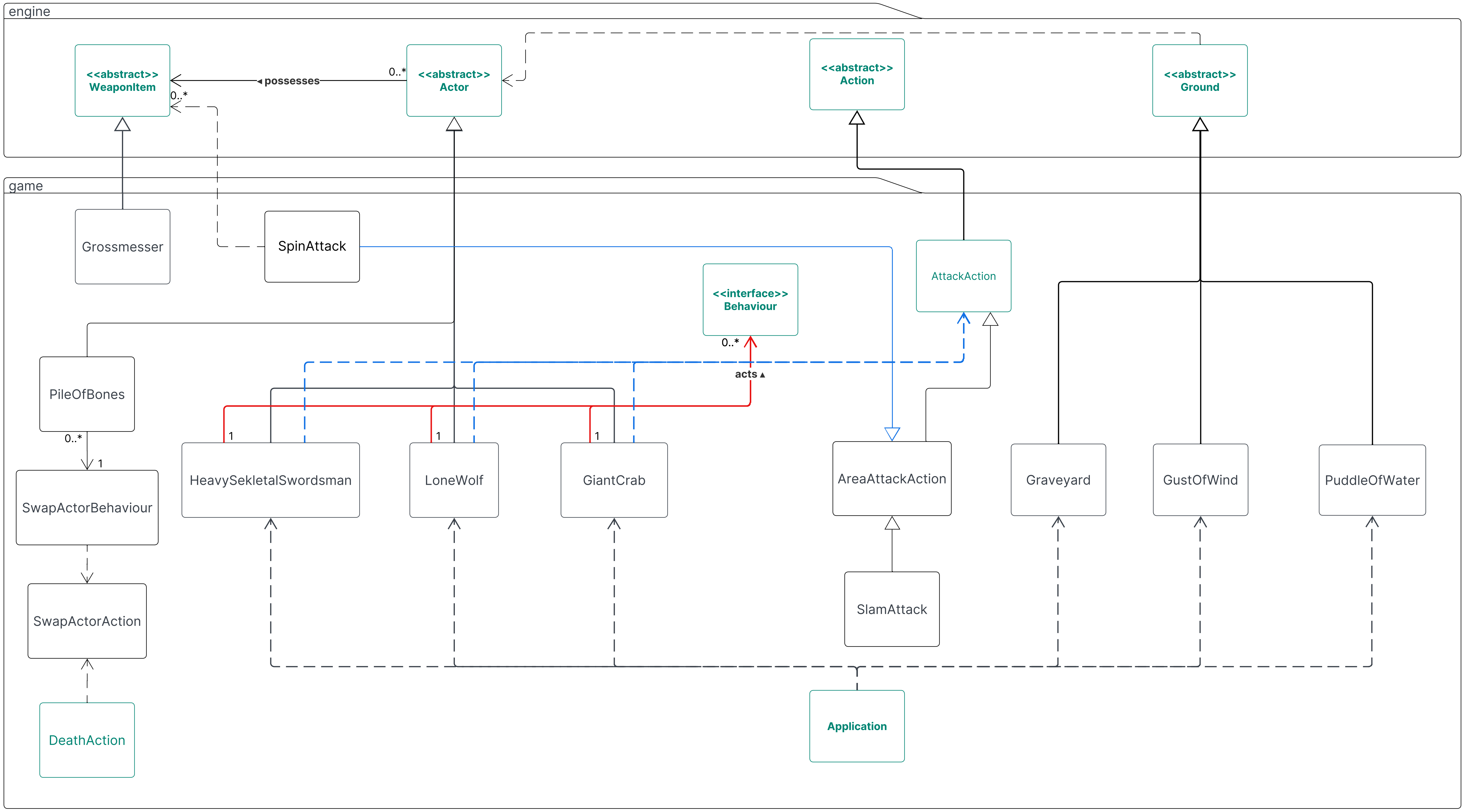
