FIT2099 Assignment 1

Applied Session 4 Group 8

Design Rationale

Group Members:
Wee Jun Lin (32620861)
Kok Tim Ming (32619138)
Lau Ka-Kiat (32967136)

Table of Contents

Overview

Design goals

The main objective involves designing an extensible codebase around the existing Elden Ring game engine. To achieve this, design goals have been set to be faithfully adhered throughout the whole design and development process.

We strive to:

- Reduce code redundancy (DRY): Achieve through abstractions to reuse code. This will make the codebase more concise and easier to maintain.

- Avoid multiple inheritance (diamond problem): Done by abstracting methods through interfaces (ISP) or utilising composition.

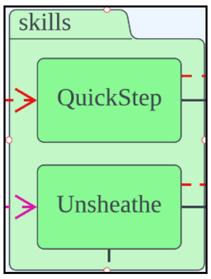
- Create an extensible codebase, accounting for future feature additions and development. Classes and modules should only be made to be extended (OCP).

- Follow the Interface Segregation Principle (ISP). This will prevent unused code and will make adding new features easier as no refactoring is needed.

- Follow Single Responsibility Principle (SRP): each class should only have one responsibility so that it is more focused. This will make the codebase easier to be extended with new functionality and easier to maintain.

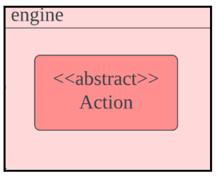
UML Notes

Newly added classes, and those expecting modification/extension to meet a specific requirement will be coloured in **green** 🟩🟩🟩. Colour contrast will increase along with package hierarchy depth.