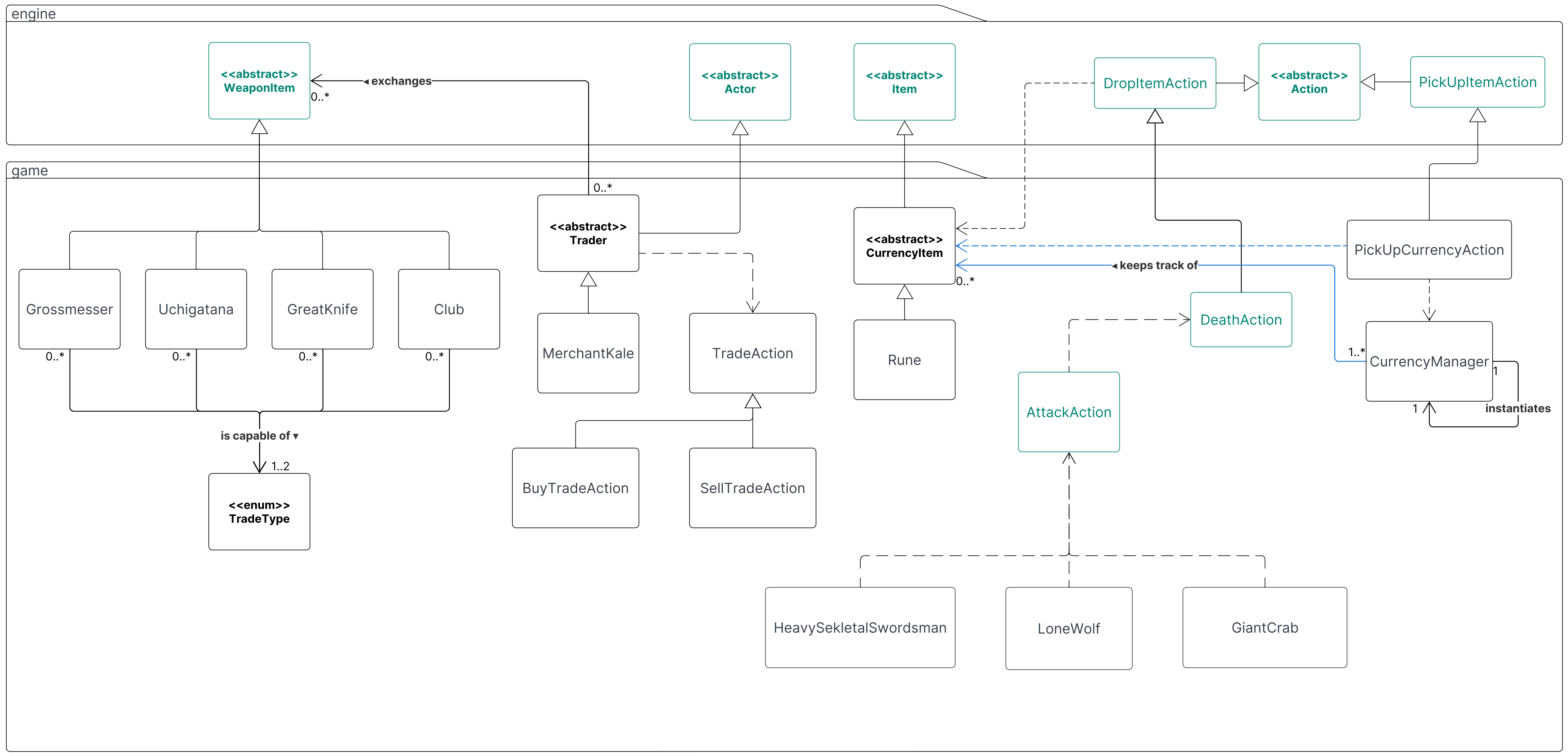