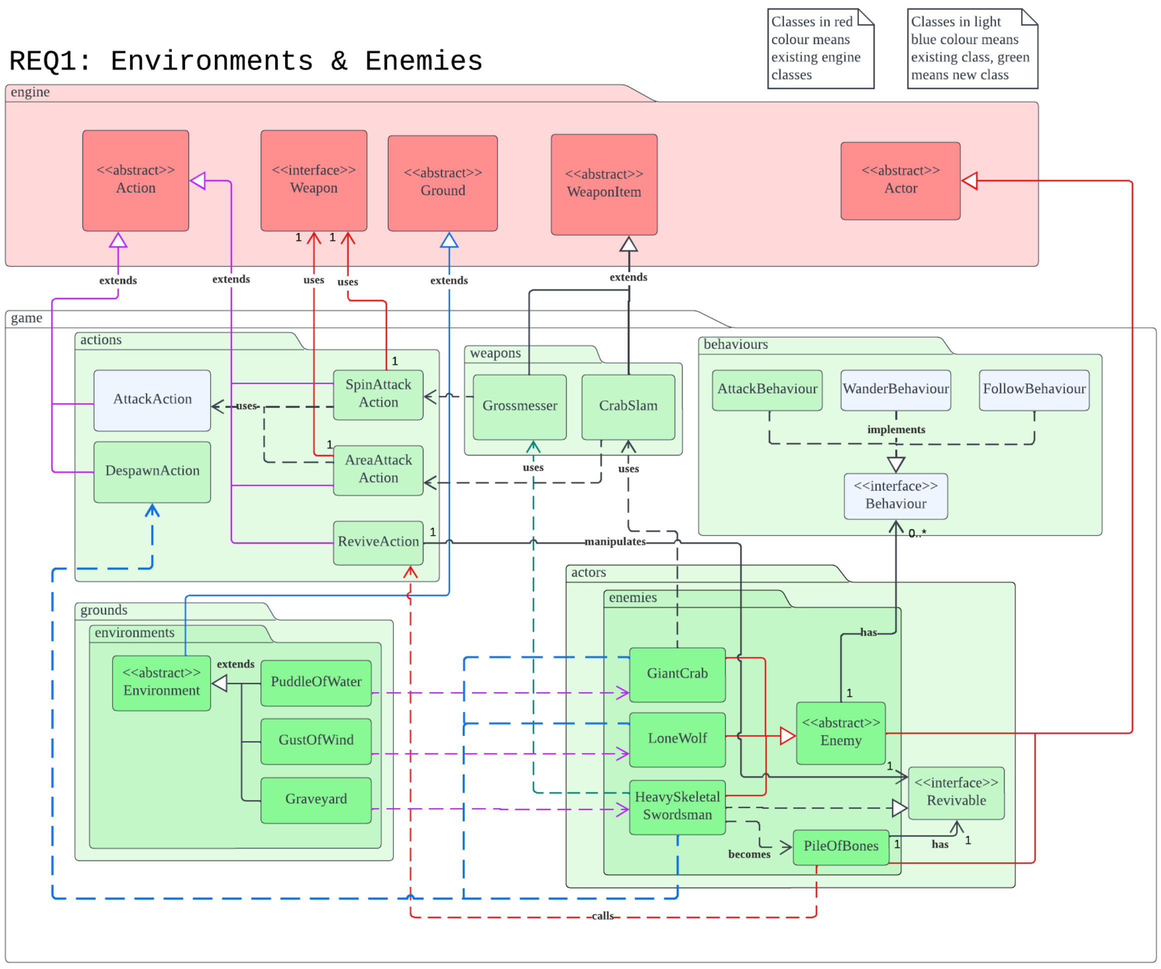


Classes coloured in **light blue** 🟦 are classes that will not be extended/modified from the previous requirement, or classes that have been provided. Classes coloured in **red** 🟥 are engine classes.

AttackAction



Requirement 1



Brief

The diagram represents an object-oriented system for requirement 1. It has 14 new concrete classes, 1 new interface and 2 new abstract classes, with all related classes in the game package categorised into respective packages.

Environments

All types of environments extend the *Ground* abstract class and implement the *Environment* interface as they share common methods (DRY).

Designing the *Environment* into an interface (using default methods) was also considered, but it was decided that the *Environment* can serve as a unique base class. Also, abstract classes could pass on instance variables if needed for future extensions.

Environments will spawn their corresponding *Actor*s through inherited methods and dependency injection, adhering to Dependency Inversion Principle (DIP). In the future if there arises the need for *Environment*s that cannot spawn *Actor*s, a *Spawnable* interface could be created to segregate the *spawn()* functionality (ISP).

Enemies

Enemy

An abstract class which will implement a method to get *Actions* derived from associated *Behaviours* & *allowableActions*. Since all enemies share common (code behaviour), enemies were designed to extend Enemy to reduce redundancy (DRY) and enforce code consistency through code refactoring.

HeavySkeletalSwordsman, PileOfBones, Revivable

HeavySkeletalSwordsman and *PileOfBones* were separated into different classes (SRP). *Revivable* interface will be implemented by *HeavySkeletalSwordsman*, indicating & forming a contract such that *HeavySkeletalSwordsman* is able to revive under its own defined conditions (ISP).

PileOfBones will store the *Revivable Actor* through dependency injection to retain the class instance (association), then call *ReviveAction* the *Revivable Actor*s (DIP). This allows future *Revivable* Actors be revived from *PileOfBones*. Furthermore, *PileOfBones* could also be classified and dissected into a *RespawnPoint* type in the future, improving extensibility.

A straight-forward alternative considered was to implement the whole process of turning into bones and vice versa into the *Actor* itself. This idea introduces several new responsibilities to be implemented and modified, potentially inviting severe maintainability issues like class bloat in the long run, violating OCP & SRP.

Weapons

Grossmesser and *CrabSlam* extends *WeaponItem* to have *SpinningAttackAction* and AreaAttackAction as a skill (OCP). This allows wielders to only have access when it is adjacent to a valid target.
CrabSlam was implemented as a *WeaponItem* instead of creating an altered class type similar to *IntrinsicWeapon*. This comes with no drawbacks, as weaponItems could be made undroppable.

Behaviours

All enemy behaviours implement the Behaviour interface, which ensures the implementation of the main method *getAction().* This abstraction allows for polymorphism, improving code maintainability. Also allows enemies to adhere to OCP, as the code in enemies will not need to be modified when new behaviours are introduced.

AttackBehaviour

Attempts to get a random valid target in surroundings and performs 50/50 attack/special attack.

Actions

DespawnAction, *ReviveAction*, *SpinAttackAction* & *AreaAttackAction* extends the abstract *Action* class to enforce common methods, allowing for polymorphism through abstractions (DIP).

DespawnAction

The *DespawnBehaviour* alternative was considered, but it seemed counter-intuitive as enemies do not have the intention to despawn.
DespawnAction will be made to be checked with an enum *Status* whether or not the enemy is following the *Player*, instead of directly implementing it into *FollowBehaviour* (violates OCP).
DespawnAction could also be used for future *Actor* removal requirements.