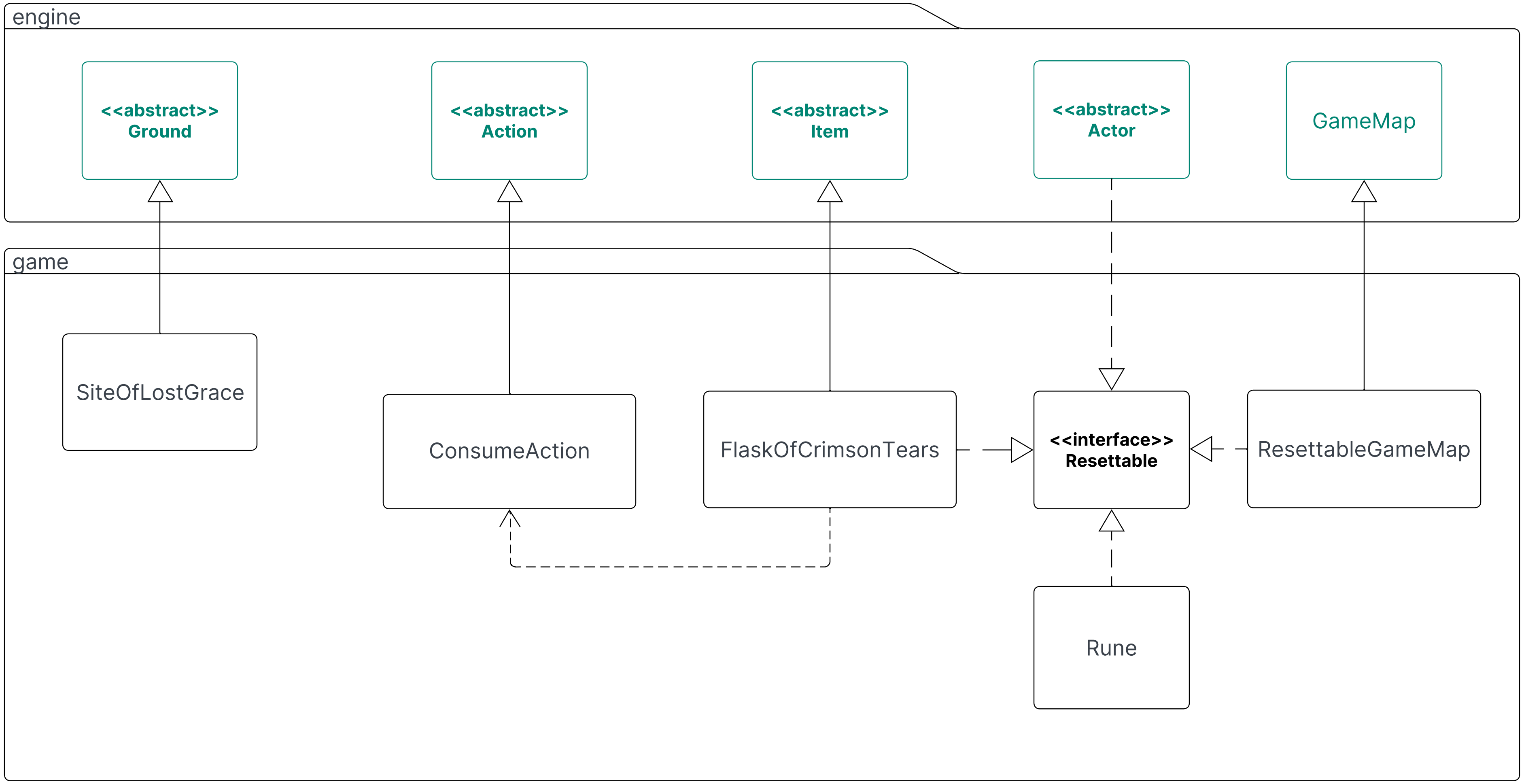


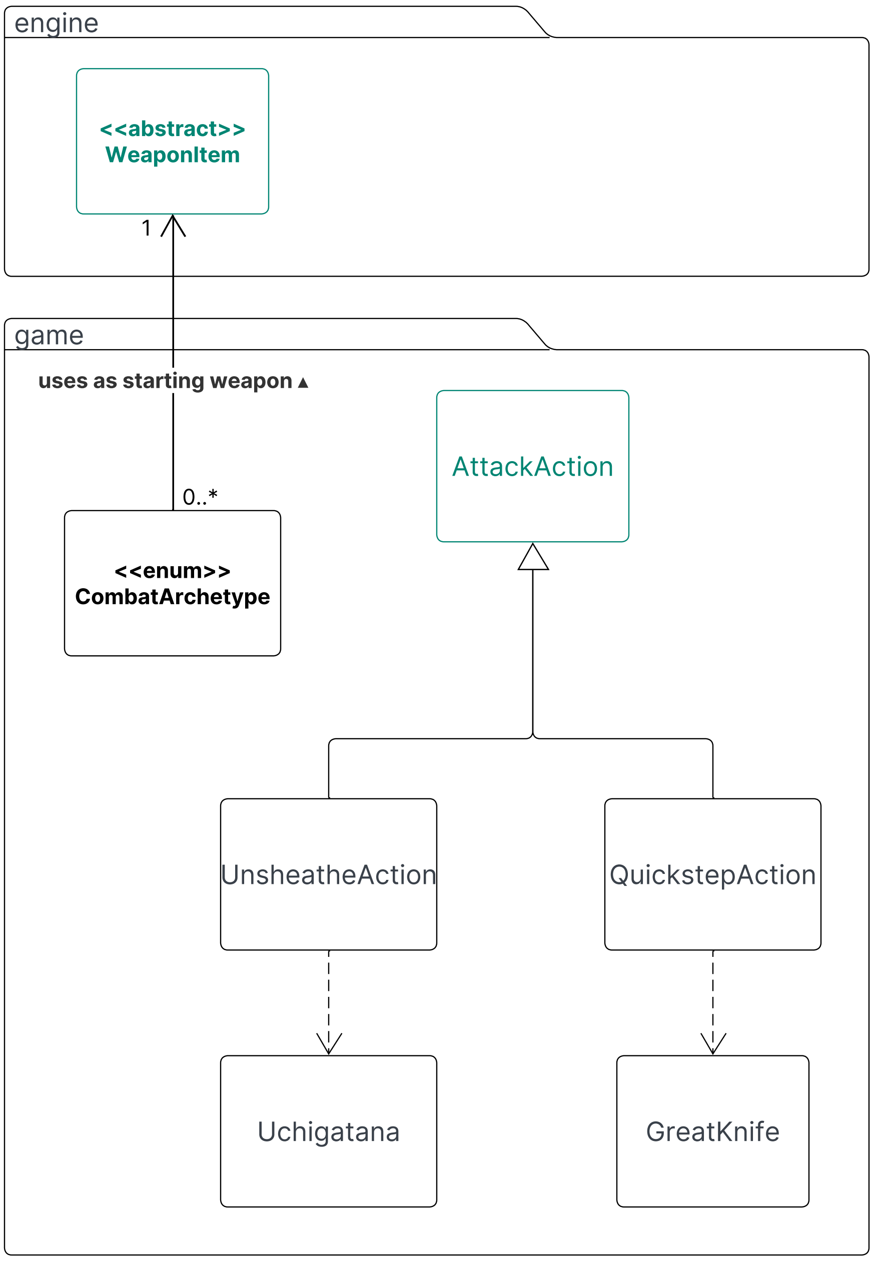
FIT2099 - Assignment 1
Applied 1, Group 3
David Kong, Xiaowen Zhou, Aung Kyaw