SpinAttackAction, AreaAttackAction

Basic *SpinAttackAction* and *AreaAttackAction* will depend on *AttackAction* as they are assumed to deal the same damage with the same accuracy (DRY).

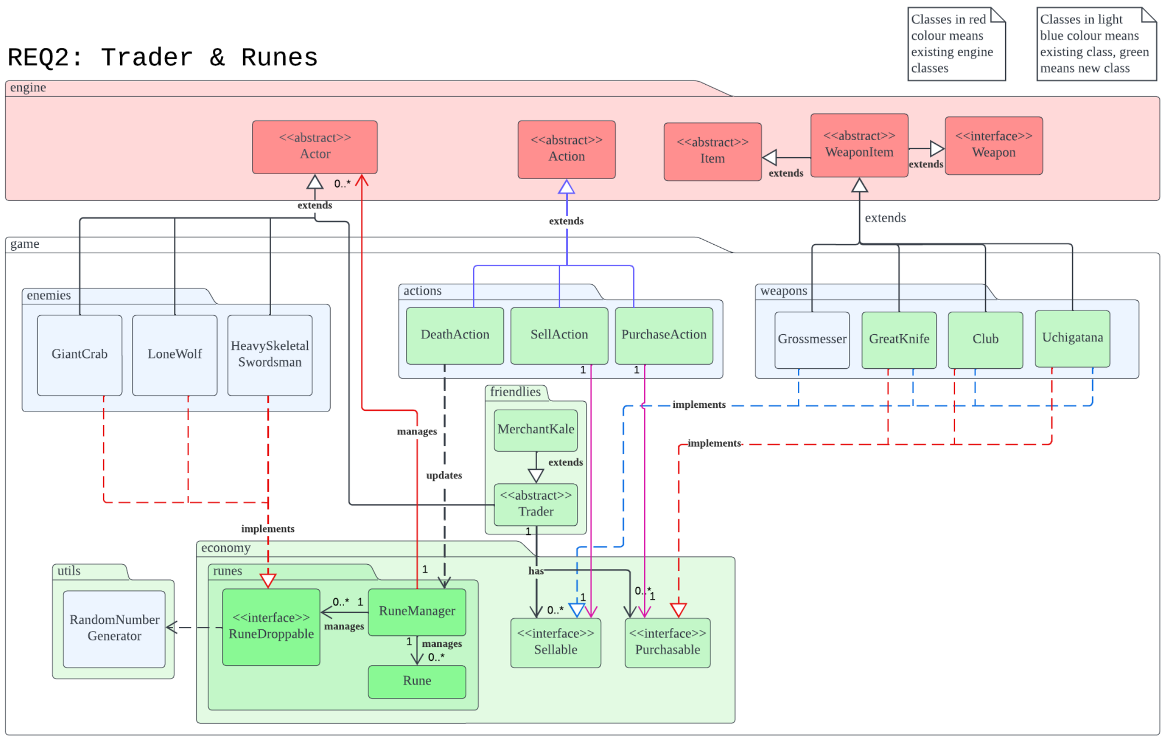
ReviveAction

ReviveAction will have an association with *Revivable* Interface. This allows *ReviveAction* to invoke the *Revivable* interface's method when executed, adhering to SRP and ISP in implementing-classes.

In requirement 1, *ReviveAction* will be responsible for replacing the *PileOfBones* with HeavySkeletalSwordsman (Revivable) if they are not killed within 3 game turns.

ReviveAction can also be used by other Revivable actors in the future, allowing for flexible extensions of custom revival features alongside other extensions (ISP).

Requirement 2



Brief

The diagram represents an UML class diagram for requirement 2. It has 9 new/modified concrete classes, 1 new abstract class and 3 new interface classes. All related classes to game economy, and Player-friendly Actors are categorised respectively into suitable packages.

Rune & RuneManager

Rune class extends *Item*. This is because it is allowed to be dropped when the *Player* dies. *Rune*s will have value as an attribute.

The *RuneManager* class follows a Registry & Singleton pattern, hence will store and handle all *Rune* related logic for registered classes. This should be done through a HashMap of *Actor* and *Rune* to track *Actor*'s corresponding *Rune*s, along with *RuneDroppable* to drop runes to tracked *Actor*s.