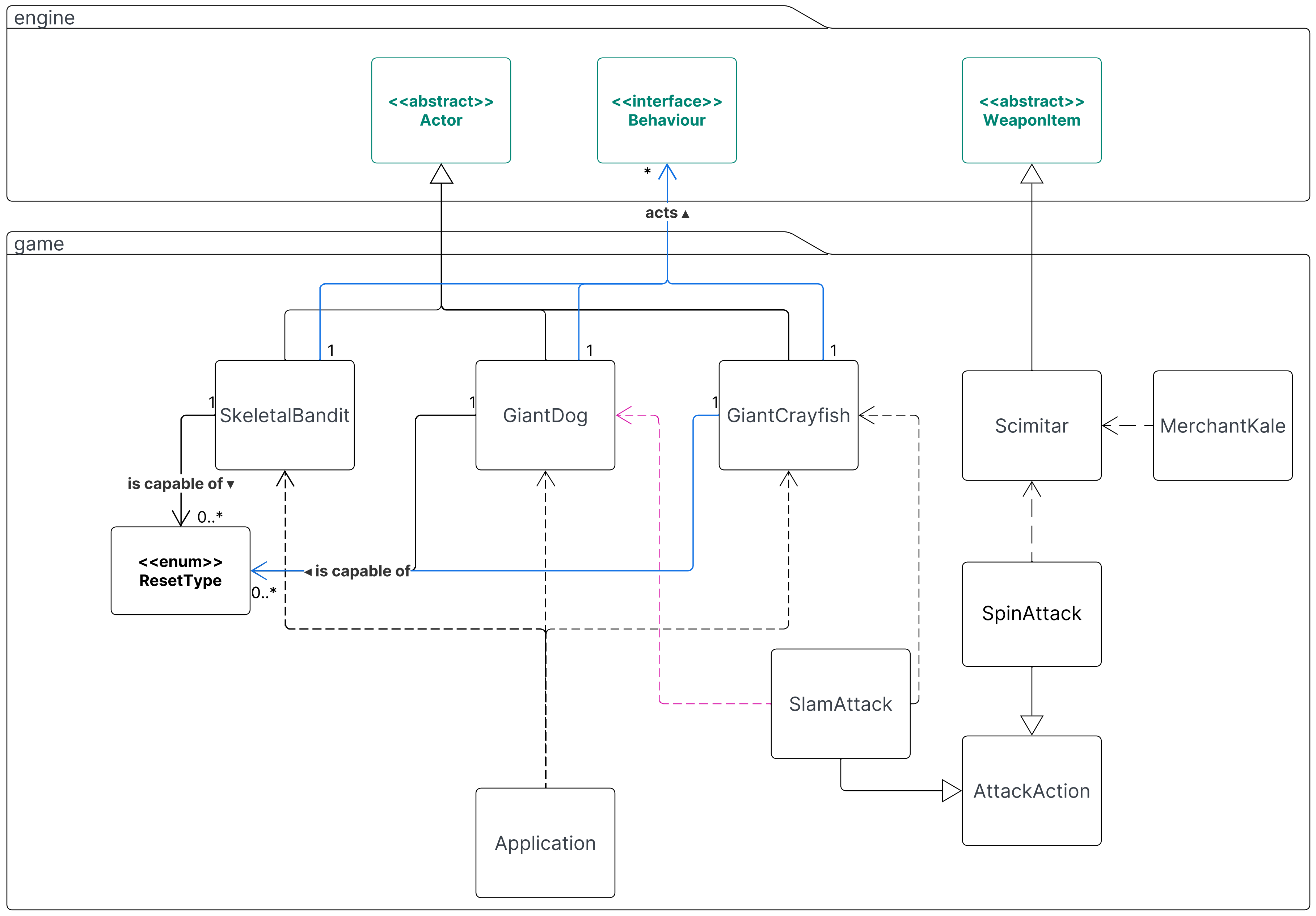
FIT2099 - Assignment 1
Applied 1, Group 3
David Kong, Xiaowen Zhou, Aung Kyaw