This approach provides a centralised control among *Rune*s, reduced coupling between other classes, allows for code reusability and simplifies object retrieval. On the other hand, it causes tight coupling between *RuneManager* and other dependent objects, and could promote the violation of SRP as the application extends (god class) if good coding practices are not exercised.

Alternatives like adding *Rune* to *Actor*'s itemInventory was considered, but that would require downcasting to retrieve the Rune Item functionality, and would require a significant portion of the classes to be modified to fully implement *Rune* functionality (violates OCP). Thus, given that *Rune*s were not an intrinsic part of *Actors* (as an attribute), and assuming that not only the *Player* can retrieve *Rune*s, it was decided that the Registry & Singleton pattern was the only choice to implement *Rune* functionality without maintainability drawbacks.

Enemies & RuneDroppable

RuneDroppable is an interface to be registered in the *RuneManager* and implemented by enemies that can drop runes upon death*.* When the *RuneDroppable* dies, it'll drop *Rune*s to the killer through *RuneManager*.

Friendlies (Trading)

Trader

A new *Trader* abstract class inheriting *Actor* abstract class was created for future *Traders*. They will have *Purchasable* and *Sellable* classes as an attribute, and retrieves respective *Actions* from them through a common method (DRY).

MerchantKale

Extends *Trader* to inherit common behaviors. *MerchantKale* will depend on *Sellable* & *Purchasable WeaponItem*s to allow purchasing & selling functionality.

Purchasable

An interface that requires *getPurchasePrice* and *getInstance* method to be implemented. Should provide a default *purchase* method that depends on *getPurchasePrice.*
An alternative for *getInstance* would be to create a class for producing *WeaponItem*s, namely *WeaponFactory,* which could be used to promote loose coupling if more *WeaponItem*s are added in the future.

Sellable

An interface that requires *getSellingPrice* and *getInstance* method to be implemented. Should provide a default *sell* method that depends on *getSellingPrice* and will remove corresponding weapon when invoked*.* If possible, *Trader* will compare *Actor*'s inventory and *Trader*'s inventory to provide valid *Sellable* weapons for the specific *Trader*.

Actions

SellAction, *PurchaseAction* and *DeathAction* extends the abstract *Action* class to enforce common methods, increasing code flexibility by allowing other methods to depend on the abstraction rather than concrete implementations (DIP).

SellAction

Returned from *Sellable*s. Stores the *Sellable* and calls *sell()* when executed, adhering to SRP and ISP in implementing-classes.