FIT2099 - Assignment 1
Applied 1, Group 3
David Kong, Xiaowen Zhou, Aung Kyaw



FIT2099 - Assignment 1
Applied 1, Group 3
David Kong, Xiaowen Zhou, Aung Kyaw



FIT2099 - Assignment 1
Applied 1, Group 3
David Kong, Xiaowen Zhou, Aung Kyaw

Req 1

Environments

The newly added environments extend the `Ground` abstract class, because they share common attributes (such as name, display character) and common methods (such as `playTurn`). Doing so reduces the amount of repetition (DRY), which increases maintainability.

This design also allows other `Ground`s to be added to the game in the future without modifying any other classes that may have `Ground`s (OCP), because they would all be treated as the same type of object.

Having the different environments extend `Ground` also allows them to be interchanged on the game map if needed since they can behave as the same object (LSP), which would be useful, say for example, if parts of the `GameMap` need to be switched out for another type of `Ground` in the game logic.

Enemies

To reduce repetition, all the enemies added to the game extend the abstract `Actor` class, as they all share common attributes such as hit points, turn-based methods, capabilities and more. These enemies also have an association with the existing `Behaviour` interface as they all exhibit similar behaviours, such as `WanderBehaviour`, which avoids repeating similar behaviours (DRY). This will allow other enemies to be added in the future, while being used in the same way as existing ones (LSP).

In case of the enemies not being able to attack the same type, an enum `EnemyType` is introduced. For example, the actor `LoneWolf` will have an enum `EnemyType.CANINE` assigned to it and when it is in the vicinity of another actor, `AttackAction` will check the enum of the target. If it is the same, then `AttackAction` will not execute. The use of enum will also ensure that the enemies added in the future will also not be able to attack actors part of the same enemy type. It also allows enemies to be part of more than one type, if desired in the future.

Another class `PileOfBones` will also be added that extends the `Actor` class as it uses common methods with other `Actor`s, such as `PlayTurn`, to avoid repeating similar code (DRY).

To implement the conversion to a Pile of Bones by the Heavy Skeletal Swordsman upon death, a new behaviour `SwapActorBehaviour` is created (which implements the `Behaviour` interface). `SwapActorBehaviour` implements the `Behaviour` interface, because it conditionally executes another `Action` – `SwapActorAction`. `SwapActorAction` extends the `Action` class so that it can be an executable action as part of the game flow. While this behaviour is only used in the skeletal-type enemies and the Pile of Bones, it is separated into another class so that the enemy class can remain the function of managing the enemy (SRP). It also allows other enemies or actors to utilise this behaviour in future versions of the game without modifying the actor-swapping code (OCP).

Area attacks of enemies, `SpinAttack` and `SlamAttack` will extend a new class `AreaAttackAction`, which extends the `Action` base class, as both attacks essentially behave in the same way (finding the 8 surrounding locations). This enables other area attack type actions to be added later in the development (OCP) without changing the area-type attack logic.

Weapons

The `Grossmesser` is a weapon that extends the `WeaponItem` abstract class. `Grossmesser` is a subclass of the `Weaponitem` abstract class so that it can be used in place of any other weapons for the actors who use it (LSP). The weapon `Grossmesser` has a unique skill `SpinAttack`. Implementing the Spin Attack as an Action allows it to work with the game engine, utilising the `getSkill` function as part of the `Weapon` interface.

Req 2

Since this section introduces the concept of currency and the currency 'Runes', a system to manage and interact with the currency should be made.

An abstract `CurrencyItem` class that extends the `Item` class is introduced to allow an extensible implementation of currencies in the game, should it be desired in the future. Since it behaves similarly to `Item`s in terms of how entities interact with it, it should extend an `Item` so that it can be stored in entities' inventory, be picked up, and be dropped.

To keep track of how much of a currency / the currencies entities have, a singleton class `CurrencyManager` is implemented. It is a singleton, because it is something that many other classes need to interact with (e.g. when trading items with traders, picking up currency off the ground). This way, only one instance needs to be created, and the records can be therefore accessible by all classes that need to manage currency. Inside the `CurrencyManager`, a data structure can be used to keep track of different currencies (if the game desires) across different entities (again, if the game desires). This enables extension of the use of other currencies and having more than one entity (other than the player) hold currency (OCP).

Beside being able to extend the game with entities other than the player holding currency, the `CurrencyManager` also follows Single-Responsibility Principle, by delegating the task of keeping track of currency/money to another class, and not done by the `Player` class, allowing the `Player` class to only handle strictly player-related functionality.

To implement the 'Runes' currency specifically, a `Rune` class should extend the `CurrencyItem` class so that it can be treated like any other currency in the game (if it exists) (LSP).