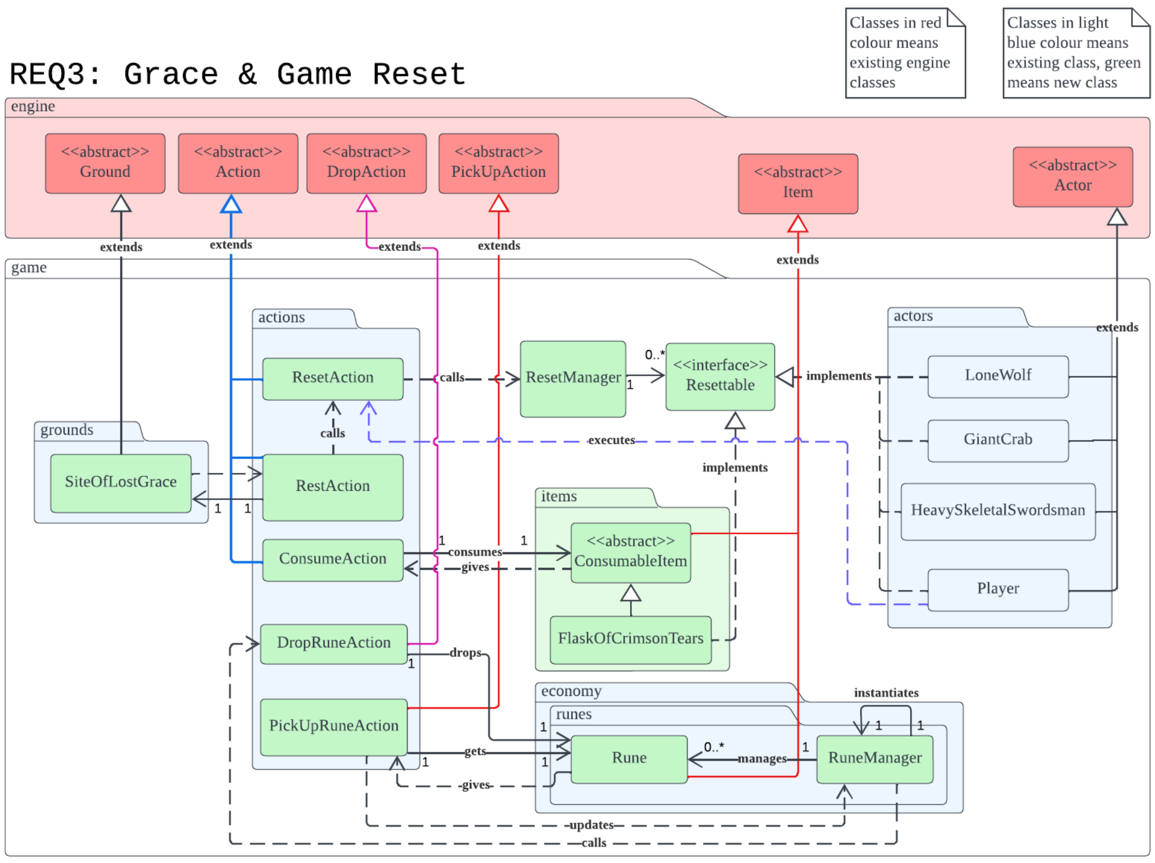
PurchaseAction

Returned from *Purchasable*s. Stores the *Purchasable* and calls *purchase()* when executed, adhering to SRP and ISP in implementing-classes.

DeathAction

DeathAction depends on the *RuneManager* class, transferring *Rune*s to killer from *RuneDroppable*.

Requirement 3



Brief

The diagram represents a UML class diagram for requirement 3. It has 10 new/modified concrete classes, and a new interface class and one abstract class, with all classes being in the game package, and organised into their appropriate subpackages.

FlaskOfCrimsonTears

A new class that inherits from the abstract *Item* class and the *ConsumableItem* class (see below).

ConsumableItem

An abstract class inherited by items that have a limited number of *useCharges*, enforcing the DRY principle through common methods like *consume()* ,*decrementCharge()* and attributes like *useCharges()*. This class will provide *ConsumeAction* to holder, utilising polymorphism (DIP) for this feature extension. The use of polymorphism also ensures that the Open-Closed Principle is not violated as no other classes are needed to be modified.

SiteOfLostGrace & Game Reset

SiteOfLostGrace is a new class that inherits from the abstract Ground class (DRY). An Actor (Player for now) adjacent to the *SiteOfLostGrace* is given *RestAction*, which executes *ResetAction* in turn.

Actions

ResetAction, *RestAction*, *ConsumeAction*, *DropRuneAction* and *PickUpRuneAction* extends the abstract *Action* class to enforce common methods, increasing code flexibility by allowing other methods to depend on the abstraction rather than concrete implementations (DIP).

ConsumeAction

ConsumeAction will call a *ConsumableItem*'s *consume()* method.

The *ConsumeAction* class will allow *Actor*s to *consume()* a *ConsumableItem* in their inventory when executed (e.g. *FlaskOfCrimsonTears*).

ResetAction & RestAction

ResetAction will *run()* the *ResetManager* class, which in turn will trigger the *reset()* methods of registered *Resettable* classes through abstractions.

RestAction will invoke *ResetAction* along with any other unique effects when the *Player* "rests" at a *SiteOfLostGrace*.

Runes

RuneManager will handle the logic of Rune dropping (on the ground) and when it is picked up. Reasoning links back to RuneManager in REQ 2.

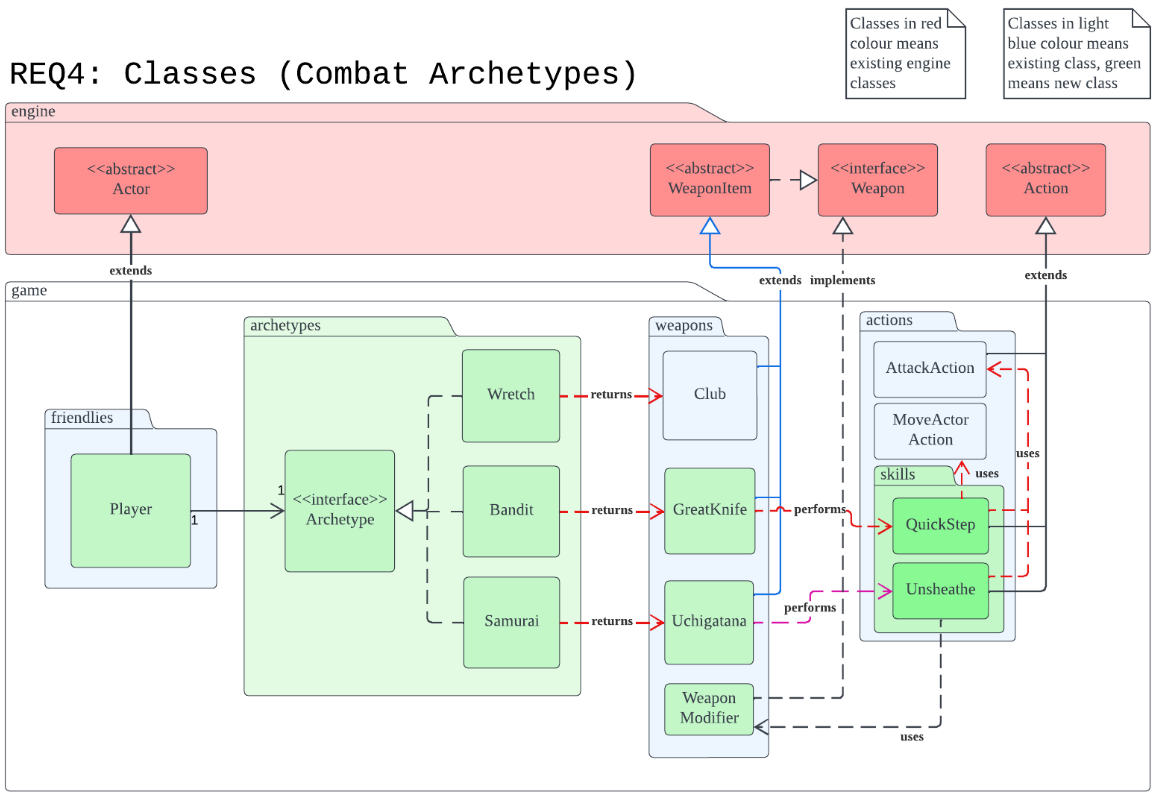
DropRuneAction

Following *RuneManager*'s Registry & Singleton pattern, *Actor*s should not hold *Rune*s in their inventory. Therefore, Actors invoke *RuneManager* which invokes *DropRuneAction* when dropping is necessary. For our case, *RuneManager* has the ability to drop the *Rune*s by itself, but it is more maintainable to let *DropRuneAction* handle the dropping logic (SRP).

PickUpRuneAction

Invokes *RuneManager* when *Rune* is picked up to update *Actor*'s *Rune* values.

Requirement 4



Brief

The diagram represents an UML class diagram for requirement 4. It involves 9 new/modified concrete classes implementing 1 new interface class for Archetypes, and 2 new concrete classes that extend Action for unique weapon skills.

Archetypes

Archetype was made into an interface so that archetypes like *Wretch*, *Bandit* and *Samurai* classes can implement common behaviour, allowing for greater extensibility as new archetypes can be added without further modifications to existing classes (OCP). *Player* will store *Archetype* as an attribute, as the requirement states that in the future, *Player* might be able to increase max HP based on the *Archetype*. This is designed so that will retrieve corresponding attributes and *weaponItem* from the *Archetype* through composition, promoting adherence to the SOLID principles.

Weapons

Club, GreatKnife, Uchigatana

The *Club*, *GreatKnife*, and *Uchigatana* classes extend the abstract *WeaponItem* class. They share common attributes as *WeaponItems*, so it is logical to abstract these classes to avoid code repetition (DRY). Open-Closed Principle (OCP) is also applied in this case, allowing for the creation of new *Weapon* classes without modifying any existing code.