Because `CurrencyItem` extends the `Item` class, its drop action can be handled by the built-in `DropItemAction`, which would deposit the `Rune` object held in the enemy's inventory onto the ground. The pick-up action should be handled by a `PickUpCurrencyAction` class that extends `PickUpAction`. Since entities (the player, for now) that pick up currency items should be recorded by the `CurrencyManager` class, its pick-up action must be different to the default `PickUpItemAction` (which just adds it to the `Actor`'s inventory). The `PickUpCurrencyAction` would link to the `CurrencyManager` to update the record of who picked up the currency and how much was picked up. Again, this would allow the possibility for more than one player to exist in future versions of the game to pick up items (OCP).

Initially, the `PickUpCurrencyAction` class was going to extend the `PickUpItemAction` class, but it was discovered that doing so could not bypass `PickUpItemAction`’s method of adding the item to the inventory, which is not desired for a currency item. Therefore, it extends `PickUpAction` to reuse the implementation of removing the item from the ground once picked up (DRY).

Enemies

To allow enemies to drop Runes upon defeat, the `Rune` object can be added to the enemies' inventory so that it can be dropped upon defeat. To ensure the Runes only get dropped if defeated by a player, the `DeathAction` can check to see whether the `attacker` has `Status.HOSTILE_TO_ENEMY` as a Capability. This would also allow other players to exist in the game and receive dropped Runes if they defeat an enemy. This would also stop Runes being dropped if an enemy is defeated by an enemy.

Trader

Although there is currently only Merchant Kale in the game, a more flexible implementation of traders in general should be established to allow for extensibility.

A `Trader` abstract class extends the `Actor` class to provide an abstract implementation of what a Trader in the game can do. It could provide trading-specific methods, and allow for other traders to exist in the future (OCP), since objects that store Traders (like Grounds) will treat them in the same way (LSP).

To allow the player (and maybe other objects in the future) to make trades with traders, new classes extending the `Action` base class are made. A base class `TradeAction` is created to store and manage common methods and attributes used across all trading-related actions (like the trader, customer, item, price, etc.). The buy and sell actions will be implemented in classes `BuyTradeAction` and `SellTradeAction` that extend the `TradeAction` class, because they all use the same attributes.

A class for `MerchantKale` can extend the abstract `Trader` class to inherit all of the trader's functionality. The object-specific items would be the items that Kale has to trade.

Weapons

To determine which weapons are buyable and sellable by the player(s), a Capability will be given to the items' class definition. A new Enum class `Trade` will contain `Trade.BUYABLE` and `Trade.SELLABLE` that will be added to the items' `CapabilitySet`. When attempting to trade, the trade function in `Trader` will check that the item contains the appropriate `Trade` Capability before proceeding to process the trade transaction. By assigning capabilities for determining tradability, new items can be added in the future to the game without modifying any existing trading code (OCP).

Req 3

Flask of Crimson Tears & Site of Lost Grace

A new class for the Flask of Crimson Tears shall be made. The `FlaskOfCrimsonTears` class should extend the `Item` class, because it is an item that is held by the player in their inventory. This allows the Flask to be treated like any other item in their inventory list (LSP). To account for the health restoration action when the player uses/consumes the Flask, a new class `ConsumeAction` shall be made that extends the `Action` class. This allows the action of consuming the Flask and restoring the player's health to be delegated to another class solely responsible for the action (SRP). At the same time, if other items in the future happen to also restore the health of the player, they can also use the `ConsumeAction` class, to avoid repeating the implementation (DRY) for several items. The `FlaskOfCrimsonTears` class also needs to implement the `Resettable` interface, because its use count has to be changed when the game is reset. It needs to implement its own version of `reset` in `Resettable`, because its reset action is different to other classes.

Game reset

A new class `ResettableGameMap` implementing the `Resettable` interface is added to be able to clear items from the map upon game reset.

A new enum class `ResetType` is added as a Capability to the classes that get reset, though not ones that run the reset logic themselves. Items that get reset include items dropped on the ground, as their existence is defined at a `Location`, so the GameMap would be the class removing and “resetting” these items. Classes that run their own reset logic, like `GameMap`, `Player`, `FlaskOfCrimsonTears`, will be registered with the `ResetManager` so their reset methods can be called.