WeaponModifier

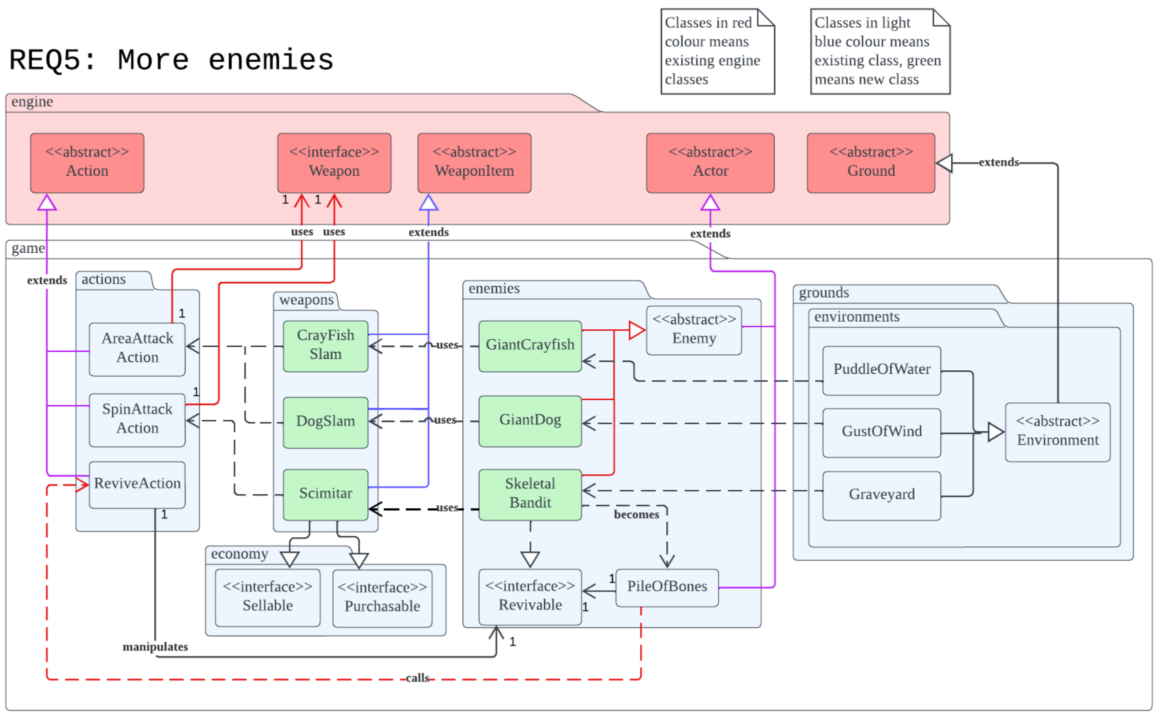
This concrete class implements *Weapon* interface to act as a substitute for *Weapon*s that temporarily require damage/accuracy modification, such as *Uchigatana*'s *Unsheathe*, hence increasing maintainability (OCP & SRP).

Skills

Quickstep, Unsheathe

These two are concrete classes that extend from *Action* to utilise polymorphism through abstractions. Both classes depend on *AttackAction* to reduce code redundancy (DRY) and promote maintainability when changes are needed.
For *Unsheathe*, *WeaponModifier* is used as a substitute to adhere to OCP.
For *Quickstep*, *MoveActorAction* is required for moving users (hence the dependency).

Requirement 5



Brief

The UML diagram for this requirement is quite similar to requirement 1 as this requirement only involves adding new enemies and weapons. There are 6 new concrete classes which are the 3 new weapons and 3 new enemies.

Enemies

Similar to requirement 1, enemies share common behaviour and thus inherit the *Enemy* abstract class for code reuse (OCP, DRY).

Skeletal Bandit

Similar to HeavySkeletalSwordsman in REQ 1, *SkeletalBandit* will implement the *Revivable* interface and will turn in *PileOfBones* through dependency injection without any modifications from REQ 1 (OCP).

Environment

Environment rationale is explained in requirement 1.
In requirement 5, the extension of map splitting could be added into the *Environment* abstract class (DRY).
If there rises a need for *Environment*s that don't spawn based on their *location*, an interface could be used to segregate the functionalities (ISP).

Weapons

Similar to requirement 1, *DogSlam* and *CrayFishSlam* extends *WeaponItem* to have *AreaAttackAction* as a skill (OCP). This allows wielders to only have access when it is adjacent to a valid target. They are implemented as a *WeaponItem* instead of creating an altered class type similar to *IntrinsicWeapon*. This comes with no drawbacks, as *WeaponItem*s could be made undroppable.

Actions

Actions will not be modified and are explained in requirement 1 (OCP)

Contribution Log link

FIT2099 MY (THU 10 AM) Assignments' Contribution Logs (https://docs.google.com/spreadsheets/d/1c-rrjRwR8Huk-2wwv6Zpmb_kAb5kch_zlCfuI5olHSo/edit#gid=0) , under the Assignment 1 tab.