By defining two types of reset, classes can behave differently depending on the type of reset, such as items that might not be removed upon a certain type of reset. This is useful for extending the game in the future, where more reset behaviours might want to be defined.

Runes

To have Runes disappear upon the player’s repeated death, the `Rune` object can hold the capability of `Reset.RESET_ON_PLAYER_DEATH` to be wiped from the map upon repeated death. The `GameMap` `reset` shall be called after the `Player` class’s `reset` so that the Runes remain on the ground.

Req 4

Combat Archetypes

The Combat Archetypes should use an enum class `CombatArchetype` that defines the starting hit points and starting weapon. By creating a new enum class to hold these types, it allows the implementation to treat these items in the same way so it can, say, be presented to the player in the same way (LSP). At the same time, it also allows easy extension of more Combat Archetypes in the future, since they can just be added to the `CombatArchetype` class, and the rest of the program can interpret it as a new option that can be chosen and used by the player without modifying other classes (OCP).

An earlier consideration was to make an abstract class `CombatArchetype`, and make the available options child classes. However, this would violate OCP, as the new classes would need to be added to the data structure containing these options in other parts of the implementation when extending the game.

Weapons

As these are new weapons, new classes that extend the `WeaponItem` class will be made. Extending the `WeaponItem` class allows the weapons to be stored and used like other weapons in the game and by the player, as they share attributes and methods (LSP). The special 'skills' available in the Uchigatana and Great Knife can be implemented using new classes `UnsheatheAttackAction` and `QuickstepAttackAction` that extend the `AttackAction` class. This is done because they are in itself `AttackAction`s, just with modified attack characteristics and, for 'Quickstep', an additional action. So, reusing the `AttackAction` class allows for less repeated code (DRY), but also so that these actions can be returned by the `getSkill()` method for `WeaponItem`s. Making these actions their own classes means other weapons can possess this skill in the future if needed, which, again, reduces repeated code (DRY) and follows OCP, as the attack does not need to accommodate new weapons using its action. Defining the actions in new classes also ensures that SRP is followed, by delegating the attack implementation to another attack-action-specific class (which also follows the game engine).

Req 5

Enemies

New enemies `SkeletalBandit` of `SKELETAL` type, `GiantDog` of type `CANINE` and `GiantCrayfish` of type `CRUSTACEAN` are added using the same method in requirement 1. These enemies have the same behaviour as the enemies added in requirement 1.

East and West sides of the maps are calculated in the ‘GameMap’ when generated by using the ‘tick’ to get the x coordinate of the ‘Ground’ and determining if it is on the left side or right side. A downside of this approach is that the method for determining whether a `Ground` is in the east or west side of the map has to be run every time a related action is called. Although not computationally expensive (would be $O(1)$), it is still a possibly unnecessary computation. It also means that this check would need to be added to any action that has east- or west-side-specific methods, which is not ideal for maintainability.

An alternative method for determining east and west would be to add 'EAST' and 'WEST' enums as a Capability to the `Ground` upon instantiation. Adding the enums would allow the addition of other capabilities to the east and west sides but as of now the way to implement these enums is still being studied.

`GiantDog` and `LoneWolf` possess the same enum type `CANINE`, `SkeletalBandit` and `HeavySkeletalSwordsman` are of type `SKELETAL`, and `GiantCrab` and `GiantCrayfish` are of type `CRUSTACEAN`. As outlined in requirement 1, enemies are ensured not to be able to attack actors of the same type, with an exception of when performing `AreaAttackAction`.

Weapons

A new weapon `Scimitar` class will extend the `WeaponItem` class, which is similar to the previous `Grossmesser` class, with the unique ability to perform a 'spinning attack'. Since both these weapons have the spinning attack as a skill, they will both use the `SpinAttack` class to avoid code repetition (DRY).

This new weapon is also bought/sold by Merchant Kale, so it is aggregated into the `MerchantKale` class.